TU WIEN Informatics

# Erkennung von Angriffen mit mikroarchitektonischen Zuständen und Maschinellem Lernen

MASTERARBEIT

zur Erlangung des akademischen Grades

**Master of Science**

im Rahmen des Studiums

**Technische Informatik**

eingereicht von

**Mariana da Silva Barros**
Matrikelnummer 12202389

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ. Prof. Dipl.-Ing. Georg Weissenbacher, D.Phil.
Mitwirkung: Alexander Pluska, MSc
Mai Al-Zubi, MSc

Wien, 8. September 2025

| Mariana da Silva Barros | Georg Weissenbacher |
|---|---|

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

# Attack Detection with Microarchitectural Traces and Machine Learning

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

**Master of Science**

in

**Computer Engineering**

by

**Mariana da Silva Barros**
Registration Number 12202389

to the Faculty of Informatics

at the TU Wien

Advisor: Univ. Prof. Dipl.-Ing. Georg Weissenbacher, D.Phil.
Assistance: Alexander Pluska, MSc
Mai Al-Zubi, MSc

Vienna, September 8, 2025

| Mariana da Silva Barros | Georg Weissenbacher |
|---|---|

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

# Erklärung zur Verfassung der Arbeit

Mariana da Silva Barros

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 8. September 2025

Mariana da Silva Barros

# Acknowledgements

First of all, I want to thank God for giving me the strength to overcome my limitations and don't give up. Without Him, I would not have arrived where I am today.

During my whole life and especially throughout my academic journey, the experiences I have lived and the people I have met made me who I am today. Therefore, I want to thank everyone in my life for all the support and help I have received. I thank my parents, Edna and Antônio, for being the basis that made me arrive here. I also thank my brother, Tiago, for all the encouragement given when I needed it.

I would like to thank my advisor, Prof. Georg Weissenbacher, for guidance and support during the development of this work. I also would like to thank Mai Al-Zubi and Alexander Pluska for their assistance during the process of writing this thesis. I also thank the Technical University of Vienna (TU Wien) for providing the education, opportunities, and infrastructure that allow the formation of so many people.

More important than where we arrive at the end of a path are the friendships that we make during the journey. I would like to thank my friends from IBCV for all the encouragement, support, and help along the way, and for helping me every day to become a better person.

# Kurzfassung

Optimierungen in modernen Rechnerarchitekturen ermöglichen einen schnelleren und effizienteren Informationszugriff, erhöhen jedoch gleichzeitig die Anfälligkeit für Angriffe und Informationslecks. Diese Schwachstellen werden unter anderem durch cachebasierte Seitenkanalangriffe (Side Channel Attacks) ausgenutzt, die gemeinsame Hardware-Ressourcen und Optimierungen der Rechnerarchitektur einsetzen, um gezielt mikroarchitektonische Seiteneffekte zu erwirken. Zahlreiche aktuelle Forschungsergebnisse zur Erkennung und Abwehr dieser Bedrohung fokussieren sich auf die Verwendung von Hardware Performance Counters (HPCs) zur indirekten Überwachung solcher Seiteneffekte. Eine zentrale Herausforderung besteht jedoch in der mangelnden Flexibilität der Überwachungsintervalle, die entweder Angriffe übersehen oder erheblichen Overhead erzeugen können, sowie in der begrenzten Generalisierbarkeit der erlernten Modelle. Diese Arbeit verfolgt das Ziel, cachebasierte Seitenkanalangriffe mittels maschinellen Lernens zu erkennen, indem ein Grenzwert-basiertes Verfahren zur Überwachung von HPCs eingesetzt wird. Sequenzen von Zeitintervallen zwischen den Überschreitungen von (automatisch ermittelten) Grenzwerten der HPCs dienen als Eingabe für ein T-LSTM-Modell, das zur Angriffserkennung eingesetzt wird. Experimente auf verschiedenen Plattformen und Architekturen und in unterschiedlichen Szenarien dienen der Evaluierung der Generalisierungsfähigkeit des Modells. Optimierungstechniken wie Normalisierung und Fine-Tuning werden eingesetzt, um die Leistungsfähigkeit weiter zu steigern. Das vorgeschlagene System erreicht eine Erkennungsgenauigkeit von über 99% im besten Fall (plattformspezifisch), sowie 98% in plattformübergreifenden Cross-Validierungs-Szenarien unter Verwendung von Fine-Tuning.

