



**POLITECNICO**  
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE  
E DELL'INFORMAZIONE

# A Machine Learning-based Security Architecture to Detect Microarchi- tectural Side-Channel Attacks in Microprocessors

TESI DI LAUREA MAGISTRALE IN  
COMPUTER SCIENCE ENGINEERING - INGEGNERIA INFOR-  
MATICA

Author: **Mattia Iamundo**

Student ID: 968591

Advisor: Ing. Luca Maria Cassano

Co-advisors: Ing. Alessandro Palumbo

Academic Year: 2021-22



# Abstract

Modern processors adopt some advanced technology, such as out-of-order execution, or the speculative execution, to improve their performance. Nowadays we put, day by day, an increasing amount of our personal data on the Internet, and this must be adequately protected. To succeed in such an objective the first line of defense is to encrypt the data, while they are transferred from one system to another one, and also while they reside on a system. In recent years a big challenge to such a defensive system comes from side-channel attacks, which can extract a secret key from a system by exploiting some weird behavior of the processor. In this thesis, we propose a hardware-based security module capable of detecting a side-channel attack, and signaling it to the Operative System, before the attack can extract anything. We focused our attention on cache-based side-channel attacks, that exploit the access pattern of the cryptographical algorithm to the cache to extract the key. The most famous attacks belonging to this category are Spectre and Meltdown. To assure a high level of precision in the detection, without the requirement to have a database of all the possible existing attacks, we based our system on machine learning, but opting for the algorithms belonging to the One-class classifier category. To test the presence of an attack, we monitor some relevant Hardware Performance Counters, that we narrowly chose among the events made available by the processor. After having conducted an extensive search of the Hardware Performance Counters, and having tested multiple ML algorithms, we find out that the best results are obtained using Isolation Forest which can reach an accuracy of 99% with all the cryptographical algorithms and the attacks we have tested it. In addition, such an algorithm has always less than 1% of false positives and zero false negatives. We have made also a hardware implementation of Isolation Forest, and we tried to compare its dimension with the one of an x86 processor. This last comparison was not easy, due to the lack of information about it, but we obtained an overhead that goes from 6.75% to 25.7%, without any sort of optimization that will be possible when you try to implement our module alongside a CPU.

**Keywords:** Embedded Systems, Hardware Performance Counter, Hardware Security, Machine Learning, Microarchitectural side-channel attack, Microprocessors



# Abstract in lingua italiana

I moderni processori adottano alcune tecnologie, come l'esecuzione out-of-order o l'esecuzione speculativa, per migliorare le loro prestazioni. Oggigiorno mettiamo su Internet sempre più dati personali, che devono essere adeguatamente protetti. Per raggiungere questo obiettivo, la prima opzione è crittografarli, sia quando vengono trasferiti da un sistema all'altro, sia quando risiedono su un sistema. Negli ultimi anni un'importante minaccia alla crittografia è rappresentata dagli attacchi side-channel, che sono in grado di estrarre una chiave segreta sfruttando un comportamento ambiguo del processore. In questa tesi proponiamo un modulo di sicurezza hardware in grado di rilevare un attacco side-channel e di segnalarlo al sistema operativo, prima che questo sia in grado di estrarre qualcosa. Noi ci siamo focalizzati sugli attacchi side-channel cache-based, che sfruttano il pattern di accesso alla cache, dell'algoritmo crittografico, per estrarne la chiave. Gli attacchi più noti di questa categoria sono Spectre e Meltdown. Per garantire un elevato livello di precisione nel rilevamento, senza la necessità di avere un database con tutti i possibili attacchi esistenti, abbiamo basato il nostro sistema sul Machine Learning, optando però per gli algoritmi di classificazione di tipo One-class. Per valutare se c'è un attacco, monitoriamo alcuni Hardware Performance Counter, che abbiamo scelto attentamente tra gli eventi resi disponibili dal processore. Dopo aver condotto una ricerca approfondita sugli Hardware Performance Counter, e aver testato diversi algoritmi di ML, abbiamo scoperto che Isolation Forest dà i risultati migliori, raggiungendo un'accuratezza del 99% con tutti gli algoritmi crittografici e gli attacchi che abbiamo testato. Inoltre, questo algoritmo ha sempre meno dell'1% di falsi positivi e zero falsi negativi. Abbiamo realizzato anche un'implementazione hardware di Isolation Forest e abbiamo cercato di confrontare le sue dimensioni con quelle di un processore x86. Tale confronto non è stato facile, a causa della mancanza di informazioni al riguardo, ma abbiamo ottenuto un overhead che va dal 6.75% al 25.7%, questo senza alcun tipo di ottimizzazione che invece sarà possibile quando si vorrà implementare il nostro modulo insieme ad una CPU.

**Parole chiave:** Attacchi side-channel alla Microarchitettura, Hardware Performance Counter, Machine Learning, Microprocessori, Sicurezza Hardware, Sistemi Embedded



# Contents

<b>Abstract</b>	<b>i</b>
<b>Abstract in lingua italiana</b>	<b>iii</b>
<b>Contents</b>	<b>v</b>
<b>Introduction</b>	<b>1</b>
<b>1 Background</b>	<b>9</b>
1.1 Caches . . . . .	9
1.1.1 Cache Flushing . . . . .	12
1.1.2 Cache Partitioning . . . . .	12
1.1.3 Cache Access Randomization . . . . .	14
1.2 Microarchitectural Side-channel Attacks . . . . .	14
1.2.1 Time driven attacks . . . . .	15
1.2.2 Access driven attacks . . . . .	16
1.2.3 Spectre . . . . .	17
1.2.4 Meltdown . . . . .	19
1.3 Machine Learning . . . . .	21
1.3.1 Local Outlier Factor . . . . .	22
1.3.2 Support Vector Machine . . . . .	23
1.3.3 Isolation Forest . . . . .	24
1.4 Gem5 . . . . .	25
<b>2 Related Works</b>	<b>27</b>
2.1 Machine Learning-based detectors . . . . .	27
2.2 Non ML-based detector . . . . .	30
2.3 Novelty of the proposed thesis . . . . .	32
<b>3 The Proposed Security Architecture</b>	<b>35</b>

4	Exploring ML for Attack Detection	41
5	Hardware Implementation	55
6	Security Analysis	59
7	Conclusions and future developments	61
	Bibliography	63
A	Gem5 Simulator	69
	List of Figures	75
	List of Tables	77
	Acknowledgements	79



# Introduction

With Information Technology (IT) systems and technology in general, being more pervasive day by day and processing more and more personal data, is our responsibility, as the scientific community, to guarantee the highest possible level of security to these systems. Cryptographic algorithms are an essential component, to be able to ensure confidentiality, integrity, and non-repudiation of communication channels and the data transmitted over them. The algorithms that are actually in use are chosen because the mathematical theorems on which they are based are considered robust to an attack, but the mathematical properties aren't the only possible attack surface. Another vulnerable side of these algorithms is their actual implementation, that can expose the key used for the encryption and decryption operations directly, due to some bugs in the code, or indirectly, through some unexpected behavior of the hardware on which those algorithms run.

Modern processors, to be able to increase their performance, cannot still count only on higher clock frequencies and the reduction of the dimensions of the components, so CPUs manufacturers studied and introduced some technologies such as: speculative execution and out-of-order execution. Speculative execution aims to reduce the time needed for the execution of a program, and to do so it starts to fetch some instructions before having all the needed information. The technique is focused on improving the CPU's efficiency in branching situations, where, to determine which is the correct next block of instructions to be executed, a certain boolean condition must be solved. Before the introduction of this technology, the fetching activity was blocked until the condition is solved and the correct following instruction is determined. Instead, with speculative execution, the CPU makes a prediction on which will be the next instruction and start to process it. When the condition is solved, if the prediction was correct, the processor saved time, otherwise, it reverts all the changes it has made and fetches the correct instruction. Out-of-order execution was implemented to improve the CPU's execution efficiency. Every single core of a processor has multiple units that can perform different arithmetical and logical operations, but only a portion of these are used simultaneously because of bottlenecks in other parts of the system. One of the bottlenecks is the time needed to access the memory, even if the data are in the cache, another example is the

execution of a complex operation such as multiplication. In these cases, the CPU must stop the execution of the following instructions until the current one finishes. With out-of-order, the processor has the capability to fetch and execute instructions in an order that is different from the original one, if there is an available execution unit and all the data needed by the instruction are ready. Even if the instructions are not executed in order, the program behavior is not modified because when an instruction is completed, it goes into a buffer, called Re-Order Buffer (ROB), and it is signed as completed only when all the preceding instructions are signed so.

One of the most recent attacks family goes under the name of side-channel attacks, such attacks do not directly involve the code of the victim program they target, but exploit the normal behavior of a part of the hardware (the CPU typically), such as power consumption or cache access timing [21, 50] to correlate those behaviors to the secret data the attacker wants to obtain.

A first division can be made between the side-channels that require physical access to the system, to be able to take the measurements required to conduct the attack, and those ones that can be performed from remote. In the former group, reenters attacks that use information about the power consumption [13] or the electromagnetic emissions [34, 42] of the processor. To acquire such data is required that the attacker is physically near the system, or, in the case of power analysis, he/she must have direct access to the CPU to insert some probes in it. To put in place these attacks is also necessary to use some laboratory equipment, like oscilloscopes, and perform preliminary sampling to correctly tune the attacking program, this requires having access to a CPU that is identical to the one you want to exploit. In addition is necessary to make a survey on where the acquisition equipment must be placed, as suggested in [34], to have the best possible results. For attacks based on power analysis, as explained by Brier in [9], there is the necessity to do a brute-force style attack, in fact, given a known plaintext and a power consumption, you have to test all the possible keys to find the one that correlates better with the measured power consumption.

The second group, instead, comprehends the attacks known as microarchitectural side-channel attacks (MSCAs) that exploit how the CPUs work to obtain otherwise secret data. One of the most attacked processor's component is the cache, whose access pattern is analyzed and used as a side-channel from which infer the value of a secret data [21, 27, 31, 50]. This gives the possibility to conduct the attack from remote, and in addition it doesn't require elevated privileges to run, in fact such malevolent programs need only user level privileges. The exploited behavior in this case is that the time needed to read a cache line is correlated to having a cache miss or a cache hit. If you put the cache in a

known state, every unexpected access timing gives you a hint on what the program you are attacking have done. For example, if you write into the cache some known data, and than let the victim program execute some instructions, and finally read again what you initially wrote, every cache miss indicates that the victim program accessed that line.

The cache-based side-channel attacks, to reach their goal, exploit the fact that the cryptographic algorithms access the cache in a manner that depend by the value of the bits of the key, and so, observing the cache access pattern, and knowing its correlation to the key value, is possible to retrieves such key. These attacks cause a huge security fault, especially because they are very difficult to identify due to the fact that they do not modify the functioning of the system. All the proposed cache-based MSCAs have the same general structure. The first phase put the observed cache lines in a known state, in order to have a baseline for the following measurements, at this point it waits that the targeted program execute for a short amount of time (this aspect isn't under the direct control of the attacker). At this point, the second phase starts and the access times to the monitored cache lines are taken and compared to the expected ones.

One of the most known attack of this class is Spectre [27] that, exploiting the speculative execution of the CPU, can force the victim program to make an out-of-bound access to an array, then extrapolate the information associated to this access with some covert channel such as Flush+Reload [50]. This is possible because the index, used to make the out-of-bound access, points to the secret data targeted by the attacker. This data is then used as an index to access a second array, which will load some data into the cache, and so there is a correlation between the secret data targeted by the attacker and the cache line accessed by the victim program. The extracted information can be some secret data of the application or some private key used for encryption/decryption functions. Another variant aims to mistrain the Branch Target Buffer (BTB), a unit responsible for making predictions on the target address of a branch instruction. In this variation, the attacker identify a gadget, that is a small bunch of instructions that can be useful for the attacker, in the victim address space, and he misleadingly trains the BTB that a certain branch goes there. When the attack is put in place, the BTB redirects the program execution towards the gadget, and, even if the CPU will revert the changes, when the misprediction will be detected, the cache will still contain the data the gadget has accessed.

Another famous cache-based side-channel attack is Meltdown, which can have access to all the memory space. It can reaches its objective forcing the access control mechanism typically adopted by processors, exploiting the out-of-order execution and the fact that the exceptions are managed at the end of the of the pipeline. The CPU will load the page requested by the attacker into the cache before checking if the process is authorized to

read it or not. When the processor will realize that a non authorized access was made it rise an exception and goes back to a previous safe state. But at this point it is too late, because the attacker can simply read the cache lines where the memory elements were loaded and have access to restricted data. This is possible because such cache lines wasn't cleaned during the revert phase. Here the bad behaviors used are two, the first one is the fact that the exception isn't managed when it is raised, and the second misused behavior is that the cache isn't cleaned after an exception.

There is a lot of effort in the research field trying to reducing this problem. There exist two possible ways to approach it, you can act proactively and solves it at the origin, modifying the algorithms or the way of working of the processors, otherwise you can try to detect the side-channel attack as soon as possible and block the executable that is conducting it. The proposed solutions which go under the first category consist of implementing some algorithms that make the cryptographic operations time/emissions independent[30] from the value of the key that is used in the cryptographic operations. Otherwise, if you want to act directly on the hardware, another idea is to change the way of functioning of the cache [26, 39, 52], for example, increasing its isolation from a process to another one [45] so that the attacker cannot access the cache lines of another process. A further possibility is to make less accurate the internal timer of the CPU so that the attacker will obtain unusable time measurements and cannot distinguish if the monitored cryptographic function involved a 1-bit or a 0-bit. For what concerns the latter research area, which tries to develop some detection systems, the vast majority of proposed solutions are software-based [1, 4, 11, 28], and they focus their attention on searching some source of information that can be a good and trustworthy indicator for the presence of an ongoing attack.

A source of information, often used by the detection systems, are the Hardware Performance Counters (HPCs). These counters was originally designed for debugging purposes. They can be set, through a specific instruction, to monitor a specific event in the plethora of available ones. For what concern Intel, the events that can be monitored are divisible into two category: architectural and non-architectural. The former category refers to those events available to all the CPUs independently from the processor's family the specific CPU belongs to, while the latter category groups together events that are family-specific. For this reason Intel provides a list with all the supported non-architectural events, divided per processor family, which is available here [25]. The events are related to the functioning and the performance of the processor itself, they include information such as the number of cache hits or the number of completed instructions. Once a counter is set, it starts tracking the number of occurrences of the selected event, storing it in a specialized

register. The CPU provides instructions to read the value stored in the HPCs, and also to reset the counter. According to the Intel manual (Volume 3B, Chapter 19) [24] there exist two possible methods to retrieve the value stored inside such counters, the first one is to use the dedicated instruction *rdpcm*, the other possibility is to set the counter to rise an exception when it overflows a certain threshold. It is important to make the right choice on which events monitor because the amount of these registers is quite limited, in fact, Intel gives only four registers per thread on its CPUs [24], while on AMD processors we can have access to six registers per thread [2].

With this thesis we have focused on cache-based access-driven microarchitectural side-channel attacks, those that, for extracting secret information, use techniques such as *Flush+Flush* [21] or *Prime+Probe* [39]. These types of attacks, measuring the time taken by the attacking program to access some specific cache lines, can deduce which of those lines have been accessed by the cryptographic algorithm. Thanks to the fact that the accessed cache lines have a correlation with the value of the key, even if the algorithm is patched to remove this correlation, accessing all the cache lines [38, 51], is possible to retrieve part or the whole encryption key. We decided to develop a hardware-based on-line detection module that, thanks to the monitoring of some carefully selected Hardware Performance Counters, can rapidly detect an ongoing attack and signals it to the operative system that will decide which counteraction is the most suitable for the actual scenario. Several other similar works used the HPCs [4, 47], but they decided a priori which counters to use, we, instead, performed a wide search to find the best ones. To explore the widest possible number of HPCs we used *gem5*, a clock-cycle accurate architectural simulator, that gives us the possibility to collect at the same time all the values of the events made available by the simulator. After having eliminated all the events not directly related to the CPU or its cache, we have left with nearly 800 counters, and later we chose the most suitable for the detection of a side-channel attack. This was possible because *gem5* not only offers the possibility to simulate a CPU in a cycle accurate manner, but you can also mount and use a working Linux system on it.

To actually make the classification of a given sample and understand if it is derived from a normal execution of a cryptographical algorithm or it comes from an execution that is under attack, we based our system on some machine learning techniques. This choice was made because this technology offers the possibility to identify the schemas that are common to the samples that come from the same origin, allowing us to distinguish between benign samples and those generated from an attacked environment. For what concerns the choice of which machine learning algorithms use, we have focused on those ones that can be trained using only samples coming from an execution that is not compromised by

a side-channel attack, this machine learning technique is called One-class classifier. This choice is due to the fact that these samples are simpler to obtain and, otherwise, to have a reliable system, we not only have to collect samples from an environment that is under attack, but also we need samples associated to all the possible existing attacks and to those not yet developed ones. That requirement of course is impossible to be satisfied, and less types of attacks you have and worse will be the classifier, instead, if you train the ML algorithm only with samples coming from a plain execution of the cryptography algorithm is possible to forget about the attacked samples.

Thanks to the use of the scikit-learn library, a well-known and used python library for machine learning, we had the possibility to test three different algorithms of this type: One-class Support Vector machine (SVM) [5, 43], Local Outlier Factor (LOF) [8] and Isolation Forest [33]. After having tuned their parameters to obtain the best possible results, we went to a comparison phase to understand if one of them has better performance respect to the others.

The scenario in which we want to operate is the one of embedded devices, powered by an x86 processor, that are specifically dedicated to encryption/decryption activities and work with just one algorithm at a time, in fact we developed a different configuration for each of the tested algorithms. During our research, we find out that the best HPCs to use are not always the same, but they differ not only between different cryptographic algorithms, but they even change depending on which machine learning algorithm you use. This choice on the scenario is done in order to make simpler the testing environment, but steel maintaining it realistic. Such protection system is really helpful in a product which the only objective is encrypt and decrypt messages making worthy the cost of developing and integrating such module into its CPU. Instead, in heterogeneous systems, like a PC, this cost can be less justifiable. Anyway exist some works that propose a program, based on the usage of HPCs, that can detect attacks also in a environment where the cryptographic algorithm isn't the only program that run [10, 29]. So, given the existence of these researches and the fact that we perform a wide analysis to select which performance counters to use, is reasonable to think that our solution can be extended even to general purpose systems.

After having identified the top four HPCs, for each of the combinations cryptographic algorithm, ML model, and discovered which machine learning algorithm performs the best, our final goal is to implement this machine learning algorithm in hardware and create a module that can be put on the side of a CPU, without the need of modifying it. The only scope of such module is monitoring the system and signal to the Operative System (OS) potentially malevolent behaviours. This choice has the disadvantage that it

can be applied only to new CPUs, and so it is not immediately deployable, but, being a dedicated piece of hardware, it does not affect the performance of the system that otherwise has to dedicate some resources to the monitoring program. In addition, a software solution it is more prone to be a victim of an attack directed to stop its service or to alter its functioning with the objective of hiding a side-channel attack.

Our module has also the possibility to change its configuration, namely the HPCs used and the parameters of the machine learning algorithm, but not the ML algorithm used. The reconfigurability is fundamental to ensure a protection to the different cryptographic algorithms that can be used by the system we want to protect, and making the module extensible and capable of defending future algorithms.

We made various test using some known cryptographic algorithms such as: AES, RSA or blowfish, all of them attacked using an ad-hoc modified version of Spectre's Proof of Concept (POC), and also a modified version of a Meltdown's POC. After a tuning of the parameters used by the machine learning algorithms, we were able to achieve a detection accuracy of 99% with all the datasets, with zero false negatives and just a few false positives.