# Abstract

Recent advances in computer architecture make access to information faster and more efficient, but also make the computer prone to attacks and information leakage. This vulnerability is exploited, for example, by cache-based side-channel attacks, which make use of shared hardware resources and optimizations in the machine, and affect its microarchitectural traces. Recent studies on the detection and mitigation of this threat focus on the monitoring of Hardware Performance Counters (HPCs). However, among the faced challenges is the lack of flexibility on the monitoring time interval, which may miss attacks or generate a substantial overhead, as well as the lack of generalization for the learned model. Therefore, this work aims to detect cache side-channel attacks with a machine learning technique by using an overflow-based approach to monitor HPCs. The dataset composed by the sequence of triggered overflows for the HPC values, along with the time interval between overflows, is submitted as input to a T-LSTM model, which is trained to detect cache side-channel attacks. The experiments conducted in different scenarios and platforms aim to assess the model's generalization ability. Optimization techniques, as normalization and fine-tuning, are used to improve the model's performance. The detection accuracy of the proposed system is of over 99% in the best scenario (separate platforms), and 98% on the cross-validation across platforms scenario with the use of fine-tuning.

# Contents

CHAPTER 1

# Introduction

## 1.1 Motivation and Problem Statement

Recent advances in computer architectures make access to information faster and more efficient. Among the introduced optimizations, we can cite the use of caches, speculative execution, and CPU pipelines, for example. However, they also make computers more prone to attacks and information leakage. A recent type of attack that poses a threat to most modern architectures is cache-based side-channel attacks, which are efficient for the fact that they don't need additional devices or physical contact with the target [TZW+20].

This type of attack makes use of the shared hardware resources in the machine, such as memory and cache, and also of the optimizations on the access to these resources. Among them, the Spectre vulnerability is one example of a side-channel attack that exploits optimizations found in modern hardware architectures, such as branch prediction and speculative execution. This kind of attack makes use of these features to leak confidential information via side-channels [KHF+20]. As a result, side-channel attacks (and particularly Spectre Attacks) have an effect on microarchitectural traces of the machine, which are reflected on Hardware Performance Counters (HPC).

In the last years, techniques to identify when the machine is under attack have been developed, based on both software and hardware. Some of the most popular early approaches are: timing analysis, cache usage profiling, statistical analysis, specialized detection tools, or analysis of system behavior [SDO23]. However, recently, the use of Artificial Intelligence has also helped in this task. According to Al-Zu'bi [AZW24], a popular approach to detect such attacks deploys Machine Learning (ML) to identify suspicious micro-architectural patterns. This can be done, for example, by monitoring the machine HPCs that are affected by this kind of attack, such as branch mispredictions or cache misses.

However, even though there are significant efforts in creating mechanisms to detect cache side-channel attacks, there are still limitations associated with this task, as it is observed in several studies developed in recent years. Gulmezoglu et al. [GMES19] affirms that the main challenge is the fact that the existing works are based on existing attacks, and therefore lack the knowledge of new approaches. Tong et al. [TZW+20] and Mukhtar et al. [MMB+20] state that the overhead in detection systems still causes a lot of damage to performance. In addition, the fact that the attacks are also evolving and adapting to avoid detection makes it difficult to identify them using conventional techniques [SDO23]. Furthermore, the usual mechanisms created with this purpose are limited to a single architecture or require radical hardware modifications [MMB+20].