### **Thesis contribution**

To summarize, with this work we achieved the following steps-forward, with respect to the other works done, for what concern the detection of the microarchitectural side-channel attacks that exploits the cache access pattern to extract a secret key from a cryptographic algorithm:

- A wide search to identify the most suitable HPCs to identify an ongoing attack. Such a search was conducted on a per cryptographic algorithm and per ML algorithm basis.
- The exploration of several machine learning algorithms aimed at identifying the most suitable one from the point of view of both the classification accuracy and the hardware implementation cost.
- The usage of One-class classifier to make the detection system independent from the specific attack used, giving it the ability to detect both known and unknown attacks.
- A small and efficient hardware-based module to be interconnected with the CPU that implements an ML algorithm with a reconfigurable set of parameters to be able to protect different cryptographic implementations.

The thesis is organized as follow: in Chapter 1 is presented the background; in Chapter 2 we focus on the previous works done on side channel attacks detection; Chapter 3 introduces, at an high level, the architecture of the proposed detection system; Chapter 4 presents the steps we have followed to acquire the samples needed for the tests, the work done exploring the various ML algorithms, in order to find their best parameter's configurations, and the relative results obtained through a simulative evaluation; Chapter 5 contains the description of the actual hardware implementation we have done, of the best performing ML algorithm, with also a rough comparison of its area occupation with respect with a couple of x86 processors; Chapter 6 is dedicated to the security analysis of our proposal, to highlight its possible faulty points; Chapter 7 contains the final conclusions and some reasoning about which could be possible future works.



# 1 | Background

In this chapter will be presented all the basic notions that you have to know to be able to have a clear idea about the addressed problematic, the tools, and the techniques used to find a solution to it.

## 1.1. Caches

Modern processors can make logical and arithmetical operations very fast, but they are mainly limited by the memory access time. To move out this bottleneck the CPUs designers introduced a new level of memory between the RAM and the processor's registers called cache. This memory, in turn, is subdivided into multiple levels, typically two or three. The cache memory is located in the CPU package to keep it as close as possible to where the data contained there will be used. What differentiates the various levels isn't only the dimension of the cache and the access time to it, but also the visibility that the various cores of the CPU have on it. In a common configuration Level 1 and Level 2 caches are core private and then there is the Level 3, or also called Last Level Cache (LLC), that is common to all the cores.

Caches are made up of lines, which represent the minimum accessible unit that can contain multiple words from the memory. To determine the cache line where to put a certain word from the main memory, the last bits of its address are used; different cache lines can be grouped together to form a *way*. Unless some protection techniques are applied, by default a cache line is overwritten only when needed but is never cleared when, for example, there is a context switch, leaving there all the data.

Write operations on the cache can be managed in different ways, but first we need to distinguish between how the actual write is done from the way a write miss is managed. For what concerns the write policies, we have two of them: *write-through* and *write-back*.

***Write-through***: This policy, when it needs to write a new value to an address that is cached, makes the write to both the cache line and also in the main memory. This strategy introduces a quite high latency because the write operation is considered done only when

the new value is stored on both the cache and the main memory, and so the time to complete it is the time required for a write in the main memory. The advantage of this solution is that it doesn't introduce any data inconsistency because the write happens synchronously to all the places that contains the data.

**Write-back:** the new value is written only in the cache level nearest to the CPU. The modified value is then updated also in the main memory only when the cache line is evicted. This approach has the advantage that if multiple writes are made on the same address, only the last one will be written back on the main memory, saving time, because a write in the cache is faster. Such policy however introduce some complications to the structure of the cache, in fact its lines need an extra bit, to keep track if a write operation was done or not. This *dirty bit* is then used when the content of a line must be substituted, if the dirty bit is set to 1 the line's value must be written back to the main memory. This situation causes a double access to the memory, one for the write back and another to read the data which caused the cache line eviction. The consequence of this double memory access is that the read is retarded by the write, increasing the latency of the read miss. A possible mitigation is to use a write buffer that enqueues the write-back operations so that the read can start immediately.

The second group of write policies defines how the cache must operate in the case of a write-miss, even here we have two possibilities: *write allocate* and *no-write allocate* (also called *write around*). To have a working cache you need to define a write policy and a write-miss policy. You can pick any of the possible combinations, but typically is one of the following two possibilities:

- *write-back* with *write allocate*
- *write-through* with *write around*

**Write allocate:** On a write miss, the data to be modified will be loaded into the cache and then are modified in accordance with the adopted write policy. This strategy gives advantages when multiple subsequent writes are done to the same address, in fact, after the first write, that load the data into the cache, the other ones can be done directly in cache. It is then evident why this write-miss policy is associated with *write-back*, because otherwise on successive writes we should need to write again in the main memory, nullifying the advantage of caching the data.

**Write around:** When a write miss is encountered, this policy writes the new value directly into the main memory, hoping it will not be read in the near future. This policy makes the opposite assumption with respect to *write allocate*, it supposes that there will

be no further writes or reads in the near future. If such assumption turns out to be true we avoided to uselessly occupy a cache line, and this is quite important given that their number is limited, so we must choose carefully which data load into the cache.

The cache is smaller than the main memory, so we need a way to map a memory address to a cache line, in the years three strategies emerged to manage such mapping: *direct mapped cache*, *set associative cache* and *fully associative cache*. All of these mapping strategies subdivide a memory address in the same way. The lowest bits are used as an offset index; in fact a cache line can contain multiple words; assumed that a cache line can have  $x$  words, then the lowest  $\lceil \log_2 x \rceil$  bits are used as a line offset. Then we have the index bits, and, as for the offset ones, their number depends on how many lines/sets the cache has, assumed that we have  $x$  lines/sets, then we need  $\lceil \log_2 x \rceil$  bits. All the remaining bits compose the tag.

***Direct mapped cache***: this is the simplest strategy that can be adopted. Each memory address is associated with exactly one cache line, and the line number corresponds to the value of the index bits of the memory address. This type of memory needs to have a big size to prevent continuous cache conflicts, otherwise it's highly probable that two addresses have the same index and evict each other.

***Set associative cache***: This mapping technique tries to reduce conflict probability by grouping together multiple lines to form a so-called *set*, each line of the set is called a *way*. The cache takes its name from the number of ways its set is composed, if it has  $k$  ways, then the cache is a *k-way set associative cache*. With this strategy, the index is used to determine only the set where the data must be loaded, and the cache line is chosen with the use of a replacement policy. The number of sets available is computed as  $k/n$  where  $k$  is the number of chosen ways per set, and  $n$  is the total number of lines in the cache, of course, such numbers must be chosen so that the number of sets is an integer.

***Fully associative cache***: This is the exact opposite of direct mapping; with such a strategy, every memory address can be loaded into any cache line, in fact the index length is zero bits. If there are some free cache lines, a random one is selected; otherwise, a replacement policy will be used. The advantage of this mapping strategy is that we maximize the usage of the cache, and, in the meanwhile, we minimize the possibility of having a cache miss. On the other hand, a so flexible strategy has the disadvantage that it increases the access latency, because, both for searching an empty line and for searching a cached data, there is the need to do a complete search in the cache space, and this takes time. A possible mitigation, to reduce the latency, is to make a parallel access to all the lines of the cache, but this takes in the disadvantage of increasing the size of the cache,

mainly due to the hardware required for the comparisons.

When comes up the necessity of selecting a cache line to evict, there are multiple replacement policies, and we will present a selection of the most frequently used. *Random replacement*, as the name suggests, will choose randomly which line to evict, between the ways of the set, in case of a set-associative cache, or any line in the cache if we have a fully associative cache. *First in First out* that replace the oldest entry, other queue policies can be adopted. *Least recently used* which evicts the cache line whose last access was the least recent one.

There are different proposals to increase the security and the privacy of the data of the caches, a collection of them can be found in these papers [14, 36, 40]. Going into more details, the most suggested solutions comprehend: cache flushing, cache partitioning, and access randomization.

### 1.1.1. Cache Flushing

Under the category of cache flushing, we have two possible solutions, one consists of removing the instruction that gives the possibility to clear the cache because it is a key component of many attacks [21, 50].

The other solutions under this category propose to flush the cache, which level/s to flush depend on the solution, and represent a trade-off between security and time needed to complete this operation. Zhang et al. [53] suggest cleaning periodically only the Level 1 cache. In other works, such as the one by Godfrey et al. [20], is preferred to clean all the cache levels, preferably using a dedicated instruction, if the ISA offers it. The advantage of using a dedicated instruction is that it makes the flushing in an efficient way. Otherwise, they propose an alternative solution consisting of loading into the cache a block of memory big enough to fill the Last Level Cache; in this way every successive access to any of the cache lines will result in a cache miss.

### 1.1.2. Cache Partitioning

The main idea under this family of solutions is to block the sharing of the cache between different processes.

Under the hypothesis that the attack program and the victim one run on two different threads, and that they share the same cache, an easy solution is to disable the Hyper-threading, as proposed by Percival in [41]. This has a significant impact on the overall performance of the CPU, and even more, some more recent attacks take advantage of the

Last Level Cache making possible to mount a cross-core attack without the need of relying on Hyper-Threading. Especially for cloud environments, where multiple VMs are located on the same machine, is recommended to disable the page-sharing. This technology gives the possibility to share a memory page between different processes, or even VMs instances if the page is the same between the accessing processes. A possible solution is represented by CacheBar [54], which is a memory manager that prevents the sharing of a memory page between different instances of virtual machines. To increment cache security Intel proposed CAT "*Cache Allocation Technology*" that makes it possible to set some cache lines as non-replaceable by other programs. This technology is used in another cache managing system, CATalyst by Liu et al. [32], that subdivides the LLC into two separate parts, one used as a normal cache and the other is intended to be a protected cache. The latter will contain sensitive data, such as AES S-boxes, whose cache lines can be accessed only by the process to which those lines were allocated. An additional proposal is to use the so-called cache coloring. This consists in changing how the physical address is interpreted, the overlapping bits between the memory address, excluded those associated with the page, and the bits of the associative set of the cache are called color bits. As you can see in figure 1.1, there are five overlapping bits, giving us the possibility of having 32 colors.

32 bit addresses, 2MB cache, 16-way associative, 64-byte offset				
Memory Term		Colour	Page, 4KB	
Memory Address	0000000000000000	00000	000000	000000
Cache term	Tag	Associative Set #2048		Offset (64 Byte)

Figure 1.1: Address mapping from main memory to cache, image re-adapted from [20]

The idea proposed in [20] is to use cache coloring as a cache partitioning technique. Their proposal consists in assigning a different color to each VMs, in this way they cannot share the same cache lines. This strategy can be extended to a local system giving a color exclusively to a cryptographic algorithm, or whatever process you want to protect, so that no other process can access their cache lines, making it impossible to conduct a cache-based side-channel attack.

### 1.1.3. Cache Access Randomization

A third modification to the cache functioning, to prevent side-channel attacks, proposes to introduce randomness in the cache accesses so that the attacker cannot predetermine which lines will be accessed by the victim program and monitor them. Such a solution is presented by Wang and Lee in [48]. The RPcache (Random permutation cache) works in the following way: instead of having a direct mapping between memory and cache exists one, or more, permutation table (PT) that acts on the bits of the memory address used to identify the correct set of the cache to be used. Such a table scrambles the previously mentioned bits so that if originally the address  $k$  goes to set  $\mathbf{S}$  and address  $i$  goes to set  $\mathbf{S}'$ , with the permutation we have  $k \rightarrow \mathbf{S}'$  and  $i \rightarrow \mathbf{S}$ . The number of permutation tables used depends on the system that you must protect. For example, to secure a common PC you just need one PT, that will be used by the cryptographic algorithms, maybe regenerating it periodically, and for all the other programs running on the system, you can simply use the direct mapping.

## 1.2. Microarchitectural Side-channel Attacks

Exists a category of cyber attacks capable of extracting private information from a system indirectly, searching for and later making use of some correlation between the data you want to extrapolate and the behavior that the system has when manipulating or using these information. These attacks, in fact, are connected to the functioning of the hardware components, and this makes them harder to be identified and eliminate. Due to the complexity of modern hardware and its life cycle, it is also complex to eliminate such weird behaviors from the processors. Normally takes years before someone change the components of his/her system, and this without even taking into consideration the time needed to project a new CPU with all the protections needed to prevent such attacks.

In the past this class of attacks wasn't considered so realistic because the proposed channels of attack requires the physical proximity of the intruder to the victim system. Nevertheless exist researches that positively explored the possibility to use the electromagnetic emissions of a cryptographic device [22, 42] to extract a cryptographic key. This attack, after having conducted an appropriate mapping, is able to infer the value of the cryptographic key thanks to the fact that the values of the electromagnetical emissions depend on the operations done by the device and ultimately by the value of the bits of the key. A similar alternative suggests to find a correlation, in order to extract the key, by analyzing the power consumption [13] of the attacked processor. As the other one this requires a physical access to the system to be able to take the required measurements.

A new category of side-channel attacks was identified in the microarchitecture of the CPU, and so they are called microarchitectural side-channel attacks (MSCAs). To be more precise, this other side-channel was identified in the cache of the processors. This was a big step forward because makes possible to conduct such type of attack from remote, these subtypes of side-channel attacks in fact do not require to physically measure something. They can be based only on software, that analyzing the usage of the cache (e.g. measuring the time needed to access it), is able to determine directly the value of the key, or indirectly understand the function called by the algorithm and from this infer the right value of the bits of the key. This channel mainly takes advantage of the fact that the cache can be shared by multiple programs running on different threads or even different cores, relaxing the requirements that an intruder must satisfy to be able to exfiltrate the desired secret data.

### 1.2.1. Time driven attacks

The attacks belonging to this subclass of MSCAs are based on the assumption that the time required to complete a cryptographic operation is tide to the value of the key used, as reported in [6, 7].

AES is a symmetric cryptographical algorithm that applies key-dependent operations such as permutations and shifting to the input bytes in order to obtain the encrypted/decrypted data. These operations are repeated multiple times in what are called rounds. To make AES fast enough to be usable in a real environment, as a server, which must perform encryptions and decryptions at an elevated throughput, the actual implementations use the so called S-boxes. These S-boxes are arrays which contain the results of the round operations, for every possible combination of key and plaintext, and the time needed to access such arrays is index-dependent, so it can give some hint about the value of the key.

Bernstein, in his work [6], conducts a research on the OpenSSL implementation of AES. In this case, the S-boxes are structured so that the time to complete a cryptographic operation depends on both the given key and the given plaintext. Called  $k$  the 16-byte array of the key, and  $n$  the 16-byte array of the plaintext, the S-box access time is influenced by the relation:  $k[0] \oplus n[0]$ , for every element of the array. The attack is done by focusing on each byte of the key at a time, and then repeating the same procedure with the other bytes. In this explanatory example we will focus only on the byte  $k[0]$ , but the procedure is identical for all the other bytes. As a first thing you have to measure, on the attacked machine, the time needed to encrypt a plaintext for every possible value of the byte  $n[0]$ , leaving all the other bytes and the key unchanged, and find the value  $y$  that

requires the highest time to be computed 1.2. At this point, in a system controlled by the attacker, identical to the targeted one, you have to conduct a search, using random values for the key and the plaintext, whose purpose is to determine the value of  $x$ , with  $x = k[0] \oplus n[0]$ , which maximizes the execution time. Finally the value of the key can be easily computed inverting the previous relation between the key and the plaintext into  $k[0] = y \oplus x$ . After having repeated this sampling phase for all the key bytes we have a bunch of possible values for each key byte. The proposed attack can be carried out remotely, sending to the victim the same plaintext used during the sampling phase (in this case a text of only 0s). The author suggests to use different lengths of the packet's size to obtain other possible set of key's values, and merging together all these information is possible to narrow down the probable key values until is feasible to conduct a complete search attack.

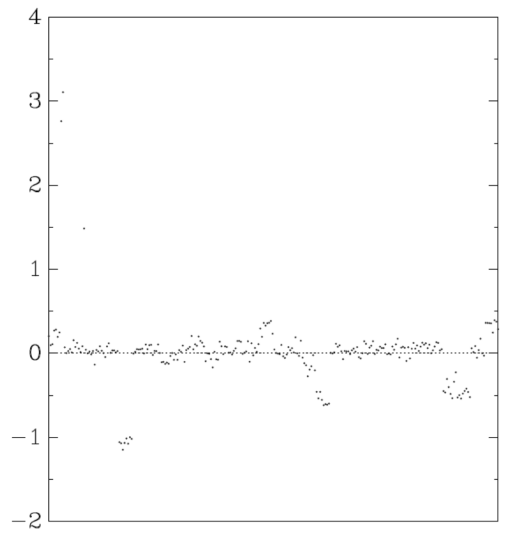


Figure 1.2: Time distribution for all the possible values of  $n[13]$  normalized over the mean time, taken from [6]

### 1.2.2. Access driven attacks

These type of cache-based side-channel attacks exploit the fact that the cryptographic algorithms, to perform encryptions and decryptions, need to have access to some elements or structures stored in memory and loaded into the cache. These elements are tied to the value of the key used by the security primitives, think of the S-Boxes for AES [39], and they are used to reduce the computation time, but are generated from the key. So, determining which cache line was accessed, is possible to retrieve the value of the key. The main requirements that unite the different strategies, that resides under this category, are:



- Both the attacker and the victim programs must share the same cache, at any level, or share the same memory page.
- The attacker must have the ability to obtain accurate measurements of the elapsed time from an arbitrary instant of time.

The way these side channels work is quite similar one from another, they have two phases. In the first, the monitored cache lines were put into a known state and then the victim program was able to execute some instructions. At this point, the second phase of the attack understands which are the lines accessed by the victim executable. This phase is done by comparing the access time to the lines, which will be different between a modified line and a line that was not touched by the victim program.

An interesting side channel is *Flush+Flush* [21] because it has a more stealth approach with respect to the others. The researchers that built it point out that the other cache-based MSCAs make some access to the cache to be able to extract the secret data, and this has an impact on the system, which can be detected by the usage of the Hardware Performance Counters. To circumvent this defensive strategy a zero-writes policy is adopted. The attack firstly flushes the cache lines we want to monitor, this is done using the **cflush** instruction; with it the CPU will not trigger the prefetcher and so it is possible to monitor multiple addresses within a page. After having measured the execution time of the cflush instruction, it is possible to determine if that line was cached or not; in fact, in the former case the time needed to evict it from all the cache levels is higher. At this point the cache is already in good state to let the victim execute again its code and then do another iteration of the attack. This attack does not require that the victim program resides on the same core of the attacker; in fact, it can be performed in a cross-core fashion. In addition, if a read-only memory page is shared between the victim and the attacker program, it is also possible to attack virtualized environments. Even if *Flush+Flush* is a more error prone approach, due to the reduced difference in time between a cache hit and a cache miss, in the end it has a throughput higher than *Flush+Reload*, which is another famous cache-based side-channel attack that targets the Last Level Cache.

### 1.2.3. Spectre

Spectre is an attack that, with the use of a cache-based side-channel, is able to gain access to the victim program memory, and extract data such as a cryptographic key. The attack is based on the improper use of speculative execution, forcing the processor to perform illegal instructions, that, even if they are reverted, they left a trace in the cache. In the paper the authors showed two variants of this attack [27].

The first variant focuses on forcing the branch predictor to make an out-of-bound access to an array. This version requires that the attacked program has a piece of code of the type listed in 1.1, where  $x$  is a variable controlled by an external untrusted source.

---

**Algorithm 1.1** Spectre variant 1, code requirements

---

```

1: if  $x < array1\_size$  then
2:    $y = array2[array1[x] * 4096]$ 
3: end if

```

---

The bound check on  $x$  is put to avoid an out-of-bound access, because, otherwise, someone could read data in the private memory of the victim process. To read an arbitrary byte  $k$  is enough to supply  $x = address\_of\_k - base\_address\_array1$  and force the CPU to execute the array access, even if the bound check fails. To make it possible to bypass the if condition, this attack exploits the functioning of the speculative execution, which makes a prediction on the next instruction and executes it before the bound condition is actually solved. The CPU initially is mistrained, sending it a great number of valid  $x$ 's, so that the branch predictor will take the *then* branch when we send it the  $x$  with a value out of the boundary. Another precondition to have a successful attack is that the secret byte  $k$  is cached, and  $array1\_size$  and  $array2$  are not. Once all the assumptions are met, the attacker can send the maliciously crafted  $x$ ; due to a cache miss the if condition cannot be resolved immediately. At this point the speculative execution activates and read the value of  $array1[x]$  accessing it from the cache. Then, also the access to  $array2[k * 4096]$  is done, and it will result in a cache miss. Hopefully it is only now that the if condition is solved, and all the executed instructions are reverted, but the access to  $array2$  may have already loaded the value into the cache. Now the attacker can use a side-channel such as *Flush+Reload* to understand which cache line was accessed, and since the accessed line is linked to the value of  $k$ , it is possible to retrieve such secret data.

In the paper is presented a second variant of Spectre. This one is focused on forcing an incorrect prediction by the indirect branches predictor that will lead to an arbitrary memory access. In this scenario we have to find a gadget, which is a small block of instructions that can be misused by the attacker, that gives us the possibility to read arbitrary memory addresses, and that leaves a trace of such reading into the cache. As in the other variant, we need to mistrain the branch predictor, but, in this case, this step must be done in a separate, attacker controlled, context. The attacker has to develop a code which has a jump at the same address of the targeted branch instruction. Then he must set this jump to go to the address where is located the gadget we want to use, and, correctly managing such jump, the predictor will be forced to make this prediction.

The next phase is to execute the victim program and let the speculative execution take the wrong prediction, which executes the gadget code that leaks into the cache some secret data. As in the other variant, the data are retrieved from the cache with a cache-based side-channel attack such as *Prime+Probe*. It is important to note that the branch predictor has a core scope, so the mistraining must be done on the same core where the targeted process will be run. Even if the predictor learns from jumps to illegal destinations, a critical aspect, during the mistraining phase, is to correctly manage the exception launched when the predictor jumps to the gadget's address to let the attacker program continue to run.

#### 1.2.4. Meltdown

Meltdown is an attack that misuses the out-of-order execution to grant access to the kernel memory, which gives the attacker the ability to read the entire mapped physical memory. The authors defined the concept of transient instructions, these are instructions executed out-of-order, before the previous instructions are completed, and that leave a measurable effect on the CPU microarchitecture. The idea behind this attack is to make that a sequence of transient instructions read a secret value in the private, user-inaccessible, memory of the attacked program, and then recover it using a side-channel. When an instruction tries to access a restricted memory page, the processor raises an exception that must be managed somehow to complete the attack; the researchers proposed two possible solutions to this.

As a first solution, you can decide to handle such an exception. Before executing the transient instructions, you make a fork and let the child process execute the unauthorized instructions; in this way only the child will be stopped and the parent process can retrieve the secret data. Otherwise, you can insert a signal handler to handle the segmentation fault.

The second method to cope with the exception is to suppress it. The suggestion is to use a transactional memory instruction to access the desired value. Due to this type of memory access, you can group together multiple read operations and perform them as an atomic action. But the great advantage of this type of instruction is that if an exception is raised, during the memory read, all the operations are reverted, the CPU will be put in the previous correct state, and the execution continues without terminating the program. Another method of causing a suppression of an exception is to make sure that the "than" ramification of the branch, which contains the transient instructions, will never be taken, and it is executed only speculatively. In this way, any exception raised

during the speculative execution is discarded after having determined that the prediction was wrong.

---

**Algorithm 1.2** Example of Meltdown code (attack phase only, no value recovery), taken from [31]

---

```

rcx = kernel address, rbx = probe array
1: xor rax, rax
2: retry:
3: mov al, byte [rcx]                #read a byte from the secret's address
4: shl rax, 0xc                        #multiply the secret byte by a page size (4KB)
5: jz retry                             #retry if read 0
6: mov rbx, qword [rbx + rax]        #access a secret dependant cache line

```

---

In 1.2 an example of code is reported, made available by the authors of the paper on Meltdown, that represents the first phase of the attack, when the secret value is read and "stored" in the microarchitecture of the CPU. In the preliminary part of the attack we put into the register *rcx* the address where is located the secret value we want, we create a probe array with a dimension of  $256 * page\_size$ , and put its base address into the register *rbx* and then we evict the probe array from the cache. At this point, the attack can start and is made up of three steps.

The initial step load into a register the chosen secret value, whose address is saved in the register *rcx*. This step is done by line 3, in the example code 1.2, where a byte is loaded into the least significant bits of the register *rax*. After having loaded the secret value, there is a time window before the CPU signs the instruction at line 2 as retired and consequently interrupts the execution due to the raised exception. If we can win the race condition and execute the second step before the exception, we can recover the secret data.

The second step is responsible for generating a change in the microarchitectural state, which will not be modified when the processor will revert instructions due to unauthorized access to a memory location. To generate such a change, Meltdown first multiplies the read byte by the size of a page (typically 4KB), as done in line 4, then uses this value as an index to access the probe array, line 6. After the last instruction, we have made a change in the cache status that is tied to the value of the secret data we want to extract. Sometimes can happen that the value in *rax* is 0, and not the desired secret, in such a case, the first step is redone. The last step will recover the secret from the architectural state; in our example is enough to cycle through the elements of the probe array, with a

stepping of page size, and measure the time needed to access it, the index's value that requires less corresponds to the value of the secret data byte.

### 1.3. Machine Learning

Machine learning is a field of Computer Science that develops algorithms that, after a training process, are able to make classification or predictions on new and never seen before data without that anyone programmed them for this. Each sample given to these algorithms is represented by an array, where each element is a value representing a certain characteristic of the sample. The values of the features are used to be able to make distinctions, find correlations between samples, and do predictions on the value of future samples. We can identify three macrogroups of ML algorithms based on how their learning processes work.

- Supervised learning
- Unsupervised learning
- Reinforcement learning

Supervised learning requires that the user gives to the algorithm a training set composed of labeled elements so that it can find a correlation between the element's characteristics and their value with the suggested output. The algorithm has an objective function that will be used to determine the labels of the new data. Such a function is optimized in an iterative way by cycling through all the training samples and modifying the function's parameters so that it will output the expected label for the given sample.

The unsupervised learning instead takes as input, for the training, only the samples, represented as a features vector, without any labeling. The algorithm in this case tries to identify commonalities between different samples in order to group them together under the same label. After having clustered the training samples, the algorithm can start to classify unknown samples, basing its decision on which is the cluster that is more similar to the given sample. In these algorithms exist some parameters that the user can change in order to obtain better performance, an example of such a parameter is the number of clusters to be created.

Reinforcement learning is typically used when there is the need to teach to a system how to interact with a certain environment, for example how to drive an autonomous driving car in a city. In this scenario there doesn't really exist a training set, but the algorithm has to interact with the environment, and to do so it has the possibility to make some

moves. After each movement that the algorithm does, it receives a feedback and uses this information to continuously improve its ability to move inside the environment.

A One-class ML algorithm is an algorithm that makes classification based only on a binary choice, typically to individuate elements that are in or out of a group of interest. An example can be to determine if an economical transaction is fraudulent or not. The elements that the algorithm classifies as non-belonging to the group of elements considered good are called outliers. The advantage of this class of algorithms is that to be trained they only need samples coming from the group you want to identify, the so-called inliers.

### 1.3.1. Local Outlier Factor

Local Outlier Factor (LOF), as explained in [8], is a machine learning technique that, instead of sharply classifying a sample as an inlier or an outlier, gives each sample a value that represents how much it is an outlier compared to its  $k$ -nearest neighbors. To be able to understand the meaning of such local outlier factor is necessary to define some useful functions. First of all, we have to define the concept of  $K$ -distance( $p$ ), where  $k$  is an integer and  $p$  is an object in our system. The  $k$ -distance( $p$ ) is defined as the distance  $d(o, p)$  between  $p$  and another object  $o$  such that there are at most  $k-1$  objects  $o'$  with a distance from  $p$  strictly lower than  $d(o, p)$  and at least  $k$  objects  $o'$  with  $d(o', p) \leq d(o, p)$ . The authors of the paper called the group of objects whose distance is less or equal to the  $k$ -distance( $p$ ) the neighbourhood of  $p$ , and to refer to it they use the notation  $N_k(p)$ . Now the notion of *reachability distance*, denoted as  $reach-dist_k(p, o)$ , of the object  $p$  respect with object  $o$  can be defined. This distance is  $k$ -distance( $p$ ) if the object  $o$  is in the neighborhood of  $p$  otherwise is the distance  $d(o, p)$ . The authors made this choice to reduce the variations of the distances of the objects which are near the point we are considering. The value of  $k$  is a crucial parameter, and the only one presented in the paper. LOF is not only based on the concept of  $k$ -nearest neighbors distance, but also takes into consideration the local density of objects. The *local reachability density* of  $p$  is defined as:

$$lrd_k(p) = 1 / \left( \frac{\sum_{o \in N_k(p)} reach-dist_k(p, o)}{|N_k(p)|} \right)$$

It is the inverse of the average reachability distance between  $p$  and all the objects in its neighborhood. At this point, we have all the necessary definitions to introduce how to compute the Local Outlier Factor; its equation is the following:

$$LOF_k(p) = \frac{\sum_{o \in N_k(p)} \frac{lrd_k(o)}{lrd_k(p)}}{|N_k(p)|}$$

is the ratio between the sum of the local densities of the objects in the neighborhood of  $p$  and the local density of  $p$  itself, all normalized by the number of objects in the neighborhood of  $p$ . It's easy to see that if the analyzed point is in a cluster of inliers samples then its local density will be very similar to that one of its neighbors and so we will have a LOF near 1. Instead, a point that is out of a cluster, and so it's an outlier, will have a lower local density, and this implies having a LOF greater than 1. To obtain a binary classification algorithm is now sufficient to fix a reasonable threshold on the value of LOF to divide the inlier from the outlier.

### 1.3.2. Support Vector Machine

A support vector machine (SVM) aims at finding a hyperplane that includes all the given training samples, this in the One-class version [43]. Then it classifies the unknown samples based on where they fall inside the feature space, basically, if they are outside the hyperplane, they are classified as outliers. The definition of a hyperplane is seen as an optimization problem. The algorithm has to find the curve that best includes the given training samples, but without being too tied to them, otherwise, we will have a predictor biased towards the training samples. Due to the fact that SVM tries to identify a linear separation between the samples, the choice of the kernel function to use can make a non-negligible difference in the obtained results. In fact, the kernel function is responsible for mapping the input data to a higher-dimensional space where they can be linearly divided. In fact, the starting problem space is defined by the features of the samples, but such features may collocate the samples in a way that makes it impossible to find a linear equation that divides them, and it is then necessary to increase the dimensionality of the problem space. The hyperplane is found by solving the following optimization problem:

$$\min_{w \in F} \frac{1}{2} \|w\|^2$$

whose constraint is  $(w \cdot x_i) \geq \rho, i \in [1, \dots, l]$ . In the previous formula  $w$  is the hyperplane that, among the ones that are able to separate the samples from the origin, has the maximum distance from the origin. Instead  $x_i$  is the  $i$ -th sample from the training set after having mapped it, through the kernel function, to the new, higher-dimensional space. Instead,  $\rho$  is a nonnegative real parameter.

### 1.3.3. Isolation Forest

This ML algorithm belongs to the family of ensemble methods which take a decision on the output to assign to a certain sample using a system based on majority voting. The voting system is represented by multiple instances of trees, each one with its own inner decisional structure, that classify a given sample, at the end the class with the highest number of votes is assigned to the sample. Other detection algorithms try to understand if a given sample has characteristics in common with those ones that they know to be inliers, and if the given object is too different from the reference samples it is classified as an outlier. The researchers that have created Isolation Forest [33] instead think it is easier to test if the given sample diverge from the learned normality. The authors made the observation that outliers have very different values of the features with respect to the inliers, so a tree can be structured to easily separate them from these last ones. Furthermore, because of their natural isolation from the other samples, they can be separated from the rest in a lower number of steps, creating a leave in the tree very close to its root. The average length of a path done by a sample to be isolated is used to determine if it is an outlier or not. This algorithm takes as input only two parameters: the number of trees that make up the forest and the number of samples used to build each tree in the ensemble. Each tree is generated starting from a subset of the training samples, and at each step a feature is selected at random and a partition value is randomly chosen, generating two daughter nodes. The partitioning is stopped once it reaches a predetermined height limits, or if the algorithm has separated all the samples in the subset, or if all the remaining samples have the same values. The path length is computed simply as the number of edges traversed from the root node to the terminal leaf, and is referred to as  $h(x)$ , then the final value used for the classification is the average length among all the trees of the forest. To have a working classifier, it only remains to define a score function and a relative threshold. Called  $c(n)$  the average length of a tree built from  $n$  elements, the authors define the anomaly score as:

$$s(x, n) = 2^{-\frac{E(h(x))}{c(n)}}$$

where  $E(h(x))$  is the mean value of  $h(x)$  computed on all the trees of the ensemble. Established this, if the mean value of  $h(x)$  is similar to  $c(n)$  we obtain an anomaly score of 0.5 this is thought to be a good threshold. Every sample that obtains a score lower than 0.5 will be classified as outlier, and more the score is near 1 more we are sure it is a normal sample.



## 1.4. Gem5

The tool `gem5` is an architectural simulator [15]. With this tool is possible to simulate different CPU's architectures as: x86, ARM and RISC-V. You have the possibility to define your own CPU model, personalizing every details from the frequency of processor to the cache composition, and even how the cache works. The tool is written in C++ language, but you interface with it writing Python scripts, which contain the detailed structure of the CPU you want to simulate. Such scripts can be parameterized in order to have a single script that can simulate multiple CPUs. An example of this is the provided script for x86 architecture which gives you the possibility to set a huge number of parameters, just as example: the number of memory channels, there is a list of possible memory belonging to different standards from DDR3 to the newer LPDDR5, or the type of branch predictor. The default script offers also the possibility to choose between different CPUs, which simulate processors of various complexity for an in-order CPU to a complete out-of-order processor. Such scripts define also how all the components you put in them interact each others.

The simplest CPU is *AtomicSimpleCPU* a basic in-order processor that performs atomic memory accesses whose latency is an estimate based only on the response time of the cache. Atomic memory in fact does not take into account resources contention or queue delay for estimating the access time to a cache line. *AtomicSimpleCPU* is useful for testing purposes, or to warm up the system because, given its simplicity, is a fast CPU to be simulated.

For our experiments we opted for the most accurate processor made available, the *DerivO3CPU*. This processor has 5 stages, and use timing memory, which is the most time-accurate memory access system offered by `gem5`. The most important aspect of it is that it is an out-of-order CPU, this gives us the possibility to reproduce the attacks that exploits such functionality, and it supports also the speculative execution.

The `gem5` simulator has two working modes: `syscall` emulation and `full system`. With the former you can run only user space programs and, as the name suggests, every `syscall` made by the process is emulated by `gem5`, but it does not support all the possible `syscalls`, and so to use this mode can be necessary to modify the source code of `gem5`. This mode is very easy to use, in addition to the CPU configuration, you only need to give the executable program you want to run. Unfortunately this simulation method isn't enough for our project, because it is too limited, by both the number of supported `syscalls` and by the fact that the program can run only in user space, making us impossible to simulate the side-channel attacks.

We opted for Full System emulation which simulates the entire Operative System (OS) that runs on a processor. This choice gives us also the possibility to put in place a more realistic testing environment. The Full System mode is a bit more complicated to run; in fact, it needs a kernel and a boot drive with an OS, in addition to the configuration file. Once you have started the system you can interact with it through a remote terminal interface. You must put all files you need to use inside the boot drive, to do this, the developers of gem5 offer an easy-to-use tool. In this emulation mode you have also access to a program called *m5* which provides some useful commands, a sub-group of these are used to control the sampling of the Hardware Performance Counters made available by gem5.

## 2 | Related Works

In this chapter, we explore works done by other researchers on the same topic of this thesis, to have a better understanding of which is the state of the art in this moment. For what concern the security detectors, which aim at disclose an attack as soon as possible, we can make a macro distinction between those that use Machine Learning and the ones that don't.

### 2.1. Machine Learning-based detectors

Bazm et al. in [4] have proposed a software system oriented to the detection of attacks that target Virtual Machines (VM) co-located on the same physical machine of the attacker. In this work the authors used some HPCs, selected on the basis of other works. They ended up using: *LLC-misses*, *LLC-references*, *iTLB-cache-misses* and *iTLB-r-accesses*. Their choice is guided by the fact that they want to monitor the Last Level Cache, which is a shared hardware resource among the VMs, and so it represents a vulnerable attack surface. Differently from the others, they used also information from the Cache Monitoring Technology (CMT), an Intel technology that monitors the cache occupancy of the processes, VMs in our case, running on the system and with the possibility to apply restrictions to those processes that are more resource-demanding. Then, the proposed security program combines all these input data and uses a Gaussian anomaly detector to find out the outliers. The proposed system requires three threads to work; one dedicated to collecting samples, one to determine the number of active VMs and the last one that runs the ML algorithm.

A group of researchers tested different Machine Learning algorithms to determine which are the most suitable to detect MSCAs [47]. As others have already done, they started by selecting the 4 best HPCs to use, from a starting set of 16 they have reduced it thanks to the Pearson's correlation coefficient. Wang and the others also tested different sampling intervals to determine which one is the most reasonable choice, taking into consideration also performance overhead and time needed to acquire the sample. The sampling frequency is a crucial parameter of a detection system, and it is vital determine

which is its best value, in fact, we have used these results in our work. In order to put in place the widest possible research they tested six different ML algorithms from different categories, such as neural networks, tree-based algorithms and support vector machines. What they have in common is that, for the training phase, they need samples from all the categories you want to be able to classify. For the conclusions they considered also the time that the ML algorithm takes to make the predictions, and not only the accuracy it provides, in this way is possible to make a better comparison between the tested algorithms.

In the work by Depoix and Altmeyer in [11] the choice fell on the optimization of Neural Networks to detect Spectre attacks. Also they relied on Hardware Performance Counters as features for ML. Their choice was based on the behavior of Spectre that led them to use the number of L3 cache misses, in combination with the L3 cache accesses, to prevent that a benign process, with elevated cache miss, will be misclassified and ultimately the total number of executed instructions. To collect the samples and later make a realistic analysis of them, the authors set up eleven scenarios which represent various situations. Such scenarios go from an attacked environment, using various proof of concept codes, to a web server, and also a couple of scenarios that run a resource intensive program to check if it will mislead the ML algorithm. The proposed security system is software-based and has a composition similar to [4], there is a program that monitors the started and stopped processes, then another service keeps track of the HPCs on a per process basis, then send these data, sampled with a frequency of 100ms, to a third program responsible for the predictions.