According to Kim et al. [KHK+24], cache side-channel attacks can be designed differently based on the target system environments, such as processor architecture, for example. Therefore, another main challenge related to the use of microarchitectural traces, such as Hardware Performance Counters, to detect Spectre attacks, is to build a model that generalizes well and is robust for different platforms. Since the affected traces are related to the computer architecture, the values will differ in different platforms, and this may make it more difficult for the model to detect attacks in these situations.

This work brings an approach to detect Spectre attacks by observing microarchitectural traces of the computer and using machine learning techniques. Hardware Performance Counters that are most affected by the side-channel attacks will be monitored, and when they reach a certain value (a pre-defined threshold), it may indicate that the machine is under attack. This case constitutes an overflow, and this information is provided to a detection system that will evaluate if the situation constitutes a threat to the computer. Since microarchitectural traces usually vary with different computer architectures, one of the main challenges is related to whether the detection model can be transferred across different platforms.

## 1.2 Aim of the Work

This work aims to provide a system that can detect the presence of cache side-channel attacks through the monitoring of microarchitectural features of the machine, using an overflow-based approach. The measured data should be analyzed over time and evaluated using Machine Learning techniques. In order to assess the robustness of the model, the detection task will be validated on different platforms.

The detection system has the goal to constantly monitor the Hardware Performance Counters and, based on the measured values and the frequency that they overcome pre-defined thresholds, identify whether the machine is under attack. Each time there is a threshold overflow at one of the measured HPCs, this information is sampled and sent to a deep learning model, which will assess if it is a benign application or a side-channel attack. This technique enables the data collection at a dynamic pace, depending on the processes running in the machine, and, at the same time, ensures a constant stream of data to the detection model. As a result, the performance overhead is expected to be

smaller than the approach of sampling at a fixed time interval, which was performed in previous works.

However, the system must be robust enough to perform the detection in different architectures and load situations. Due to the fact that the monitored features are microarchitectural traces, they might differ across different architectures, which makes it a challenge to transfer the detection system across platforms. This work will evaluate whether this is possible and will try to overcome this obstacle through the use of an adapted LSTM model that dynamically adapts the learning with the input timing information. In addition, the calibration phase aims to find threshold values that enable the system to adapt to different architectures, while still being able to detect attacks. The system will also be evaluated with different benign applications, to guarantee that it is robust enough to detect attacks with different workload intensities.

The following research questions (RQs) should be answered at the end of this work:

- How well does the system generalize? In other words, is it possible to transfer the detection across different platforms?
- Is it possible to find thresholds for an unknown platform such that the system can detect Spectre Attacks? In other words, is it possible to calibrate the system for the microarchitectural features of each machine?
- Does the timing information on the measurements help to detect the attacks? In other words, does the LSTM model with dynamic temporal adaptation yield good results in this application?
- Does the calibration target influences in the learning processes?
- Do the normalization and fine-tuning techniques improve the system's performance?
- What is the minimum dataset size necessary for the fine-tuning step?

## 1.3 Methodological Approach

The methodological approach consists of the following steps:

1. **Literature Review**
   Theoretical background and related works must be researched in order to understand the state-of-the-art, the techniques that have already been used to detect attacks, and the achieved performance.
2. **Datasets Creation**
   Before training and testing the model that will detect the attacks, we need to generate datasets both in attack situations and while running benign applications.

These datasets are composed of samples taken each time a Hardware Performance Counter triggered an overflow, or, in other words, each time one of the monitored HPC values was above a previously defined threshold.

- **Calibration Phase**
  Since different platforms have different standard values for microarchitectural traces, this phase is necessary in order to define the thresholds for each monitored Hardware Performance Counter. This is done by sampling the HPCs for each architecture in various benign scenarios and selecting the thresholds that would allow us to detect attacks.
- **Overflow-based Dataset Creation**
  We generate the datasets for training and evaluation of the model under different workloads, for the different platforms, using the thresholds determined in the previous step. Each sample of the dataset reflects an overflow triggering from one of the monitored HPCs, per process running in the machine. In addition to the overflow trigger, the sample also includes the timing information since the last overflow occurrence, and the information on whether the process is malicious (attack) or a benign application.