Chiappetta et al. have conducted a study [10] on three different techniques to detect a cache-based MSCA attack, two of them are based on machine learning. All the proposed methods use Hardware Performance Counters to reach their objective. For the two ML-based methods, the authors found the best results with the following HPCs: total instructions, total CPU cycles, L2 cache hits, L3 cache misses, L3 cache accesses. The first ML algorithm tested is an anomaly detection one, even if they didn't name it, but its way of working is described. The training consists in finding the mean and the squared variance of each feature. Then it computes the probability density function for each of the training samples and finds a threshold such that the probability density function of all the testing samples is lower than this threshold, if the sample comes from a spy process. The second technique, based on machine learning, proposed by Chiappetta is to use a Neural Network to classify the acquired samples. This approach requires to have samples from all the classes you want to recognize, in this case they are only two: *malicious process* or *benign process*.

Alam et al. have proposed in [1] a Machine Learning-based detection system that is structured on three levels. The first one, as commonly done by other solutions, samples at a given frequency some Hardware Performance Counters that were selected to be significant for attack detection. Such samples are then given to a One-class SVM for anomaly detection, and this ends the first step of the proposed security system. The second phase is meant to reduce the number of false positives, in fact the samples previously labeled as anomaly is now passed through another ML algorithm. Such algorithm is trained to classify different type of cache-based side-channel attacks, and this finer classification is needed to perform the following step. The final phase of the security architecture tries to correlate the potential attack with the key of the cryptographical algorithm. To be able to make such a step, the result of the previous phase is used to determine which counters must be monitored to see if a correlation exists between the cryptographical key and the signaled process. The amount of existing correlation is obtained computing the alignment cost of a temporal sequence of samples generated by the encryption algorithm with a temporal sequence of samples generated by the suspicious process. If such a cost is low enough, then the analyzed process will be permanently classified as an attack; otherwise, it will be treated as a false positive, and this result is used to retrain the classification algorithm.

PerSpectron [37] is the only other MSCAs detector that is hardware-based. They proposed a hardware-based Machine Learning algorithm to detect as soon as possible an ongoing attack, this using information from pre-selected HPCs. For the HPCs selection they have firstly trained an ML model using all the features they had; at his point they found those ones that better separate benign samples from the attacked samples, this for each unit of the processor's pipeline. Finally, they ulteriorly reduced the features, but still maintaining some overlaps between them, to be able to detect different versions of an attack, ending up having a binary vector of 106 features. Each element of the vector is a feature, and if its scaled value is greater than 0.5 then we will have a 1 in the vector, otherwise we will have a 0, which simplifies the operations that the predictor has to do to determine whether the given sample is benign or not. Such a vector is then used to train the perceptron algorithm that will be actually used for detection, the result of the training is an array of weights. The authors suggest to use the detector only to signal potential attack to the OS, instead of directly taking some actions, this to prevent that a false positive cause a benign program to be killed or that it stops the entire system. The researchers also proposed to make their hardware system capable of receiving updates when new attacks will be found, in this way the weights used by the perceptron can be adjusted to detect also the more recent attacks.

The authors of [29] made a proposal to detect an attacking program, searching for an high and abnormal resources contention in the CPU, this utilizing a semi-supervised anomaly detection algorithm. Kulah and the others pointed out that an aspect common to all the cache-based side-channel attacks is that they will cause an abnormal level of contention on the CPU resources, and especially on the cache. The authors of the research have suggested to measure the levels of contention with the Hardware Performance Counters. The first phase to do this is individuate on which resources, the attacks you want to detect, cause a relevant level of contention. Once you have found such resources, the next step is to find the HPCs that give you the possibility to monitor the contention level on the previously determined hardware resources. For the actual detection, they have trained a k-means clustering algorithm only with samples from normal execution, so that every sample that will not reside in any of the clusters defined by ML is classified as an anomaly. During their research, they discovered that the workload level of the system can affect the precision of the results, so they have trained another ML model that is able to classify the system's workload level, and then, based on this information, they will select the MSCAs detector specifically trained to work on such workload. The authors made their tests with a software-based implementation, but they suggested to develop a dedicated hardware-based system.

## 2.2. Non ML-based detector

Yu et al. conducted an extensive analysis to determine how the cache structure influences the success of a time-driven attack [52]. They used gem5 to create various platforms with different cache configurations; then they measured how much such configurations are vulnerable to a time-driven cache-based attack. They focused their attention on the size and associativity of both private and shared cache, the size of a cache line, the replacement policy adopted, and the cache clusivity (namely, if a data is present in all the cache levels, or only in the one it is used). For what concern the attack, they made the tests with the one proposed by Bernstein on AES [6]. To be able to make comparisons between different configurations they defined the concept of Equivalent Key Length, a measure of how much the attack is able of reducing the key space.

In [28] three different approaches were proposed to defend a system from cache-based side-channel attacks. The first proposal is *preloading and PLcaches*, these aren't new protection mechanisms, but, when used alone, they have some vulnerabilities. The idea of the authors is to combine them to obtain better security, in fact, the preloading of all the secret data in the cache avoids different access timings due to cache misses. The

addition of Partition-locked to the cache, instead, blocks the attacker from changing such a situation where all the data are already loaded in the cache. To be effective, this solution requires that all the critical data can fit inside the cache. In addition, to minimize the number of locked cache lines, the authors proposed a modification to the Partition-Locked cache, adding the possibility to evict a locked line, but only if the owner process is not active. The second solution is *Informing Loads and RPlines*, where informing load is a particular instruction that raises an exception on a cache miss. Random-permutation caches instead are caches that apply a permutation to the mapping table between main memory addresses and cache addresses, so that attackers cannot predetermine where secret data will be loaded. The problem with such caches is that they are still vulnerable to collision attacks. The authors proposed to use informing loads to access secret data, so that, if a cache miss happens, a handler can load, in a random order, all the secret data, in this way the following accesses will result in a hit. The last proposal is *Informing loads and regular caches*, this is like the previous one, but it doesn't require dedicated hardware. This solution makes the permutation of memory address at the software level, and to reduce the overhead, still maintaining a good level of security, changes the permutation used only on a cache miss when it also loads all the secret data.

An example of a method thought to defend a system proactively is the one proposed by Fadiheh et al. in [12], which aims at detecting possible side-channels directly by analyzing the CPU design. They defined a set of properties, that can be checked by some commercially available software, and if those properties are satisfied then the chip can be considered devoid of side-channel vulnerabilities. This is a proactive method because the properties are checked during the design phase of the processor, and so, if revealed, the vulnerabilities can be removed before the CPU goes on the market. The authors say that a program has a Unique Program Execution (UPEC) if for different values of its secret data the microarchitectural state of the processor does not change in a clock-by-clock comparison, and also if the same architectural state is reached at the same time instant. The researchers transformed such definitions into a Computation Tree Logic (CTL) formula that can be checked by already available model checkers. They also defined two types of alarm that can be raised: *L-alert* and *P-alert*. *L-alert*, or leakage alert is raised when the simulation done by the model checker reaches a state where two architectural state variables differ with different secret data. An architectural state variable is a "state variable that defines the execution of a program at ISA (Instruction Set Architecture) level". *P-alert*, or propagation alert, is raised when the model checker's simulation reaches a state where there is a difference in the value of a microarchitectural state variable, which is not an architectural one. The microarchitectural state variables

are "the set of state variables belonging to the logic part of the CPU microarchitecture (e.g. registers and flip-flops)".

Another hardware-based approach to cache-based side-channel attacks, but without the use of neither ML nor Hardware Performance Counters, is proposed by Arikan et al. in [3]. They proposed to add a Security Checker (SC), connected between the instructions memory and the fetching unit of the CPU, that analyzing the fetched instructions and their frequency can statistically determine the presence of an attack. The checker uses a count-min sketch (CMS) to keep track of the frequencies of sets of instructions of particular interest. The checker works on a user-defined time window during which it analyzes the fetched instructions, storing through the CMS the frequency of the instructions sequences to be monitored. Such instruction sequences are stored inside a component of the proposed hardware module. At the end of the time window, the checker compares the information stored in the CMS with a database containing pairs of instruction sequences and frequency thresholds that are known to be the consequence of an attack. If the checker finds a match between an element in the attacks database and the observed fetching activity, an alert signal is sent to the OS. Both the size of the time window and the attack's database can be changed by the user through a dedicated interface.

Chiappetta et al. have conducted a study [10] on various methodologies to identify side-channel attacks. The first idea is to make use of the fact that both the victim program and the spy program access the cache in a regular and correlated way. The researchers proposed to constantly monitor every program running on the system in order to find overlapping traces of total LLC accesses, if such traces have a high correlation then there is a high probability that one program is attacking the other one. LLC accesses are monitored thanks to the commonly used Hardware Performance Counter.

### 2.3. Novelty of the proposed thesis

The previous works on this subject have done a lot of research, and they obtained excellent results for what concern the detection accuracy. By the way, such proposals suffer of various criticalities that can bias them towards some specific attacks or impede their usage in a scenario with limited computational resources available. A critical point common to almost all of them is that they are software-based solutions, this means that a portion of the computational power, or even an entire core, of the system must be reserved to the protection program. This may not be a problem for a High Performance Computing system, which has numerous cores available, and dedicating one of them to improve the system's security can provide more advantages than the cost required by the adopted



solution. But, if we consider embedded devices, they have limited resources, in some cases they have only one single core without any multithreading technology, and so it is unthinkable to adopt a security program that requires up to three dedicated threads to protect the system. Even if there exist some solutions that adopted algorithms for Machine Learning that can be trained using only samples from a non attacked execution, the majority of researches require to have access to a plethora of attack examples. This requirement is because, or the ML algorithms used need samples for each class they have to detect, or to retrieve from such examples a pattern that characterize each of the attacks you want to be able to detect. Such a solution inevitably causes the security system to be skewed towards the known attacks. In addition, just few of the researches explicitly told about the possibility of updating their model, so that it can accounts for newer attacks. Another critical aspect concerns the selection of HPCs to use for detection. The authors of the different papers did not go into depth on how they have chosen the HPCs used by their solutions, but seems they opted for a choice driven by some logical deductions based on how the attacks they analyzed works. Such a strategy led them to focus mainly on the cache, using events as the cache misses or the number of accesses to it, without exploring the existence of others, equally valid, events to monitor.

The security architecture proposed in this thesis is based on the results obtained from previous works on the detection of Microarchitectural Side-Channel attacks, and it makes some steps further to improve the obtained results and increase the security of the proposed architecture. Two are the most innovative choices we have made, the adoption of a hardware-based security checker, and the usage of a detection flow, from the HPCs selection to the ML model used, that is totally agnostic of the attacks that can target the protected system. Such choices give us different advantages. First of all using a dedicated hardware to make detection, we offload the system's processor from the defensive role that it has with other solutions; this is important because the devices targeted by our solution are embedded systems, which have a limited computational power. The second benefit, of a hardware-based detector, is that it is harder to be attacked because the exposed surface to a remote attack coming from the OS is reduced to the minimum; it is just the interface to update the detector's configuration. Instead, for what concerns the reasons of the adoption of a detection flow which is attacks agnostic, it is because it enables our system to not be biased towards a specific attack, and so it is able to detect effectively multiple attacks, even those ones that are not yet identified. We have also made a complete search in the HPCs space to identify which are the best ones to use, and, to the best of our knowledge, no one ever made such a wide search.



# 3 | The Proposed Security Architecture

In this chapter we describe the general structure of our proposed security module. We have developed a hardware module to detect an ongoing cache-based side-channel attack. This module have to be connected to the CPU, without requiring to modify it, of an embedded device, which main task is doing encryption and decryption operations. The purpose of such module isn't preventing an attack, but to detect it before it is able to extrapolate enough information to retrieve easily the key used by a cryptographical algorithm. When a security violation is identified our module communicates this information to the Operative System, which will be responsible for the selection of the counteractions to be done. We opted to focus on embedded devices because they are built with a precise scope in mind, and they run few or just one program at a time, and this simplify the detection of an attack. Their dedicated role makes them also more subject to an attack because it's more probable that high value information go through them.

Our detection system is based on a set of carefully selected Hardware Performance Counters whose values can be used to distinguish a normal execution of a cryptographical algorithm from an execution that is compromised by a side-channel attack. Due to the restricted number of HPCs monitorable at the same time in a real processor, we used gem5, a precise architectural simulator, to be able to collect all the performance counters offered by such a simulator. After having collected them we went through an extensive search to identify the best possible HPCs to use for the detection phase. In fact, a core aspect of the success of our solution is to carefully select which HPCs to monitor, modern processors give the possibility to track the value of very few events at the same time. As we based our research on the x86 architecture, we took as a reference the Intel manual [24] about this, and they offer only four configurable registers per thread, to monitor such events. Even if AMD offers 6 registers we preferred to choose at most four HPCs, mainly because we are working with embedded systems which are relatively low-performance devices and so it is reasonable to assume that they do not have many resources dedicated to performance monitoring. Another reason to work with the lowest amount of counters is

that, despite we are proposing an external module to process such information, we need to use some CPU components, as the registers, and especially bus bandwidth to transfer them. Because such resources are scarce, we want to minimize our impact on the system that otherwise cannot be able to perform its task acceptably.

To actually differentiate between samples representing a normal execution and those ones representing an attacked execution, we took advantage of some machine learning algorithms studied to make a binary classification. To achieve the best possible results we conducted tests on three different One-class ML algorithms, namely: Isolation Forest [33], One-class SVM [43] and Local Outlier Factor [8], with the final objective of identifying the best performing one in our application field. To test these algorithms we have used the Python library scikit-learn [44], a very famous library for making use of machine learning. It offers both ML algorithms and a variety of functions to manipulate and transform data, and also a set of functions which gives back some metrics which are useful to understand the effectiveness of tested algorithm. We opted for those algorithms that belongs to the One-class classifier, because the peculiarity of this group of ML algorithms is that for the training phase is required to provide only benign samples. This represents a great advantage for us, in such a way we don't need to retrieve an enormous database of samples representing cryptographical algorithms attacked by the vastest possible number of side-channel attacks. For what concerns the tested cryptographical algorithms, instead, we opted for: AES, Blowfish, Idea and RSA due to their extensive usage in modern programs and communication protocols for security purposes.

We can distinguish two separate phases in our security system, a training phase, and a detection phase. The former is done offline because it requires a non-negligible amount of time; in fact, in this phase, we collect all the samples needed for the training of the machine learning algorithms, conduct some tests to find out the best HPCs to make predictions on the existence of an attack, and then we train and tune the machine learning algorithms. The second phase, done by our proposed hardware module, is where we actually protect the system through an online monitoring of the CPU, sampling the values of the previously selected performance counters at a predetermined frequency. Then we analyze such samples using the tuned and trained ML algorithm and if it identifies an attack, our module sends an alert signal to the Operative System which will take some countermeasures.

As previously said, the training phase is done in a separate environment which does not have any time constraint for its execution. To complete this phase you must acquire a sufficiently large samples set, which represents a normal execution of the cryptographical algorithm you desire to protect using our system. To create such a dataset of samples

we used gem5, in its Full System emulation mode. For each cryptographical algorithm you want to protect you need two copy of its implementation, one that executes only the algorithm, and another, that in parallel to the security primitives executes also an attack. While the program was running, we acquired the samples, at the previously determined frequency, using a tool directly provided by the developers of gem5. Once you have obtained a dataset of a reasonable dimension, you setup a wide search to find the set, of four features, the module will use to detect an on going side-channel attack. As a first thing the search space can be reduced considering the variance of the features value, of only the samples originated by the non attacked version of the program, and putting some constraints on it. Such constraints can filter out those features that we are quite confident will be useless in identifying an attack. For example, features with an elevated variance will made difficult for the machine learning algorithm to identify the boundary of the region which contains only benign samples, or it will identify a too big region. After having reduced the search space, you start to train the ML algorithm, and find out which are the best features sets, making some testing with the samples originated from an attack. Taking in consideration only the best sets, you can figure out also which are the best parameters for the ML algorithm, and ulteriorly improves the effectiveness of the system. Once you have obtained acceptable results, you can save the trained model, with its parameters and the set of features, that will be loaded into the security module when needed. It is worth noting that this phase, from the sampling to the selection of parameters and the features, is done on a per cryptographical algorithm basis, and for each machine learning algorithm. At the end you will have a configuration (which is composed by the parameters of the ML model, the ML model structure, if needed, and the set of features to monitor) that must be loaded into the security module, through a dedicated interface, and changed every time you change the cryptographical algorithm used by the protected system.

The choice of the sampling frequency is based on another research, conducted by Wang et al. [47], which explored different sampling intervals. The choice of this parameter is important because we need to find a trade-off between the accuracy of the ML algorithm and the time required to acquire a sample. This research has found out that increasing the dimension of the time window the accuracy increases, but it rapidly reaches a plateau; at the same time, the performance overhead is reduced due to the lower amount of samples per unit of time to be acquired. Wang has also found that a good balance for the time interval between a sample and the following one is  $500\mu s$ . In the paper is not mentioned the frequency at which runs the processor used for the tests, but they said to have used an Intel I5-3470, and from the official product page [23] we found out it runs at 3.2GHz.

So, for our detection system, we decided to use this frequency, opportunely recomputed over the frequency of the CPU simulated by gem5, which has a frequency of 1GHz. Of course, the sampling frequency is another parameter of the module's configuration, but this one is not tied to the specific cryptographic algorithm used, or to the machine learning algorithm implemented by the security module; instead, it exclusively depends on the working frequency of the system, and must be set to have a corresponding sampling time of  $1.6ms$  at 1GHz.

The main innovative aspect of our thesis is that we want to use an external hardware module, a schema of which you can see in Figure 3.1, to make the detection, instead almost all the others proposed a software solution which must be run on the OS of the device to be protected. The latter solution has the disadvantage that it occupies system resources, and maybe this is not a problem in a high performance system, but in an embedded one it will. Other critical aspects of a software solution is that it's more exposed to cyber attacks that aims to disable it or masking some attacks changing the values of the hardware performance counters returned to the security application. This is a possibility because such values are not retrieved directly, but through another application which offer such a service, and that can be targeted by an attacker to manipulate the values it returns. Our solution, instead, being directly connected to the CPU, is harder to attack because it offers a very limited attack surface, in fact it is exposed to the Operative System only the interface used to load the module's configuration.

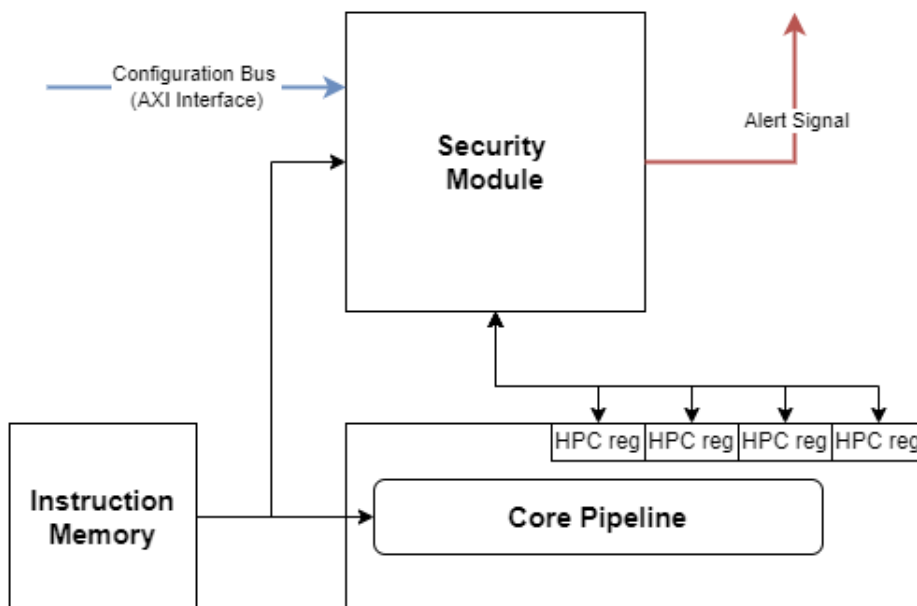


Figure 3.1: High Level schema of the proposed security module

Our module is structured to provide some configurability. It implements a certain Machine Learning algorithm, and this cannot be changed, but its parameters can, this is done to create the most configurable system we can, to address scenarios where different cryptographical algorithms require different parameters to maximize the accuracy of the ML model implemented by the security module. From our research it is evident that the set of features to be used changes from one cryptographical algorithm to another, and so this is the main parameter to be set to be able to achieve the best possible protection. Such a parameterization is done through a dedicated hardware AXI interface. In our thesis, we only made the hardware implementation of Isolation Forest because it gives the best results. The hardware module is composed of a series of sub-elements, there are multiple instances of the tree component, to represent all the trees of the ensemble, a timer to correctly manage the sampling frequency, a table with the features to be monitored, and a component that makes the final prediction based on the outputs of the trees. The module has an output bus, used to send an alert signal to the OS, when a sample is detected as an outlier, and an input bus on which it receives, through an external AXI interface, the configuration to be used. The only other external communication lines are those used to receive HPCs data and/or fetching activity. One of these lines communicates in a bidirectional way with the CPU and is used, initially, to set up the HPCs registers, and later to read their values. The second communication line is the one directly connected to the bus which transports the instructions from the instructions memory to the fetching unit of the processor, and this line is unidirectional from the bus to the security module. All the data coming from HPCs registers or the fetching activity are sent to each of the trees inside the module (these lines are not present in the schema to leave a clearer view of the main internal components of the security module). Once every tree has computed the path length of the given sample, it sends such value to the predictor which computes the average height of the sample and from this decides if it is or not an outlier. If the sample is predicted to be an outlier, and therefore an attack is detected, the predictor sends a signal to the OS over the dedicated *Alert signal* bus. If the sample is classified as a normal one, then the timer is reset and a new sample will be acquired.

The first thing to do to make the security module operative is to correctly set up it, a configuration file must be sent to it, through the dedicated communication bus. Such configuration file contains the trained ML model, the set of at most four Hardware Performance Counters to be monitored, and the eventual fetching activity to be considered. At this point, the module loads the trained model and sends to the processor the instructions to set up and start the monitoring of the features indicated inside the configuration file; this concludes the setup of the module. Now the online monitoring phase can start,

and our security system, every time window, controlled by the internal timer, retrieves the values of the monitored counters. These values form a sample on which the security module has to predict its outlierness. Such a sample is given to the Machine Learning predictor which will decide if it is representative of a normal execution or if, instead, there is evidence of an attack. In the latter case, the module will send a warning signal to the OS, through a dedicated signaling bus. How the Operative System has to manage the warning signal is out of the scope of our thesis, consequently, we will not do further reasoning about it. In the case the ML algorithm predicts that the given sample is benign, nothing happens, and the module gets ready to process the next sample from the following time window.



# 4 | Exploring ML for Attack Detection

Here we explain the methodology adopted to make our experiments, describing in a detailed manner all the steps we have done, and the obtained results, with some comments about them and reasoning on which are the machine learning algorithms that represent the best choice to be implemented in our security module.

Before everything else, we have selected a set of cryptographical algorithms to be tested, and we opted for the widely used: AES, Blowfish, Idea, and RSA. For each of these algorithms, we took a basic implementation of it, which is available on GitHub [17] that we have later modified to meet our needs in terms of usability for the testing we have conducted. Then we selected a couple of cache-based side-channel attacks to attack our system and to verify that the security module we are proposing is able to detect them. We took two Proof Of Concept (POC) from GitHub, one is a basic implementation of Spectre [19] and the other one is a demo of Meltdown [18]. We modified such POCs so that they attack the cryptographical algorithms selected for the testing and extract their secret key. The last preliminary choice regards the architecture on which to conduct the tests, we decided to use the X86 architecture due to its wide usage.

The first step to have some experimental results is to acquire some samples representing both a normal execution and an attacked one. We saved such samples in a separate dataset for each cryptographical algorithm and for every attack we have selected. To do this, we used the gem5 simulator in its Full System emulation mode; such a modality requires three elements to work properly: a processor's configuration file, a boot drive, and a Linux kernel. For the configuration file, we used the one provided by gem5's developers because it already offers a lot of parameters to set up the processor as we like. For the kernel, we took one, already compiled, which is provided by a tutorial page written by gem5's developers [16], we downloaded the one named *vmlinux-4.4.186*, that is a Linux kernel in its 5.2.3 version (there is an error in the downloaded file naming). At the bottom of this tutorial's page is also present a table that shows the compatibility

between some Linux kernels and different configurations of the simulated CPU. Finally, for what concerns the boot drive, we were able to find one based on the Linux distribution Gentoo [46].

To make a decision on how to configure the simulated CPU for our tests, we started from a work done by Jason [35], who is one of the main developers of gem5. In fact, he made a tutorial on how to reproduce Spectre on gem5 but using it in Syscall Emulation mode. Nevertheless, we can take advantage of some of his suggestions about the CPU configuration. Jason pointed out that to be able to reproduce Spectre you need to use the *DerivO3CPU* processor, because it is an out-of-order processor with support for speculative execution, which is a technology exploited by Spectre, and so without it will be impossible to reproduce it. He also changed the default branch predictor to *LTAGE*, which, due to its better performance, makes it easier to reproduce Spectre on gem5. On top of this, we have completed the processor's configuration by adding a two-level cache so configured:

- 16KB L1 Instruction cache
- 64KB L1 Data cache
- 256KB L2 cache

While we left the frequency of the CPU at the default value of 1GHz, and the core count to 1, without multi-threading.

For each cryptographical algorithm, we have three versions of it; a plain one that simply does For each cryptographical algorithm, we have three different implementations of it; a plain one that simply has a loop that first encrypts and then decrypts a randomly generated string, always with the same key, and for a number of times given by the user as an input parameter; a version attacked by Spectre, where a thread operates like in the plain version while another parallel thread conducts the attack, the program terminates as soon as the attack is concluded; and a version attacked by Meltdown which operates in the same way as the Spectre version. The attacked versions, as already said, are left to run until they conclude the attack; for the plain version, instead, we configured the programs to do 5,000 iterations of encryptions and decryptions for Idea, 8,000 for AES, 10,000 iterations for Blowfish, and 50 for RSA. The number of iterations was chosen to obtain nearly the same amount of samples for each algorithm, in order to have a balanced dataset among the various algorithms. The difference in the number of iterations required is due to the need for a different amount of time, to make the cryptographical computations, by the algorithms. Multiple runs, of the attacked implementations, may be needed to obtain

a number of samples sufficient to have significant results and to be compared with the other cryptographical algorithms. During the sampling, the only user-launched program is the one we want to track. To actually collect the HPCs data we used a tool, that is integrated with gem5, called *m5*, which offers some commands to interact with the simulation and to interact with the Hardware Performance Counters. The command line we have used to acquire the samples is the following:

```
m5 resetstats && m5 dumpresetstats 1600000 1600000 && [target_program]
&& m5 exit
```

where the *m5*'s command *resetstats* will reset the value of the HPCs, this is done to put the system always in the same starting condition, and to eliminate the traces of some commands which can have been launched before this one. While *dumpresetstats [delay] [frequency]* will wait for the amount of time specified by the *delay* optional parameter, expressed in simulated nanoseconds, then will dump and resets the values of all the HPCs that gem5 has, this is done every *frequency* simulated nanoseconds. *target\_program* is simply the path to the executable you want to profile. Finally, the command *exit* simply shut down the simulation. Each simulation saves the values of the HPCs in a file called *stats.txt*, appending each dump to the previous one. The sampling frequency was chosen based on a previous research [47] which find in  $500\mu s$  the more balanced sample interval, so we used it, transforming it to take into consideration the frequency of our CPU, so we obtained a sampling frequency of  $1.6ms$

The second step is to prepare the acquired data to be processed by the Machine Learning algorithms, in fact, they require as input a matrix having on the columns the features to use and on the rows the various samples. To reach this goal, we have written two help programs in Python, the *stats\_reader* and the *matrixer*. The former takes in input the *stats* file outputted by the simulation, which contains all the samples acquired, and split each sample into a different file, this is easy to do because each dump starts with "----- Begin Simulation Statistics-----" and ends with "----- End Simulation Statistics -----". The files generated by this program have a standardized naming convention so that it will be easier to distinguish plain samples from the attacked ones and merge samples originated by multiple runs of the programs. The samples originating from the plain versions of the cryptographic algorithms are named *stats\_part\_XXX.txt*, where *XXX* is a unique identifier number for the sample, and the enumeration starts from 0. The samples obtained from an attacked version, instead, are named *troj\_part\_XXX.txt*, where *XXX* is the unique identifier number of the sample, and starts from 0. The enumeration of the samples is independent between plain and attacked versions. Such a helping program additionally gives the possibility to

start the samples enumeration from a value different than 0, this in the case you have multiple runs to be merged together. The second helping program takes in input a directory containing all the sample files created with the *stats\_reader*, both the plain ones and those ones coming from an attacked simulation, and generates a .csv file that can be given to the ML algorithms. Such a program in fact creates a matrix putting in its rows the samples found in the given directory and on the columns the features that are common to all the samples. This program makes also some checks, in the specific excludes all the samples which have an execution time lower than the given sampling frequency, this can happen for example to the last sample due to the fact that the execution time of the tested program may not be a multiple of the sampling frequency. Another check is on the features, in fact, we excluded all the ones concerning the power consumption of the processor, because they may not be so precise given that we are using an emulator and not a real CPU. Then this helping program takes into consideration only the features inherent to the CPU and its cache, wiping out all the data about the RAM, for example. At this point, from its internal structure, builds up the final matrix.

The third and last phase, of our experimentation, involves the training and tuning of the One-class Machine Learning algorithms we have selected. This phase was done using Python and the library scikit-learn [44], which implements various ML algorithms, between them we have tested: One-class SVM, Isolation Forest, and Local Outlier Factor (LOF). The first step was to obtain some good sets of 4 features to use for training. For this, we have computed a submatrix that contains only samples from the plain execution of the cryptographical algorithm that we have called *plain\_matrix*. On the *plain\_matrix* we have computed the variance of each feature, then we considered only those features with a variance between 1 and the 80% of the maximum variance computed. This choice was done to exclude the features that have a nearly constant value between the samples, which can indicate features that are not affected by the execution of the program, and so, maybe, they remain constant also when an attack is involved. We also excluded the features that cause a too high dispersion between the samples, making it more difficult for the ML algorithms to define a model of normality. This filter prevents also obtaining a model of normality too wide, which then will be ineffective in the individuation of outliers because the majority of them will fall into the normality model. After this filtering, we have started to extract random sets of 4 features and test them against the two attacked datasets we have. Then, taking into consideration only the sets that give us the best results, we tried to improve their results by modifying the parameters of each ML algorithm we selected. After the tuning, to ulteriorly check the quality of such tuning, we have repeated the random extraction step. For each cryptographical algorithm and

for each machine learning algorithm, we have performed 200 extractions of possible sets of four features, from the group of filtered features, and tested them against the dataset containing plain samples and samples attacked by Spectred. From these 200 feature sets, we took only those that, after 10 runs of the ML algorithm, have a mean accuracy of 99% at least, except for One-class SVM, where we have set a threshold of 98% to have some useful results. At this point, we tested the sets that pass the previous selection with the plain sample data set plus the samples referring to the execution attacked by Meltdown, and we took only those that, on a mean of 10 runs, have at least an accuracy of 99%. At this point, we have a small selection of sets that are very promising. To complete our experiments, we tested the sets that have passed the selections, starting from the one that has shown the highest value of accuracy, and computed the average value of some metrics on 1,000 runs of the ML algorithm, for both the dataset with Spectre's attacked samples and the one with samples attacked by Meltdown.

After the tuning phase, we have found that the best parameters for the ML algorithms are the following:

- Isolation Forest
  - max samples: 256
  - contamination: 0.01
  - max features: 2
  - random state: 57
- One-class SVM
  - kernel: rbf
  - gamma: 0.00015
  - nu: 0.005
  - tol: 0.00001
  - shrinking: True
- Local Outlier Factor
  - n\_neighbours: 16
  - algorithm: kd\_tree
  - novelty: True

The main metrics we decided to use are accuracy, the F1 score, and the Area Under the Curve (AUC), because they represent a common choice among various other works that adopted some machine learning techniques. In addition to this, such metrics, considered together, give a more accurate image of how your ML algorithm performs. Before entering into the details on how such metrics are computed, we need to define some common terminology; with *True Positives* we refer to samples which come from an attacked execution and are correctly classified as outliers; with *True Negatives* we refer to samples coming from a plain execution, and correctly identified as inliers. With the term *False Positive* we refer to samples from a plain execution wrongly identified as outliers; and finally *False Negative* are those samples from an attacked execution which are classified as inliers by the ML algorithm. The accuracy is simply the percentage of samples correctly classified by your algorithm, and is computed as follow:

$$accuracy = \frac{TruePositive + TrueNegative}{TruePositive + TrueNegative + FalsePositive + FalseNegative}$$

While the F1 score is the harmonic mean of precision and recall

$$F1 = 2 * \frac{precision * recall}{precision + recall}$$

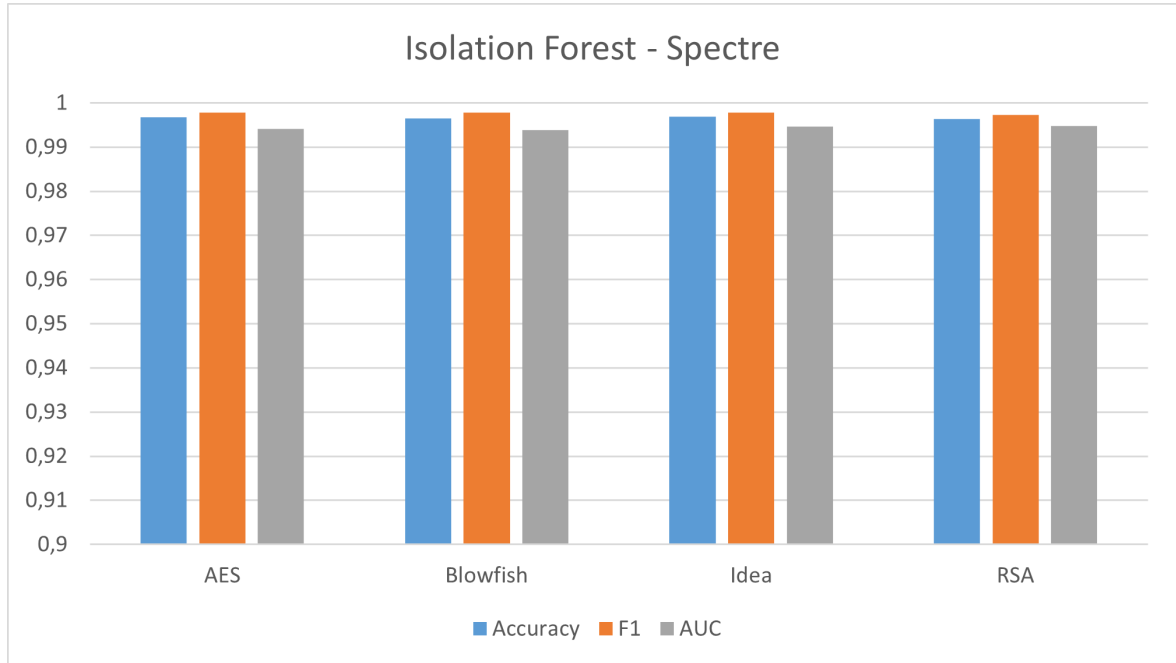
where the *precision* represents which is the percentage of samples classified as anomaly that really are anomaly, while the *recall* is the percentage of anomaly correctly classified as anomaly.

$$precision = \frac{TruePositive}{TruePositive + FalsePositive}$$

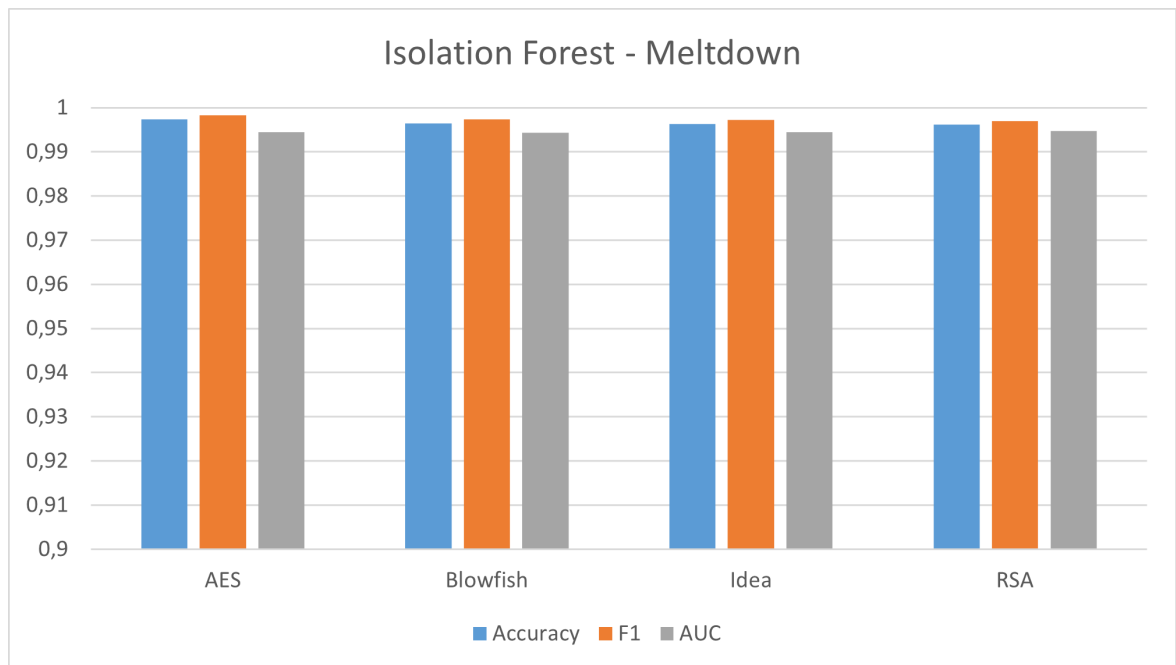
$$recall = \frac{TruePositive}{TruePositive + FalseNegative}$$

Finally, the AUC is the area under the receiver operating characteristic (ROC) curve compared to the maximal area that it can have. Where the ROC curve is a plot of recall against false-positive rate  $\frac{FalsePositive}{FalsePositive + TrueNegative}$ . All the results we have obtained are reported in the graphics in the Figures [4.1, 4.2, 4.3]. As you can see from these graphs we have obtained very good results. It is immediately evident that Isolation Forest outperformed in respect with the others algorithms, in fact it reaches values of over 99% in any tested scenario, with all the considered metrics. Immediately after it we have the One-class version of SVM, which is able to reach an accuracy greater than 99% in most of the situation, and in any case is always over the 98%. So, it is evident that these two algorithms are the ones to prefer for a hardware implementation. For what concerns the last tested ML algorithm, Local Outlier Factor, instead, it presents results which are good, but too variable to make it a usable solution. In fact, with LOF, we have

an accuracy that, in general, is between 97% and 99%, but presents a big drop with RSA, where it barely reaches the 94%.