3. **LSTM Modeling**
   Once we obtain the labelled overflow-based datasets, they will be applied to a supervised RNN (Recurrent Neural Network) model, which has the goal of learning whether there is an attack or not based on the monitored microarchitectural traces. The chosen approach, more specifically a LSTM (Long-term Short Memory) model, enables the use of the timing information to detect an attack. The architecture for an LSTM model that includes dynamic temporal information will be defined, based on the one proposed by Baytas et al. [BXZ+17]. This will enable the model to use the timing information on the datasets to learn to detect an attack.

4. **Model Training and Evaluation**
   The defined model will be trained and evaluated using the generated datasets. Different scenarios will be tested to evaluate the generalization of the approach, not only among different applications, but mainly among different platforms. Normalization techniques will also be included in the system in order to improve performance.

5. **Model Fine-Tuning**
   Another technique for performance optimization, the fine-tuning approach will be used to enable the model to perform the detection in an unknown platform with smaller overhead.

6. **Results Analysis**
   The results obtained in the previous steps will be analyzed and compared in terms of performance and overhead.

CHAPTER 2

# Theoretical Background

The chapter is organized as it follows. Section 2.1 explains the principle of cache side-channel attacks, section 2.2 describes Hardware Performance Counters, and section 2.3 talks about Machine Learning Techniques employed in this work.

## 2.1 Cache Side-Channel Attacks

### 2.1.1 Cache Memory and Hierarchy

In modern computers, the cache memory is employed between the CPU and main memory (RAM memory) to address the performance gap between them [LM18]. In other words, the cache is used by the CPU to provide quicker access to data without the need to wait for the slower RAM. The cache works by buffering recently used data, so that the overall memory access time is significantly decreased [LM18].

Moreover, according to Lyu et al. [LM18], modern processors have the cache divided in different levels, as a further improvement to the access time. This way, the higher levels, closer to the processor, are smaller, faster, and more expensive, while the lower levels are bigger, but slower. According to Shen et al. [SCZ21], modern processors contain 3 levels of cache:

- L1 caches: per-core instruction and data caches.
- L2 caches: per-core unified caches.
- L3 cache: large cache shared across cores, also called Last-Level Cache (LLC).

Depending on whether the information requested by the CPU is available in the cache or not, we have a cache hit or miss, respectively. During the past decade, the difference in

the access times of cache hits and misses has been exploited by cache-based side-channel attacks [HL17] [LM18].

According to He [HL17], an effective attacker is able to observe all of the victim's memory accesses and infer whether it is a cache hit or miss. Furthermore, by observing the cache shared with the victim, the attacker can also infer the memory address used by the victim. This way of getting the leaked metadata is what characterizes cache timing side-channel attacks, and it is used, for example, to derive the secret key to encryption algorithms [HL17].

However, it is not always the case that the attacker can observe all the times the victim accesses the cache. In this case, the attacker uses indirect observations, which can be done in two different ways, as described by He [HL17]. The first one, called "time-based" cache side-channel attacks (SCA), is where the attacker measures the access time of its own memory access, after interfering with the victim sometimes. The other one, called "access-based" SCA, happens when the attacker looks at the total time of the victim's security-critical operation.

### 2.1.2 Speculative Execution

According to Kocher et al. [KHF+20], modern processors deploy speculative techniques for optimization. This approach is related to the fact that, when the control flow of a program depends on a value stored in memory (and not in the cache), it may take hundreds of clock cycles to retrieve this value, while the processor is idle. Instead, with speculative execution, the processor predicts the direction of the control flow and executes the program speculatively [SCZ21]. If, when the actual result comes, the prediction is following the wrong path, the processor discards the precomputed results and go back to the last correct state. However, if the prediction is accurate, the processor commits to these results and continues to execute. According to [SCZ21], the speculative execution technique improves the performance significantly. However, it can be exploited by attackers to leak information and access sensitive data [AZW24] [GMES19].

### 2.1.3 Attack Techniques

According to Shen et al. [SCZ21], there are various examples of microarchitectural cache side-channel attacks, and they usually involve the combination of different techniques. According to its relevance for this work, we will present two groups of approaches that are often combined to create attacks.

#### Instrumented Attacks

In instrumented attacks, the attacker uses the cache side-channels to interfere with the victim's access patterns and then capture the victim's secret [SCZ21]. The difference in the latency of cache accesses in different situations is leveraged by the attacker in order to infer the victim's behavior [SCZ21].

These type of attacks can be further classified in two groups, as described by [SCZ21]. The first class, "flush-based" , uses "clflush" instructions to flush the memory lines. The most common ones are Prime + Probe, Evict + Time, Flush + Reload, and Flush + Flush.

- Prime + Probe [MAB+18]

  The Prime + Probe attack can be divided in three distinctive parts [LM18]

  1. Prime: Initially, the attacker occupies specific (or all) cache sets with its own data.
  2. The victim executes its own process
  3. Probe: The attacker accesses the same data that it has previously loaded into the cache. If the data loaded by the victim is mapped to the same cache sets and evicts the attacker's data, there will be a cache miss, and consequently, a longer probe time. Otherwise, if the data is still in the cache, the probe time will be smaller.

  Therefore, as Lyu et al. [LM18] explain, the attacker only measures its own running time, which makes it effective and noise-resistant. This attack technique uses the last-level cache (that is shared among many processors), but it can also be implemented at L1-data and L1-instruction [BRN24].

- Evict + Time [OST06]

  According to [LM18], this is a time-driven attack that will use the execution time from the victim to learn information. Lyu [LM18] divides it into 3 stages:

  1. The attacker triggers the victim process
  2. Evict: The attacker fills a specific cache set with its own data, hoping to evict the victim data
  3. Time: The attacker measures again the victim's execution time

  The basic idea of the attack is to first establish a baseline time by preloading the victim's data [SCZ21]. Then, after evicting and running the victim code again, it gets a new execution time. This new time will be longer if the victim accessed the line evicted by the attacker [SCZ21]. On the other hand, the second execution will be faster if the evicted cache lines are not the same. Therefore, the timing of the second execution reveals the memory access pattern of the victim process [LM18].

  According to Lyu et al. [LM18], there are some weaknesses in this technique. Firstly, there are strong assumptions made, such as, for example, the knowledge of the memory address of useful data. Secondly, the time measurement is imprecise in this case [LM18].

- Flush + Reload [YF14]

  Initially defined for the L1 cache and later extended to L3, this attack can determine a specific instruction or data accessed by the victim process [LM18]. The stages of the technique, as defined by Lyu et al. [LM18], are:

  1. Flush: The attacker flushes a memory line (the target address) from the cache.
  2. The attacker waits for the victim process to run, which is given by a pre-configured time [SCZ21].
  3. Reload: The time to reload the memory line is measured. If the victim accesses the line evicted by the attacker, the execution time is longer [SCZ21].

  According to He et al. [HL17], in this attack, the attacker shares a library or data with the victim. If the victim uses the shared data, those shared memory lines will be fetched into the cache. Then, after the reload operation, a hit in the attacker's reloads indicates that the corresponding memory line has been used by the victim [HL17].

- Flush + Flush [GMWM16]

  Created as a variation of Flush + Reload, this attack exploits timing variation in the "flush" instruction itself [SCZ21]. According to Ferracci [Fer19], the technique has only one phase, that is repeated in a loop: the execution of the "flush" instruction on a chosen shared memory line. The attacker measures this execution time, and based on that, he decides whether this memory line has been cached or not [Fer19]. Since the attacker does not load anything from the memory to the cache, the execution time refers only to a load by the victim process. As Ferracci [Fer19] also mentions, the repeated "flush" instruction also evicts the cache lines for the next loop.