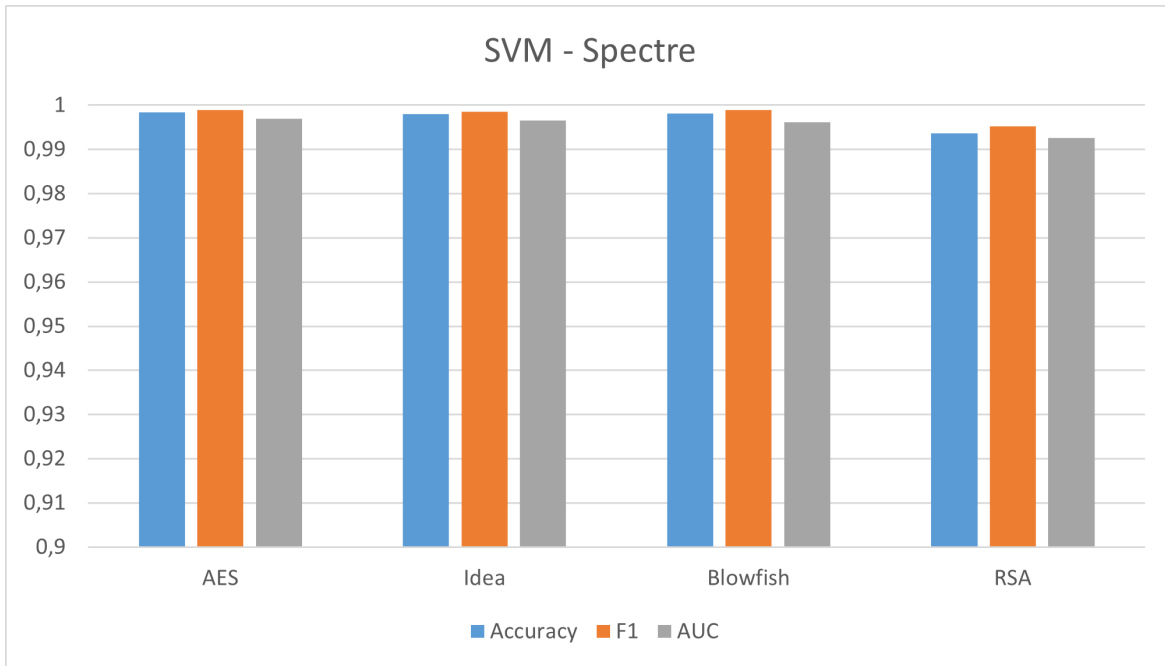


(a) Spectre

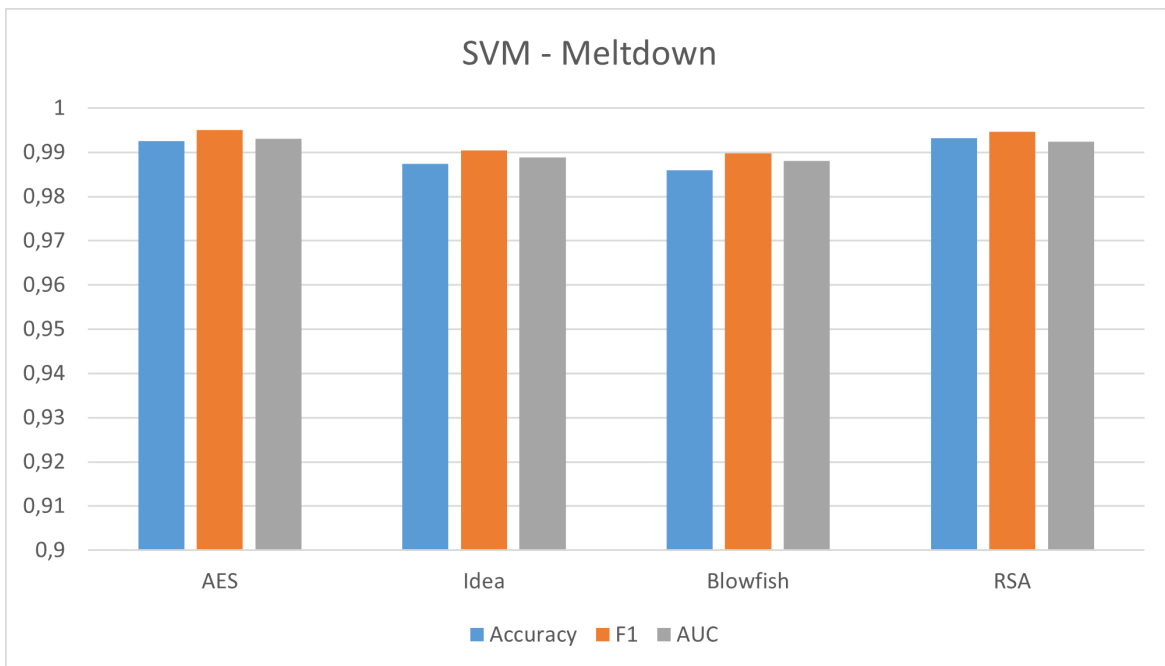


(b) Meltdown

Figure 4.1: Metrics for all the datasets tested with Isolation Forest



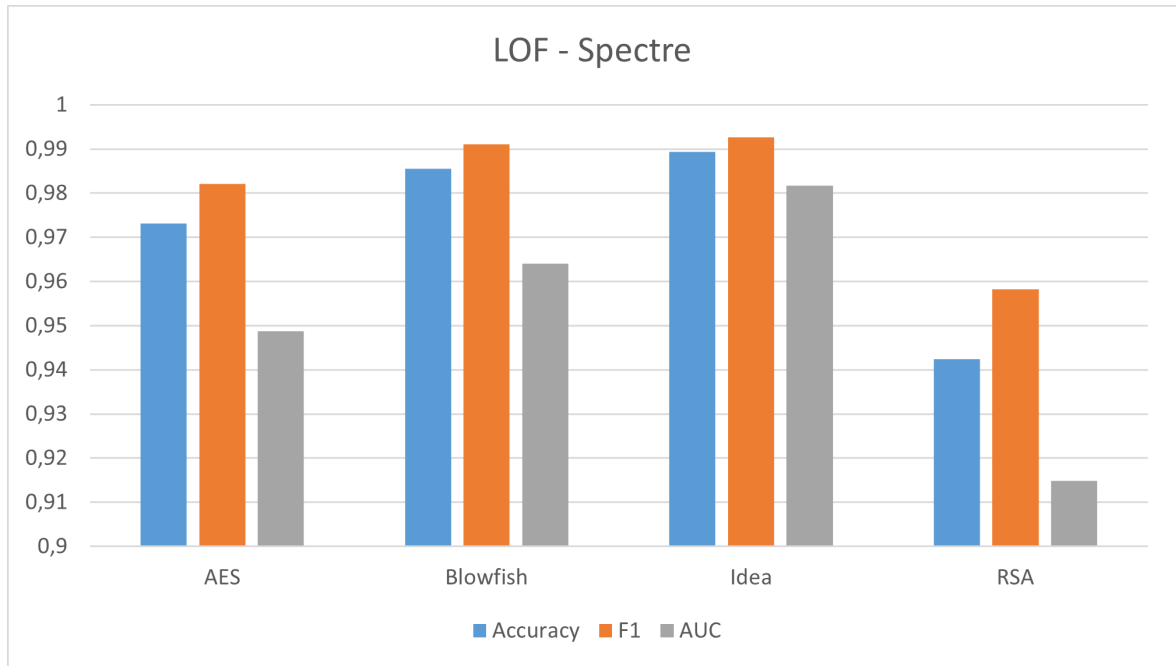
(a) Spectre-attacked



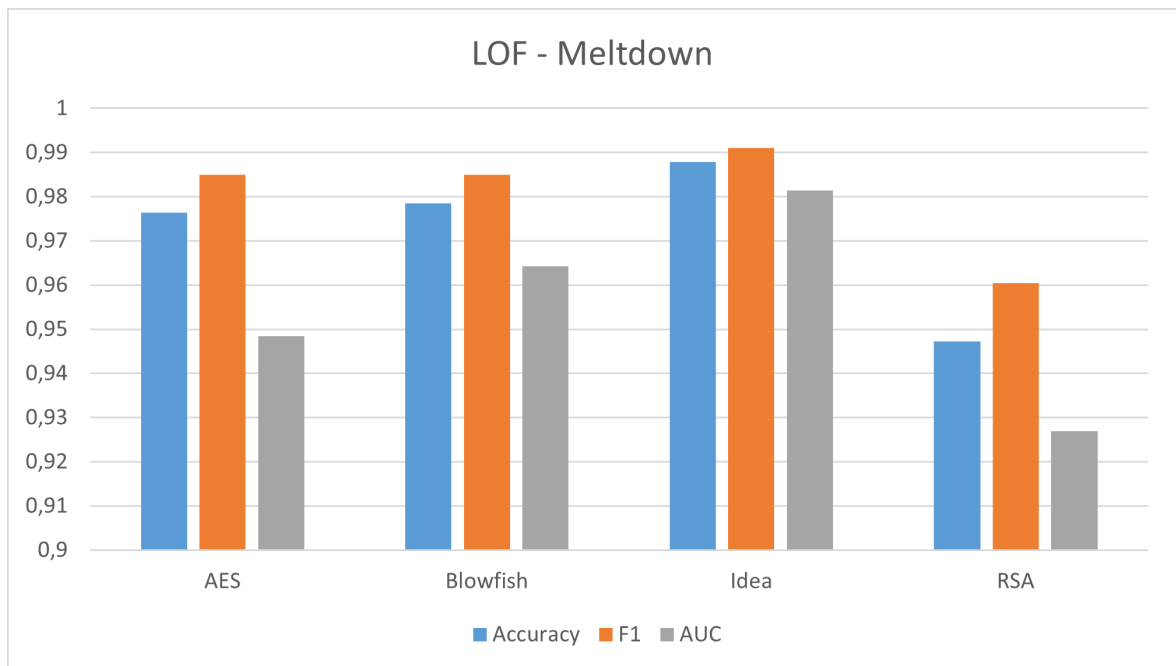
(b) Meltdwon-attacked

Figure 4.2: Metrics for all the datasets tested with One-class Support Vector Machine





(a) Spectre-attacked



(b) Meltdown-attacked

Figure 4.3: Metrics for all the datasets tested with Local Outlier Factor

In addition to the metrics mentioned above, we also generated a confusion matrix for each cryptographic algorithm with all the Machine Learning algorithms, and for both the attacks we considered, Spectre and Meltdown, the results are reported in the tables [4.1,

4.2, 4.3]. A confusion matrix is nothing more than a 2x2 matrix which plots the predicted labels against the true labels, grouped together by the classes detected by the ML model, and so in our case benign, or outlier. From such a matrix is then possible to retrieve the values of: True Positives, True Negatives, False Positives, and False Negatives. The results obtained are in line with the other metrics we talked about above, in fact, such matrices are showing that the best performing algorithm is Isolation Forest, while the worst one still be Local Outlier Factor. As can be seen in the table, Isolation Forest does not have any false negatives and it has less than 1% of false positives in any of the tested case. These results are extremely good, because not only means that we are able to precisely distinguish between a normal execution and an attacked one, but also means that we never classify an attacked sample as a benign one. The fact we do not have false negatives is crucial, because otherwise means that can happen that our detection module misses the presence of an attack, which then can reach its objective of stealing the secret key. Local Outlier Factor, instead, presents the worst results, reaching a 10% of false negatives with RSA, that is the dataset that, in general, is the most challenging to detect when it is attacked. Local Outlier Factor also presents a quite high percentage of false positives, between 1% and 7.5%. This results clearly indicates, as we already said, that such algorithm does not represent a good choice for our objective.

<b>Isolation Forest</b>									
<b>Dataset</b>	<b>TP</b>	<b>TP %</b>	<b>TN</b>	<b>TN %</b>	<b>FP</b>	<b>FP %</b>	<b>FN</b>	<b>FN %</b>	<b>Total</b>
<i><b>Spectre</b></i>									
<b>AES</b>	714	63,35%	405	35,94%	8	0,71%	0	0%	1127
<b>Blowfish</b>	990	70,97%	402	28,82%	3	0,21%	0	0%	1395
<b>Idea</b>	1004	70,8%	406	28,63%	8	0,57%	0	0%	1418
<b>RSA</b>	908	65,94%	464	33,7%	5	0,36%	0	0%	1377
<i><b>Meltdown</b></i>									
<b>AES</b>	858	67,5%	406	31,94%	7	0,56%	0	0%	1271
<b>Blowfish</b>	931	69,69%	401	30,01%	4	0,30%	0	0%	1336
<b>Idea</b>	844	67,09%	410	32,6%	4	0,31%	0	0%	1258
<b>RSA</b>	822	63,67%	468	36,25%	1	0,21%	0	0%	1291

**Table 4.1:** Confusion Matrices for datasets tested with Isolation Forest. TP = True Positives, TN = True Negatives, FP = False Positives, FN = False Negatives. Samples originated by an attacked execution are considered to be positives.

Local Outlier Factor									
Dataset	TP	TP %	TN	TN %	FP	FP %	FN	FN %	Total
<i>Spectre</i>									
<b>AES</b>	721	63,58%	328	28,92%	85	7,5%	0	0%	1134
<b>Blowfish</b>	1000	71,17%	387	27,54%	18	1,29%	0	0%	1405
<b>Idea</b>	1014	71%	400	28%	14	1%	0	0%	1428
<b>RSA</b>	775	55,91%	414	29,87%	55	3,97%	142	10,25%	1386
<i>Meltdown</i>									
<b>AES</b>	866	67,7%	356	27,83%	57	4,47%	0	0%	1279
<b>Blowfish</b>	940	69,89%	388	28,85%	17	1,26%	0	0%	1345
<b>Idea</b>	852	67,3%	403	31,83%	11	0,87%	0	0%	1266
<b>RSA</b>	830	63,9%	410	31,56%	59	4,54%	0	0%	1299

Table 4.2: Confusion Matrixes for datasets tested with Local Outlier factor. TP = True Positives, TN = True Negatives, FP = False Positives, FN = False Negatives. Samples originated by an attacked execution are considered to be positives

One-class SVM									
Dataset	TP	TP %	TN	TN %	FP	FP %	FN	FN %	Total
<i>Spectre</i>									
<b>AES</b>	714	63,35%	364	32,3%	49	4,35%	0	0%	1127
<b>Blowfish</b>	990	70,97%	391	28,03%	14	1%	0	0%	1395
<b>Idea</b>	1004	70,8%	375	26,45%	39	2,75%	0	0%	1418
<b>RSA</b>	908	65,94%	384	27,89%	85	6,17%	0	0%	1377
<i>Meltdown</i>									
<b>AES</b>	858	67,5%	337	26,51%	76	5,99%	0	0%	1271
<b>Blowfish</b>	931	69,69%	392	29,34%	13	0,97%	0	0%	1336
<b>Idea</b>	844	67,09%	383	30,45%	31	2,46%	0	0%	1258
<b>RSA</b>	822	63,67%	384	29,75%	85	6,58%	0	0%	1291

Table 4.3: Confusion Matrixes for datasets tested with One-class SVM. TP = True Positives, TN = True Negatives, FP = False Positives, FN = False Negatives. Samples originated by an attacked execution are considered to be positives

We have tested the effectiveness of our security solution also measuring how many bytes of the secret key an attack is able to extract from the cryptographical algorithm before our security module signals the ongoing attack to the Operative System. All the results presented and discussed above refer to the prediction of a single sample at a time. So, to test how many bytes of the key an attacker is able to extract before we stop it, we let run each combination of cryptographical algorithm and attack that we have for 1.6ms on gem5. The execution time was set in this manner because that is the time required to acquire a sample. After conducting these measurements, we have that none of the tested combinations leads to a single byte being extracted by the considered attack. This means that our proposed security architecture is able to detect and signal a microarchitectural cache-based side-channel attack before it can extract any useful information from the system.

For completeness, we report here also the sets of features we individuated as the most suitable for the detection of MSCAs, and that we also have used for the tests whose results we have presented and discussed above. To make them more readable we have plotted these sets in three different tables [4.4, 4.5, 4.6], one for each machine learning algorithm we have used. For every table there is a row for each of the cryptographic algorithm considered. The features in each set are not ordered in any way, and so must be intended equally important. As you can see, many of the individuated features are related to the cache, but not all of them. There are many features which refer to the branch predictor, and some, even if they are few, which refer to the number of committed instructions. Such result put more emphasis on the necessity of doing a wide search in the Hardware Performance Counters space, because the cache is not the only hardware component affected by these attacks, and focusing only on it may cut out a lot of other equally or more valid sources of information. An important example of this is the set used by Isolation Forest to identify RSA, where none of the individuated features is related to the cache.

<i>Local Outlier Factor</i>				
Dataset	Feature 1	Feature 2	Feature 3	Feature 4
AES	Alt Match prediction Hit	dcache ReadReq hits	icache overall MSHR hits	TAGE longest Match Provider on component 5
Blowfish	dcache writebacks	dcache overallAccesses	TAGE longest Match Provider on component 4	TAGE longest Match Provider on component 5
Idea	committed Int-Mult instructions	TLB accesses on write requests	ReadReq MSHR misses	memory responses ignored due to instruction squashed
RSA	dcache average miss latency	TAGE provoder for alt match on component 1	number of cycle squashed	insts committed each cycle

Table 4.4: Sets of features individuated for each dataset with Local Outlier Factor

<i>Isolation Forest</i>				
Dataset	Feature 1	Feature 2	Feature 3	Feature 4
AES	icache ReadReq misses total	dcache WriteReq mshrUncacheable	instructions issued by Float-MemWrite	icache average miss latency
Blowfish	insts committed each cycle	squashed instructions skipped in execute	dcache WriteReq accesses	iocache tag accesses
Idea	stdev of latency between load issue and its completion	icache ReadReq MSHr misses	dcache WriteReq MSHR un-cacheable	branches incorrectly predicted NotTaken
RSA	insts issued each cycle	instructions fetched each cycle	committed FloatCvt instructions	BTB lookups

Table 4.5: Sets of features individuated for each dataset with Isolation Forest

<i>One-class SVM</i>				
Dataset	Feature 1	Feature 2	Feature 3	Feature 4
AES	TAGE provider for alt match	decode detected a branch misprediction	icache overall MSHR misses	floating instruction queue wakeup accesses
Blowfish	dcache writebacks	icache read + write hits	icache ReadReq misses	insts issued each cycle
Idea	TAGE longest Match Provider on component 9	insts issued each cycle	loads that had data forwarded from stores	loads committed
RSA	dcache overall hits	branch mispredicts detected at execute	TLB misses on write requests	instructions fetched each cycle

Table 4.6: Sets of features individuated for each dataset with One-class SVM

# 5 | Hardware Implementation

This chapter is dedicated to the hardware implementation of our security system. Here we give an overview of the architecture of the module, implemented using Vivado, and we make some dimensional comparisons with some x86 processor's architectures to have a rough estimation of the area occupied by our security system.

We decided to make an hardware implementation of Isolation Forest, because it is the most promising algorithm to be used for detection, according to our experiments. For the actual implementation, we used Vivado, an industry-lead software for designing and implementing hardware components on a Field Programmable Gate Array (FPGA). We based our implementation on the Xilinx xc7z020clg484-1 board and made an implementation of one of the tree outputted by the Isolation Forest implementation offered by scikit-learn, this using the behavioral description mode of the VHDL language. An example of how a node of a tree is composed is depicted in 5.1, for the rest is a common non-balanced binary tree. The idea is that the component we implemented, that represent the tree structure, is replicated for the number of trees in the forest, and so 100 times in our case. Of course the implementation is thought to be reconfigurable because its structure changes tree by tree and from a cryptographical algorithm to another.

In Figure 5.2 is reported a schema of the inner components of the security module that we are proposing. As you can see the almost totality of the area is occupied by the instances of the tree component, one for each of the tree of the ensemble. Then we have the *predictor* which is responsible of receiving, from all the tree components, the length of the path required to isolate a given sample from the other ones, and based on these values make a mean of them and take a decision on the outlieriness of the analyzed sample. If the sample is signed as an outlier, then te predictor send a signal on the alert bus. The other two components of the module are, the timer and the features table. The timer is simply used to keep tracks of the elapsed time and send the signal to start the acquisition of a new sample, when the current time window is terminated. The configuration table, instead, memorizes the set of features to be monitored , and/or the fetching activity to be acquired through the dedicated bus.

We had to implement a standalone device, instead of picking an existing CPU and adding it our module, because x86 is a proprietary architecture and we weren't able to find any open-source implementation of an x86 processor. To make some comparisons, and understand roughly how much area our module will occupy, we used the values on the number of Look Up Tables (LUT) needed by an x86 CPU given by Wong in his doctorate thesis [49]. Wong reports in his work the occupation, in terms of LUTs, of some x86 processor's microarchitecture he implemented in an FPGA. The presented microarchitectures are not very recent, and they are based on different technologies, both in terms of bits used for memory addressing and techniques used to improve the computational power of the CPU.

To make the hardware implementation of the binary tree, we have created a finite-state machine where each state models a specific node of the tree. Such a state is defined so that it checks the node's condition to determine which will be the next node, if there is one, or which is the predicted label of the sample, if the state represents a leaf node. From the figure 5.1 is possible to see that we have to treat differently inner nodes, with respect to the leaf nodes because they have a different internal structure. The former type is characterized by 4 elements; The first one, and most important, is the feature of the sample to test, and the associated threshold value used to determine which will be the successive node. Then we have the number of samples, from the training phase, that have reached this node, and the mean outlierness value, with its squared error, associated with the samples that during the training reached this node. The leaf node, of course, does not have a splitting value based on a feature, because, when a leaf node is reached, the algorithm succeeded in separating the given sample from all the others, and can assigns it an outlierness value, that is based on the length of the path from the root node to the leaf one.

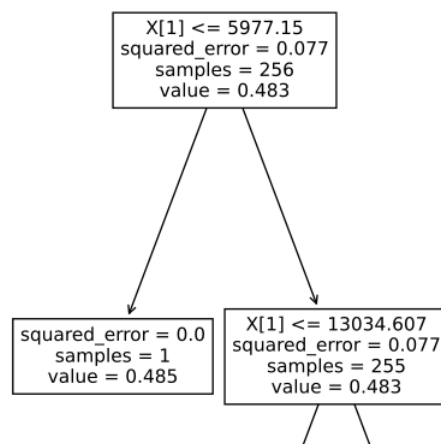


Figure 5.1: Caption



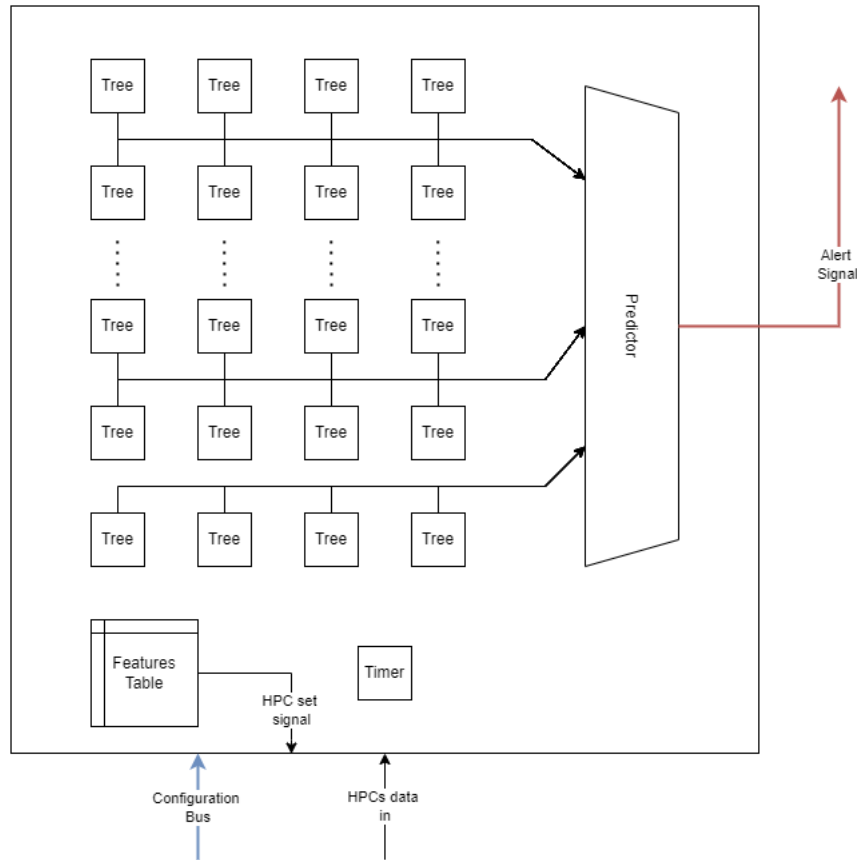


Figure 5.2: The internal schema of a module implementing Isolation Forest

After having synthesized and implemented the tree we have described, we obtained that it will utilize 452 LUTs and 61 Flip-Flops (FF). Such values are for a single tree of the ensemble that will characterize a real implementation of the security module, so to have an idea of the total amount of resources utilized by such a module we need to multiply the value we have for a single tree by 100, that is the number of trees that compose the forest used to conduct our experiments. So, we have that our security module will have a compressive dimension of 45,200 LUTs and 6,100 FFs. To give a meaning to these numbers, we compared them with the numbers of LUTs required by some microarchitecture of x86 processors that we found in the doctorate thesis of Wong [49]. Unfortunately in such a research was reported only the value for the Look Up Tables required by those processors, while no mention was done about the numbers regarding the required FFs, and so we can make a comparison based only on the required LUTs. From Wong's thesis we take into consideration for the comparison the CPU's architectures reported in the table 5.1.

Before starting the comparisons, it is important to underline that the data provided by Wong do not take into account the Level 2 cache of neither the Nehalem nor Atom

Name	Tech.	Mem. bits	Microarch.	LUTs
Intel Nehalem	45-nm	64-bits	4-wide out-of-order	670k
Intel Atom	45-nm	64-bits	2-wide in-order	176k
AMD Geode GX2	150-nm	32-bit	1-wide in-order	130k

Table 5.1: Considered x86 architecture, from the thesis [49]

microarchitectures. For what concerns the Intel Atom, it has 176k LUTs, which means that our module has an overhead of 25.7%. Instead, the Nehalem microarchitecture requires 670k LUTs, and so, if our module will be implemented in a processor which utilizes such a microarchitecture, it will have an area overhead of 6.75%. As you can see, there is a wide difference in the number of LUTs needed by a processor, this can be imputable to the different technology adopted by them, in fact, the Atom microarchitecture is based on an in-order processor while the bigger Nehalem is an out-of-order processor. This statement is supported by the fact that if we compare our module, with an even older processor's microarchitecture such as the one of AMD Geode GX2, which has 130k LUTs, we have an overhead of 34.7%. In addition to the previous consideration, when you want to make some reflections on such values of overhead, you must take into consideration that we have made a standalone implementation. This choice inevitably causes higher resource requirements in respect to an implementation that comprehends also the CPU to be protected, because in this latter case the synthesis tool can make further optimizations, reducing ulteriorly the number of LUTs required by our security module. Finally, our implementation lacks proper optimization, also from the point of view of the code that describes its behavior. The missing of such optimizations is due, on one hand, to the impossibility to find an open-source implementation of an x86 processor, and on the other hand that defining a correctly optimized hardware implementation isn't the main objective of this thesis and then is left to future works.

## 6 | Security Analysis

This chapter is dedicated to the security analysis of the proposed security module. This analysis is focused on understanding which can have been the pitfalls we have committed during all the phases of the development of the security module, and which scenarios we did not consider. Such critical points will represent a good starting point for future works which want to improve ulteriorly our results, which are already very promising and at the same level, and in some cases even better, than those ones presented in the actual state-of-the-art.

As anything in security, does not exist a perfect solution capable of ensuring security with a probability of 100%, and our module does not represent an exception to this. The first limitation that comes to mind when you study our module is that our system is strongly based on the assumption that an attack will affect, in a visible way, the values of some Hardware Performance Counters. This lead us to two possible bad scenario for our system.

The first, more probable to happen, is that someone will develop an MSCA with a very restricted footprint over the Hardware Performance Counters, and so it might be able to evade the proposed detection technique, not modifying the features selected during the training phase. In such a case a possible solution, to be explored in future works, can be to identify the specific HPCs able to detect this particular attack and understand if these counters can detect also the others attacks. If a mixed set of features can be determined, then the problem is easily solved, it is necessary to just update the set used by our security module. If instead, the discovered HPCs are very specific for the attack an alternative can be to introduce a time multiplexing and monitor alternatively the general-purpose events and the attack-specific ones. This solution is not ideal because it doubles the time needed to acquire a sample referred to the same HPCs set, but given that with the proposed dimension for the time window the attacker cannot leak anything, even if we double it, we still maintain a high level of security.

The second scenario, instead takes into consideration a new attack capable of not modifying any of the Hardware Performance Counters made available by the CPU. It's clear

that with a so advanced attacker there is nothing that our module can do to detect it. However it must be said that it's highly improbable that such attack can exist because of the elevated number of events that a processor can monitor. To be able to not leave any trace the attack basically has to have a nearly zero interaction with the targeted system. It for sure cannot interact with the cache, that is heavily monitored by HPCs, and also it has to be conservative on the number of instructions executed because they are controlled too. Of course these considerations are valid as long as we remain in the field of microarchitectural side-channel attack on which our thesis is focused.

An important criticism that can be addressed towards our work is that we have considered only two side-channel attacks. In particular, we have tested our solution only against Spectre and Meltdown, and even if we have obtained results with an accuracy close to 100%, someone could point out that this is a too restrictive test set. This choice was done mainly for two reasons; firstly it is difficult to find out some Proof Of Concept of a cache-based side-channel attack, and here I am referring to the attacks themselves, and not to the side-channel on which they are based, such as Flush+Flush or Prime+Probe. The second motivation is related to the huge amount of work, that can be needed, to modify and make working the discovered PoC on the gem5 simulator. In fact, if for Spectre was quite easy to have a functioning executable, the same cannot be said about Meltdown, which required months of work and troubleshooting to have an implementation that correctly works. This additional workload is mainly imputable to the usage of gem5, which, being a simulator, cannot have a behavior that perfectly matches the one of a real system. This is especially true if we consider that the considered attacks exploit a weird behavior of the CPU's microarchitecture, that happens naturally on a real processor, but it is harder to simulate.

Our module sends an alert signal, when it detects the signs of a cache-based side-channel attack, and the Operative System will take some blocking action as a consequence of this. Such behavior can be exploited by an attacker whose objective is to perform a Denial Of Service (DOS) on the system. If the attacker is able to create a program which mimic the execution of side-channel attack like Spectre, he is able to force the OS to take blocking actions that will result in a reduction of the computing capability of the embedded device, or, worse, in a complete stop of the normal activity of the protected system.

# 7 | Conclusions and future developments

In this thesis, we presented an innovative methodology to detect cache-based side-channel attacks, before they can harm the system. The proposed solution is machine learning-based, and, because it is oriented towards embedded systems, it is encapsulated in a dedicated hardware module, so that it will not interfere with the normal activity of the system it is protecting. The detection is made by monitoring some carefully selected Hardware Performance Counters at a fixed sampling frequency. For the sampling, we made use of gem5, which gives us the possibility to retrieve all the HPCs available, and then we took a decision on which of those to use. After having tested multiple One-class ML algorithms, we find out that the best one is Isolation Forest, for which we also provided an implementation on an FPGA. With this ML algorithm, we can detect an attack with an accuracy of more than 99% in all the tested scenarios, and with a false positive rate lower than 1% and zero false negatives. The proposed hardware implementation causes an overhead of around 25%, but it is a non-optimized implementation, so there is a lot of room for improvements under this aspect.

Future works, in the light of the above, must optimize the hardware implementation, and maybe also try to implement another ML algorithm such as One-class SVM, that offers also good results, with an accuracy that is every time in the neighborhood of 99%. Further researches can be done on the utility of the fetching activity, whose data, even if present in the feature space considered, it is never resulted in any of the features set used. Anyway, due to how we structured our hardware module, it is possible to retrieve the fetching activity independently from the HPCs, and without the associated limitation on the number of monitorable ones.

Additional research can be done to better stress out our security system, with the testing of newer cryptographical algorithms, to further expand the set of supported ones, or testing others implementation of already tested algorithms. This will improve the usability of the system in a real scenario, giving to the final users a database of configurations that

covers the vast majority of the cryptographical algorithms utilized. Another direction in which our research can be expanded is toward adding others attacks, to better test the effectiveness of our security module, and especially the goodness of the choices on the sets of features to monitor.

Future works can also focuses on expanding further the search space of the features sets, because in our work we applied a filter to it, and we have also generated only 200 random sets, for each combination of ML algorithm and cryptographical algorithm. Someone could try to generate all the possible combinations of four features, after having applied the filter, and see if maybe exists some other sets that performs better than the ones we have discovered. Otherwise there is also the possibility to remove completely our filter, or maybe testing various configurations of it, with the objective of understanding which are the more reasonable bounds for the variance's value. This research can be useful because we have set the boundaries using some values that seemed logical to us, but we have never done a precise and deep enough study to be sure that those boundaries values are the more correct ones.

## Bibliography

- [1] M. Alam, S. Bhattacharya, D. Mukhopadhyay, and S. Bhattacharya. Performance counters to rescue: A machine learning based safeguard against micro-architectural side-channel-attacks. *IACR Cryptol. ePrint Arch.*, 2017:564, 2017.
- [2] AMD. *Open-Source Register Reference for AMD Family 17h Processors*. AMD, 2018. URL [https://developer.amd.com/wp-content/resources/56255\\_3\\_03.PDF](https://developer.amd.com/wp-content/resources/56255_3_03.PDF).
- [3] K. Arikan, A. Palumbo, L. Cassano, P. Reviriego, S. Pontarelli, G. Bianchi, O. Ergin, and M. Ottavi. Processor security: Detecting microarchitectural attacks via count-min sketches. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, (01):1–14, may 5555. ISSN 1557-9999. doi: 10.1109/TVLSI.2022.3171810.
- [4] M.-M. Bazm, T. Sautereau, M. Lacoste, M. Sudholt, and J.-M. Menaud. Cache-based side-channel attacks detection through intel cache monitoring technology and hardware performance counters. In *2018 Third International Conference on Fog and Mobile Edge Computing (FMEC)*. IEEE, apr 2018. doi: 10.1109/fmec.2018.8364038.
- [5] R. A. Berk. Support vector machines. In *Statistical Learning from a Regression Perspective*, pages 1–28. Springer New York, 2008. doi: 10.1007/978-0-387-77501-2\_7.
- [6] D. J. Bernstein. Cache-timing attacks on aes. 2005.
- [7] J. Bonneau and I. Mironov. Cache-collision timing attacks against aes. In L. Goubin and M. Matsui, editors, *Cryptographic Hardware and Embedded Systems - CHES 2006*, pages 201–215, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-46561-4.
- [8] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander. Lof: identifying density-based local outliers. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, SIGMOD '00, page 93–104, New York, NY, USA, 5 2000. Association for Computing Machinery. ISBN 1581132174. doi: 10.1145/342009.335388. URL <https://doi.org/10.1145/342009.335388>.

- [9] E. Brier, C. Clavier, and F. Olivier. Correlation power analysis with a leakage model. In *Lecture Notes in Computer Science*, pages 16–29. Springer Berlin Heidelberg, 2004. doi: 10.1007/978-3-540-28632-5\_2.
- [10] M. Chiappetta, E. Savas, and C. Yilmaz. Real time detection of cache-based side-channel attacks using hardware performance counters. *Applied Soft Computing*, 49: 1162–1174, dec 2016. doi: 10.1016/j.asoc.2016.09.014.
- [11] J. Depoix and P. Altmeyer. Detecting spectre attacks by identifying cache side-channel attacks using machine learning. *Advanced Microkernel Operating Systems*, 75, 2018.
- [12] M. R. Fadiheh, D. Stoffel, C. Barrett, S. Mitra, and W. Kunz. Processor hardware security vulnerabilities and their detection by unique program execution checking. Dec. 2018.
- [13] H. Gamaarachchi and H. Ganegoda. Power analysis based side channel attack, 2018.
- [14] Q. Ge, Y. Yarom, D. Cock, and G. Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, 8(1):1–27, dec 2016. doi: 10.1007/s13389-016-0141-6.
- [15] gem5. Gem5 homepage. URL <https://www.gem5.org/>.
- [16] Gem5Art. Full system tutorial. URL <https://gem5art.readthedocs.io/en/latest/tutorials/boot-tutorial.html>.
- [17] GitHub. Cryptographical algorithms implementations, . URL <https://github.com/B-Con/crypto-algorithms>.
- [18] GitHub. Meltdown poc, . URL <https://github.com/IAIK/meltdown>.
- [19] GitHub. Spectre poc, . URL <https://gist.github.com/ErikAugust/724d4a969fb2c6ae1bbd7b2a9e3d4bb6>.
- [20] M. Godfrey and M. Zulkernine. Preventing cache-based side-channel attacks in a cloud environment. *IEEE Transactions on Cloud Computing*, 2(4):395–408, oct 2014. doi: 10.1109/tcc.2014.2358236.
- [21] D. Gruss, C. Maurice, K. Wagner, and S. Mangard. Flush+flush: A fast and stealthy cache attack. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299. Springer International Publishing, 2016. doi: 10.1007/978-3-319-40667-1\_14.