### Transient-Execution Attacks

As described by Shen et al. [SCZ21], transient instructions execute unauthorized computations out of the intended code or data paths. Even though their results are never committed to the machine's architectural state, they still leave traces on the micro-architectural state [SCZ21]. Canella et al. [CBS+19] states that this feature has been exploited by transient execution attacks, which usually follow the flow shown in Figure 2.1.

According to Shen et al. [SCZ21] and Canella et al. [CBS+19], the general transient execution attack can be divided in 5 phases:

1. The attacker first prepares the micro-architecture and brings it to the desired state.
2. The attacker executes a trigger instruction, which is any instruction that will cause the following operations to be eventually squashed (as a mispredicted branch or data dependency, for example).

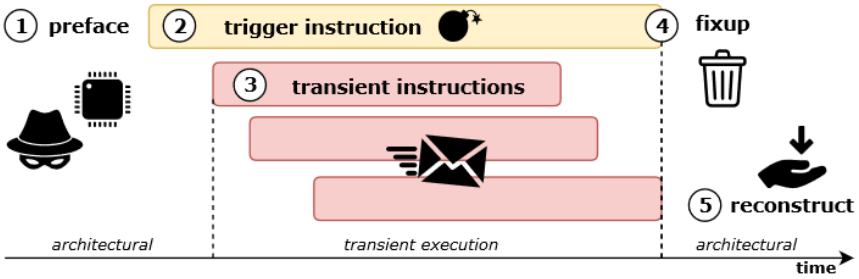



Figure 2.1: High-level overview of a transient execution attack [CBS+19]

3. Before committing the trigger instruction, the CPU continues to execute a sequence of transient instructions, which encode unauthorized data through a micro-architectural covert channel.

4. When it is retiring the trigger instruction, the CPU discovers the exception or misprediction and flushes the pipeline. This is done to discard any architectural effects of the transient instructions.

5. Finally, the attacker recovers the unauthorized transient computation results at the receiving end of the covert channel, and reconstructs the secret from the micro-architectural state.

Depending on what causes the trigger instruction, transient execution attacks can be classified into two groups [SCZ21]:

- Spectre-type attacks: they exploit transient execution following control or data flow misprediction.
- Meltdown-type attacks: they usually exploit transient execution attacks following a faulting instruction.

Being one of the main focuses of this work, the Spectre attack will be better described in the following section.

### 2.1.4 Spectre Attack

A general description of Spectre Attacks is that they trick a victim into transiently diverting from its intended execution path, which is done particularly by poisoning the processor's branch prediction system [SCZ21]. They guide the transient execution from the victim to some code snippets, which use the micro-architectural state to expose secrets. A key factor for this type of attack is that the execution is entirely in the victim's domain, and it can only leak data that is architecturally accessible [SCZ21].

Currently, there are several variants of Spectre attack, each one exploiting a different system component. Canella et al. [CBS+19] proposes a classification in two levels: firstly, according to the micro-architectural buffers that can trigger a prediction, and secondly, to the mistraining strategies that can be used to steer it.

**Spectre variant 1**

The first variant of Spectre was proposed by Kocher et al. [KHF+20]. This approach poisons the Pattern History Table (PHT) to mispredict the direction of conditional branches (whether they are taken or not) [CBS+19]. In other words, as described by Ferracci [Fer19], the attacker mistrains the CPU branch predictor to mispredict the direction of a branch. This causes the processor to execute code that would not have been carried out otherwise.