- [22] Y. ichi Hayashi, N. Homma, T. Mizuki, H. Shimada, T. Aoki, H. Sone, L. Sauvage, and J.-L. Danger. Efficient evaluation of EM radiation associated with information leakage from cryptographic devices. *IEEE Transactions on Electromagnetic Compatibility*, 55(3):555–563, jun 2013. doi: 10.1109/temc.2012.2222890.
- [23] Intel. Intel ark page i5-3470, 2012. URL <https://ark.intel.com/content/www/it/it/ark/products/68316/intel-core-i53470-processor-6m-cache-up-to-3-60-ghz.html>.
- [24] Intel. Intel® 64 and ia-32 architectures software developer’s manual, volume 3, Apr. 2022. URL <https://cdrdv2.intel.com/v1/dl/getContent/671447>.
- [25] Intel. Intel performance events, 2022. URL <https://perfmon-events.intel.com/>.
- [26] G. Keramidas, A. Antonopoulos, D. N. Serpanos, and S. Kaxiras. Non deterministic caches: a simple and effective defense against side channel attacks. *Design Automation for Embedded Systems*, 12(3):221–230, apr 2008. doi: 10.1007/s10617-008-9018-y.
- [27] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, may 2019. doi: 10.1109/sp.2019.00002.
- [28] J. Kong, O. Aciicmez, J.-P. Seifert, and H. Zhou. Hardware-software integrated approaches to defend against software cache-based side channel attacks. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. IEEE, feb 2009. doi: 10.1109/hpca.2009.4798277.
- [29] Y. Kulah, B. Dincer, C. Yilmaz, and E. Savas. SpyDetector: An approach for detecting side-channel attacks at runtime. *International Journal of Information Security*, 18(4):393–422, jun 2018. doi: 10.1007/s10207-018-0411-7.
- [30] E. Käsper and P. Schwabe. Faster and timing-attack resistant AES-GCM. In *Lecture Notes in Computer Science*, pages 1–17. Springer Berlin Heidelberg, 2009. doi: 10.1007/978-3-642-04138-9\_1.
- [31] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, M. Hamburg, and R. Strackx. Meltdown: Reading kernel memory from user space. *Commun. ACM*, 63(6):46–56, may 2020. ISSN 0001-0782. doi: 10.1145/3357033. URL <https://doi.org/10.1145/3357033>.
- [32] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee. CATalyst:

- Defeating last-level cache side channel attacks in cloud computing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, mar 2016. doi: 10.1109/hpca.2016.7446082.
- [33] F. T. Liu, K. M. Ting, and Z.-H. Zhou. Isolation forest. In *2008 Eighth IEEE International Conference on Data Mining*. IEEE, dec 2008. doi: 10.1109/icdm.2008.17.
- [34] J. Longo, E. D. Mulder, D. Page, and M. Tunstall. SoC it to EM: ElectroMagnetic side-channel attacks on a complex system-on-chip. In *Lecture Notes in Computer Science*, pages 620–640. Springer Berlin Heidelberg, 2015. doi: 10.1007/978-3-662-48324-4\_31.
- [35] J. Lowe-Power. Reproducing spectre on gem5. URL <http://www.lowepower.com/jason/visualizing-spectre-with-gem5.html>.
- [36] Y. Lyu and P. Mishra. A survey of side-channel attacks on caches and countermeasures. *Journal of Hardware and Systems Security*, 2(1):33–50, nov 2017. doi: 10.1007/s41635-017-0025-y.
- [37] S. Mirbagher-Ajorpaz, G. Pokam, E. Mohammadian-Koruyeh, E. Garza, N. Abughazaleh, and D. A. Jimenez. PerSpectron: Detecting invariant footprints of microarchitectural attacks with perceptron. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, oct 2020. doi: 10.1109/micro50266.2020.00093.
- [38] A. Moghimi, J. Wichelmann, T. Eisenbarth, and B. Sunar. MemJam: A false dependency attack against constant-time crypto implementations. *International Journal of Parallel Programming*, 47(4):538–570, nov 2018. doi: 10.1007/s10766-018-0611-9.
- [39] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: The case of AES. In *Topics in Cryptology – CT-RSA 2006*, pages 1–20. Springer Berlin Heidelberg, 2006. doi: 10.1007/11605805\_1.
- [40] D. Page. Defending against cache-based side-channel attacks. *Information Security Technical Report*, 8(1):30–44, mar 2003. doi: 10.1016/s1363-4127(03)00104-3.
- [41] C. Percival. Cache missing for fun and profit, 2005.
- [42] A. Sayakkara, N.-A. Le-Khac, and M. Scanlon. A survey of electromagnetic side-channel attacks and discussion on their case-progressing potential for digital forensics. *Digital Investigation*, 29:43–54, jun 2019. doi: 10.1016/j.diin.2019.03.002.

- [43] B. Schölkopf, J. C. Platt, J. Shawe-Taylor, A. J. Smola, and R. C. Williamson. Estimating the support of a high-dimensional distribution. *Neural Computation*, 13(7):1443–1471, jul 2001. doi: 10.1162/089976601750264965.
- [44] scikit learn. scikit-learn homepage. URL <https://scikit-learn.org/stable/index.html>.
- [45] R. Sprabery, K. Evchenko, A. Raj, R. B. Bobba, S. Mohan, and R. Campbell. Scheduling, isolation, and cache allocation: A side-channel defense. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, apr 2018. doi: 10.1109/ic2e.2018.00025.
- [46] StackOverflow. Boot drive. URL <https://stackoverflow.com/questions/37906425/booting-gem5-x86-ubuntu-full-system-simulation>.
- [47] H. Wang, H. Sayadi, A. Sasan, S. Rafatirad, T. Mohsenin, and H. Hodayoun. Comprehensive evaluation of machine learning countermeasures for detecting microarchitectural side-channel attacks. In *Proceedings of the 2020 on Great Lakes Symposium on VLSI, GLSVLSI '20*, page 181–186, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450379441. doi: 10.1145/3386263.3407586. URL <https://doi.org/10.1145/3386263.3407586>.
- [48] Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. *ACM SIGARCH Computer Architecture News*, 35(2):494–505, jun 2007. doi: 10.1145/1273440.1250723.
- [49] H. T.-H. Wong. A superscalar out-of-order x86 soft processor for fpga. Master’s thesis, University of Toronto, 2017.
- [50] Y. Yarom and K. Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *Proceedings of the 23rd USENIX Conference on Security Symposium, SEC’14*, page 719–732, USA, 2014. USENIX Association. ISBN 9781931971157.
- [51] Y. Yarom, D. Genkin, and N. Heninger. CacheBleed: a timing attack on OpenSSL constant-time RSA. *Journal of Cryptographic Engineering*, 7(2):99–112, feb 2017. doi: 10.1007/s13389-017-0152-y.
- [52] X. Yu, Y. Xiao, K. Cameron, and D. D. Yao. Comparative measurement of cache Configurations’ impacts on cache timing Side-Channel attacks. In *12th USENIX Workshop on Cyber Security Experimentation and Test (CSET 19)*, Santa Clara, CA, Aug. 2019. USENIX Association. URL <https://www.usenix.org/conference/cset19/presentation/yu>.

- [53] Y. Zhang and M. K. Reiter. Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 827–838, 2013.
- [54] Z. Zhou, M. K. Reiter, and Y. Zhang. A software approach to defeating side channels in last-level caches. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS '16*, page 871–882, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450341394. doi: 10.1145/2976749.2978324. URL <https://doi.org/10.1145/2976749.2978324>.

# A | Gem5 Simulator

This appendix is dedicated to the architectural simulator gem5, which is an important component of our research; in fact, we used it to acquire the samples of all the HPCs offered by the processor, and that we later used to train the machine learning algorithms and also test them. We thought that a guide on how to use such a powerful simulator may be helpful given the various obstacles we have encountered during its use.

The gem5 simulator is an open-source simulator, that is written in C++, which wants to simulate precisely a processor architecture for research reasons. At the moment it supports different, commonly used, architectures at various levels of details, according to the official website [15], the current support status is:

- ARM: which is fully supported, and able to boot unmodified versions of Linux and Android.
- x86: fully supported, and able to simulate a common PC in its entirety. It can boot unmodified versions of Linux.
- RISC-V: this is a work in progress, especially for what concerns the privileged instructions of this architecture.
- SPARC: with this architecture is able to boot Solaris similarly on how also other simulators of such architecture do.
- Alpha: they reproduced in a sufficiently accurate manner a DEC Tsunami system so that you can boot unmodified versions of Linux.

The first thing to do, of course, is to download the source code and build the simulator. The authors of gem5 offer two possibilities to do this, you can simply download the source code from their git repository and build it, or they offer a pre-built Docker image. In our work we opted to set up a virtual machine, then download and build directly the source code. Even if they claim that any Linux distribution, with the requested dependencies, can be used, turns out that this is not correct. In fact, we have tried to build gem5 on a newer version of Ubuntu with respect to the one suggested, but we did not obtain a

working version of the simulator, so we revert to the suggested Ubuntu 20.04.

After having installed the dependencies required by gem5, it is time to compile it. Before doing so, there are two choices you have to pick, the architecture to simulate and the build variant. For what concern the architecture, you have to choose one among the available ones, previously listed. While the build variant affects the speed of the simulation, but also the debugging option available; gem5 offers three build variants. The *debug*: this has all the optimizations turned off, it is the slowest one, in fact, it's suggested only to those that need to use some debug tool like gdb. The *opt* is the balanced one, it has the optimizations turned on, but still maintains some debugging features, this makes it possible to increase the simulation speed, and in the meanwhile maintains some internal information in case something goes wrong, This is the variant suggested by the developers, and the one we have used for our tests. The last variant is the *fast*, which has the optimizations turned on, and no support for debugging, making it the fastest one, but also the more unstable one. In any case is possible to have multiple builds, with support for different architectures and build variants. Once these two choices are made you can proceed with the build command that is the following:

```
scons build/{ISA}/gem5.{variant} -j {cpus}
```

where the *cpus* is the number of cores to use for the building, to speed it up; in any case, this step can take up to an hour to be completed, depending on the computational power of the PC you use. At this point, you have your simulator is ready to be used.

The next step is to create a configuration, this is done independently by the emulation mode you want to use. Such a configuration file is written in python, and inside it, you have to specify all the elements that compose the processor you want to simulate. In such a file, you can also take an element, that was defined by the developers, and define a new class that extends it. Another possibility is to create a file, that instead of having a fixed configuration, offers multiple options for the various components in the form of parameters, in this way is possible to set up the processor's configuration when you start the simulation. This last option is the one adopted also by the developers that made available a couple of configuration files ready to be used, and with a lot of parameters available.

Once you have the configuration file, the next step is to decide which type of simulation you want to run, the Syscall Emulation one, or the Full System. The first option is the fastest and also the easiest to run, it executes a given program simulating all the system call it does, and so its usage is limited by the syscalls that the gem5's developers had implemented or the ones you have added. The command to run it is the following,

assuming you are in the folder where you built gem5, and you opted for x86 architecture and opt variant:

```
build/X86/gem5.opt configs/example/se.py -d [destination] [options]
-c [source]
```

where *destination* is an optional parameter that is used to define a folder where all the files generated by the simulation are put, otherwise the default folder is called *m5out* and is positioned in the main gem5's folder. The simulation's files comprehend a description of the configuration of the simulated CPU, both a textual one, that a graphical one. A basic example of a configuration is depicted in Figure A.1.

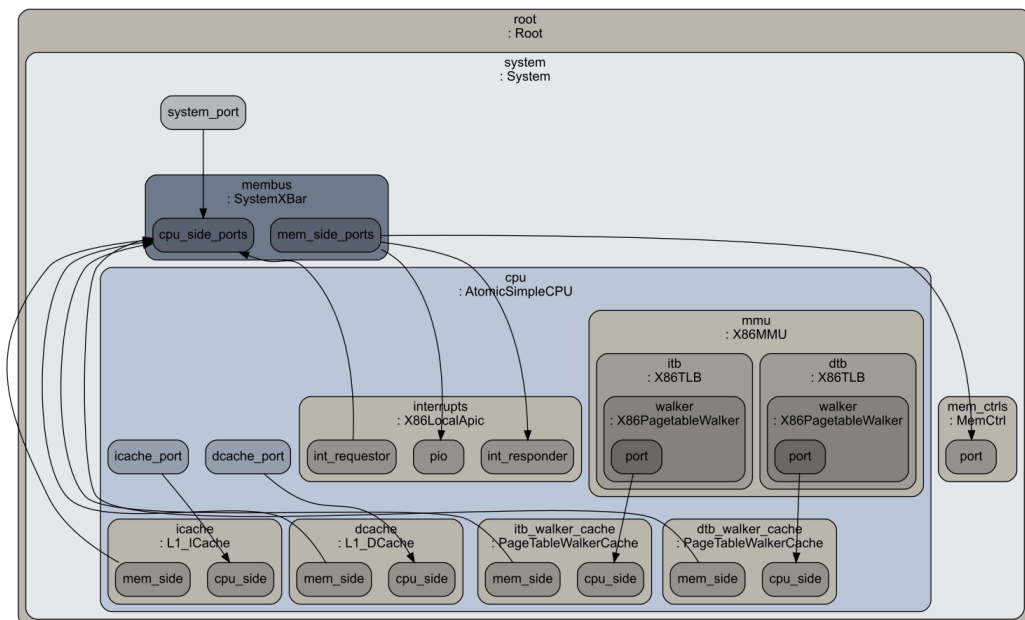


Figure A.1: gem5 configuration example

The *source* parameter is the path to the executable you want to run with gem5, if your program expects some inputs when you run it, you can pass these with the option *-o* followed by all the inputs enclosed by `" "`.

The *options* field, instead contains all the parameters for the processor's configuration. These comprehend basic options such as the number of cores or the CPU's frequency. The configuration file provided by the developers provides also some options on the RAM setup, you can define its size, and the number of memory channels, and then there is a list of different memory types which differ by the technology on which they are based (e.g. DDR3, DDR4, LPDDR5, HBM) and also by their frequency. Of course, if none of the provided memory options satisfy your needs, there is the possibility to define your

own memory configuration and import it into the CPU's configuration file. Other options regard the cache; for this, they provide support for a three-level cache, to use it, you have to enable it with the option `-caches`. Then the configuration file provides support for a first cache level divided into an instruction cache and data cache, and then for two more homogeneous cache levels, you can enable only the levels you want. The options relating to the cache are about their size and their associativity, this is independently configurable for each level, while the line size is unique for all the levels. There are options for choosing which type of branch predictor to use, and also for which prefetcher to adopt, this last one can be set independently for the l1-icache, l1-dcache, and the level 2 cache.

If you want to use gem5 in its Full System emulation mode, instead, things become more complicated. This modality requires, in addition to the configuration file, also a Linux kernel (which with the x86 architecture does not require any sort of modification) and a boot drive with the Operative System. There are a lot of guides on the Internet on how to set up and run a Full System emulation, but all of them seem to be somehow incomplete, or with certain steps that do not work when you try to reproduce them. One of the most complete is the one written by the gem5 developers and can be found here [16]. In this guide is explained how to create a boot drive with Ubuntu, how to create a boot drive with Ubuntu, but the proposed method does not work if you are on a virtual machine, and the provided disk image is configured to shut down immediately after it has completed the boot, this makes it unusable. Anyway, we succeeded in creating a boot drive with Ubuntu, but when we tried to boot it, after 24 hours the simulation was stuck. After searching for a solution to this, we have found one on a StackOverflow thread [46] where there is a link to a boot drive based on Gentoo.

The next step is to find a Linux kernel that works with gem5 and the boot drive you have; for this, we went back to the guide on gem5art, where there are some kernels already compiled, and also the instructions on how to compile one by yourself. Sadly, not all the kernels already compiled work with our setup, and we opted for the kernel named *vmlinux-4.4.186*, which by the way is a 5.2.3 kernel. In fact, many of the proposed kernels do not start at all, or in other cases, they start to boot up the system, but at a certain point, they will crash. An interesting thing reported in such a guide is a group of graphs that report the compatibility between the kernels tested by the developers with different configurations of CPU type used and memory type selected, from which is evident that it is not easy to find a working setup, and multiple tests may be needed.

Once you have retrieved all the required elements you can proceed with the simulation, which can be performed with the following command:



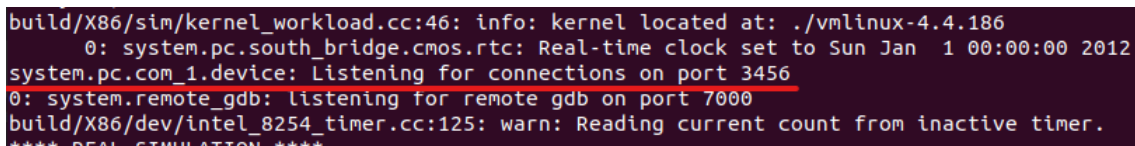
```
build/X86/gem5.opt -d [out_dir] configs/example/fs.py --disk-image
[boot_drive] --kernel [kernel] [options]
```

Here again, it is supposed that you run such a command from the main folder of gem5 and that you will use the configuration file given by the developers. In this case is strongly suggested to use the already provided configuration file because otherwise you have to manage all the procedures to correctly interconnect the various processor's components to be able to start a Full System emulation, and such procedures are not well documented. The *out\_dir* and *options* parameters are the same as the Syscall emulation, but with this configuration file, we have two other parameters that you must provide, which are the kernel and the boot drive files, respectively specified by *kernel* and *boot\_drive*.

Once you have started the simulation you can connect to the simulated system from remote with another terminal, that from now on we refer to as the remote terminal, with the following command, intended to be launched by the main folder of gem5:

```
util/term/m5term localhost [port]
```

Where the port can be founded on the first terminal, that one where you have executed the command to launch the simulation and that from now on we refer to as the control terminal. After a bit it shows a line which indicates the port to use, by default it is the 3456. This line is the one underlined in the Figure A.2.



```
build/X86/sim/kernel_workload.cc:46: info: kernel located at: ./vmlinux-4.4.186
0: system.pc.south_bridge.cmos.rtc: Real-time clock set to Sun Jan 1 00:00:00 2012
system.pc.com_1.device: Listening for connections on port 3456
0: system.remote_gdb: listening for remote gdb on port 7000
build/X86/dev/intel_8254_timer.cc:125: warn: Reading current count from inactive timer.
**** REAL SIMULATION ****
```

Figure A.2: Line indicating which is the port to connect to the Full System emulation

When the boot phase is complete you can make a checkpoint to save the current status of the system, so that the next time you can restart from here, instead of having to wait again for the system to boot up. To create such a checkpoint you have to execute the command `m5 checkpoint` on the remote terminal, this will create a folder inside the output folder of the simulation, you can create multiple checkpoints for the same simulation. Then, to resume from a checkpoint, that are time ordered, you need to add some parameters to the command used to launch the simulation, it must be underlined that such a possibility is given by the fact that we are using the developers' provided configuration file. The parameters to be added are:

```
-r [checkpoint] --checkpoint-dir [simul_dir]
```

Where the *simul\_dir* is the directory that contains the checkpoints folders, and *checkpoint* is an integer that indicates to the simulator which checkpoint to resume, it starts from 1 and points to the i-th checkpoint's folder.

## List of Figures

1.1	Address mapping from main memory to cache, image re-adapted from [20]	13
1.2	Time distribution for all the possible values of $n$ [13] normalized over the mean time, taken from [6]	16
3.1	High Level schema of the proposed security module	38
4.1	Metrics for all the datasets tested with Isolation Forest	47
4.2	Metrics for all the datasets tested with One-class Support Vector Machine	48
4.3	Metrics for all the datasets tested with Local Outlier Factor	49
5.1	Caption	56
5.2	The internal schema of a module implementing Isolation Forest	57
A.1	gem5 configuration example	71
A.2	Line indicating which is the port to connect to the Full System emulation	73



## List of Tables

4.1	Confusion Matrices for datasets tested with Isolation Forest. TP = True Positives, TN = True Negatives, FP = False Positives, FN = False Negatives. Samples originated by an attacked execution are considered to be positives. . . . .	50
4.2	Confusion Matrixes for datasets tested with Local Outlier factor. TP = True Positives, TN = True Negatives, FP = False Positives, FN = False Negatives. Samples originated by an attacked execution are considered to be positives . . . . .	51
4.3	Confusion Matrixes for datasets tested with One-class SVM. TP = True Positives, TN = True Negatives, FP = False Positives, FN = False Negatives. Samples originated by an attacked execution are considered to be positives . . . . .	51
4.4	Sets of features individuated for each dataset with Local Outlier Factor . .	53
4.5	Sets of features individuated for each dataset with Isolation Forest . . . . .	53
4.6	Sets of features individuated for each dataset with One-class SVM . . . . .	54
5.1	Considered x86 architecture, from the thesis [49] . . . . .	58



## Acknowledgements