Al-zubi [AZW24] describes the steps followed by Spectre attack:

1. Initially, the attacker tries to trick the CPU into making incorrect branch predictions, by continuously forcing the victim to engage in misleading behaviors.
2. Then, the attacker uses "flush" instructions to clear the cache, to force the victim process to get any data from the main memory (and not from the cache).
3. On the next step, the attacker tricks the victim process into executing speculative execution on sensitive data (the secret that the attacker is looking for).
4. The victim has to retrieve the secret from the main memory, since the cache was emptied by the attacker.
5. When the processor realizes the misprediction, it discards the results of the speculative computation. But the side effects on the cache are still visible to the attacker.
6. The attacker uses a high-precision timer to determine which values were read from the memory, and then, decode the secret. That is possible because a cache miss implies the victim has to read data from memory, and therefore takes longer, while a cache hit implies reading data from the cache, which is faster.
7. Finally, the attacker retrieves from the cache the secret value.

The implementation of variant 1 of the Spectre attack is available at [Spe18]. The example code for x86 architecture is provided by the authors of the original paper [KHF+20]. The specific idea is that the attacker uses bound checking to mistrain the victim process in the speculative execution of conditional branches. The code snippet on the figure 2.2 below is one example that can be exploited by this attack.

Kocher et al. [KHF+20] describe the execution of this example by the Spectre Attack. In this example, the variable $x$ contains data that is controlled by the attacker. The

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

Figure 2.2: Code snippet for bound checking in Spectre Attack [KHF+20]

conditional statement in this code needs to verify if the value of $x$ is within a legal range. This is necessary to verify if the memory access to $array1$ is valid. According to Kocher et al. [KHF+20], during the phase 1 described above (mistraining phase), the attacker invokes the code while providing valid inputs, training the branch predictor to expect the conditional branch result to be true. On the next phase (exploit phase), the attacker provides a value of $x$ that is out-of-bounds of $array1$. Instead of waiting for the calculation of the branch result (whether the value is valid or not), the processor employs speculative execution and guesses that the bound check will be true, executing the next instruction. Eventually, when the calculation of the bound check is completed, the CPU realizes the misprediction and reverts the changes made to the microarchitectural state. However, the cache state remains visible to the attacker, who can retrieve the secret byte.

According to Shen et al. [SCZ21], this variant of the attack takes advantage of the access to out-of-bounds memory that is done speculatively before the bound check resolves. Moreover, the Spectre attack combines speculative execution, branch prediction, and cache-based side-channels [AZW24].

### Other Spectre Variants

Since the proposal of variant 1 from Spectre attack, there have been developed other works that exploit this attack with different machine components.

- Spectre Variant 2 [KHF+20]

  The Variant 2 of Spectre, proposed by the same authors, is also called Spectre-BTB (Branch Target Injection) [CBS+19]. This approach exploits indirect branches and poisons the Branch Target Buffer (BTB). The main idea here is that the attacker chooses a "gadget" from the victim's address space and influences the victim to speculatively execute the "gadget". Instead of relying on a vulnerability in the victim code, the attacker trains the BTB to mispredict a branch from an indirect branch instruction to the address of the gadget, which consists of a speculative execution of the gadget. In a similar manner as before, the processor reverts the effects of the incorrect speculative execution, but not the effects on the cache, therefore allowing a leakage of sensitive information via a cache side-channel.

- Spectre-RSB (Return Address Injection) [MR18] [KKSAG18]

  According to Ferracci [Fer19], the third variant of Spectre is slightly different from the other ones because it uses the Return Stack Buffer (RSB). The main idea is that the return address value in the RSB is different from the one in the software stack, which leads the program to misspeculate to the address in the RSB. By

triggering this misspeculation intentionally, an attacker can force a process to execute arbitrary code [Fer19].

- Spectre-STL (Speculative Store Bypass) [Hor18]

  According to Canella et al. [CBS+19], speculation on modern CPUs also includes predicting dependencies in the data flow. The variant 4 from Spectre shows how the mispredictions by the memory disambiguator could be abused to speculatively bypass store instructions [SCZ21]. It is based on Store To Load (STL) dependencies, which require that a "load" instruction to the memory shall not be executed before all the previous "store" instructions that write to the same location have been completed [CBS+19]. The attack takes advantage of a performance feature that allows loads to speculatively execute even if the address of a preceding potentially overlapping store is unknown [SCZ21].

### 2.1.5 Other Attacks

In this section, there will be discussed other variations and versions of related attacks that have been developed in the last few years. Since this work is focused on a comparison analysis of Intel and ARM architectures, we could only include in the experiments attacks that work on both platforms. Nevertheless, we will also describe other relevant attacks.

#### Meltdown

Developed by Lipp et al. [LSG+20] in 2018, the Meltdown attack is one of the most famous in the group of transient-execution attacks. This attack targets data that is architecturally inaccessible by exploiting illegal data flow from faulting or assisted instructions [SCZ21]. As described by Canella et al. [CBS+19], in other words, Meltdown-type attacks exploits that exceptions are only raised upon the retirement of the faulting instruction. This allows transient instructions ahead in the pipeline to compute on unauthorized results of the instruction about to suffer a fault. So, even though the CPU discards any architectural effects of this computation, secrets may be leaked through microarchitectural covert channels [CBS+19].

#### Rowhammer

The Rowhammer attack was proposed by Gruss et al. [GMM16] in 2016. It is based on the fact that DRAM cells have the possibility to leak charge over time. By accessing neighboring rows repeatedly, Rowhammer triggers the leak, which leads to bit flips and enables adversaries with low access rights to gain system privileges [GMES19]. In other words, the attack targets DRAM modules by repeatedly accessing a memory location, in order to cause voltage fluctuations and discharge nearby memory locations [BYL21].

Since this attack was adapted for the ARM architecture (Raspberry Pi platform) by Bekele et al. [BYL21], we also included it in the experiments conducted in this work.

#### Foreshadow

Proposed in 2018 by Bulck et al. [BMW+18], the Foreshadow attack targets Intel SGX technology, more specifically enclaves, a private region of memory defined at the user-level or operating system code [Fer19]. The technique extracts a single byte from an SGX enclave, and can be divided in three distinct phases, as explained by Ferracci [Fer19]:

1. Plain text enclave data is cached.
2. The attacker dereferences the enclave secret and loads a secret-dependent oracle buffer entry into the cache, speculatively executing the transient instruction sequence.
3. The attacker executes the same as the receiving end of Flush + Reload technique, and reloads the oracle buffer slots to establish the secret byte.

## 2.2 Hardware Performance Counters

### 2.2.1 Micro-architectural Traces and HPCs

According to Gregg [Gre14], Hardware Performance Counters (HPCs) are special-purpose registers included in modern processors that are used to track diverse processor events, such as clock cycles, instructions, branch misses, and cache hits. They are responsible for storing low-level hardware-related events in the CPU, which are tracked as counters and available in these registers [GMES19]. In other words, the purpose of HPCs is to monitor a hardware event, or to gather information on this event [Fer19].

The counters are used to collect information about the system behavior while an application is running, and are available for various performance events in all major architectures, as Intel and ARM, for example [GMES19]. According to Zhang et al. [ZZL16], most modern processors provide a Performance Monitor Unit (PMU), which enables applications to control HPCs.

Originally designed for software debugging and system performance tuning, HPCs have been recently exploited to detect security breaches and vulnerabilities [ZZL16]. As Zhang et al. [ZZL16] explain, the intuition behind is that HPCs can reveal characteristics of the programs' execution, which in turn can reflect the programs' security states. In addition, these counters offer advantages such as a minimal effect on speed and resilience to modification by an attacker [AZW24], as well as the fact that they introduce negligible performance overhead [ZZL16].

### 2.2.2 Relevant HPCs

Although there are several different events in each CPU model, only a limited number of events can be monitored concurrently in the system because the number of HPCs available in a processor is limited, according to Kim et al. [KHK+24]. This means that

it is necessary to have a proper selection of events to monitor the effective utilization of HPCs.

In addition, as stated by Ferracci [Fer19], the events that can be monitored using HPCs depend on the available processor architectural family. That means that only certain events are available for monitoring on each architecture. That happens because the generation of a hardware event is physically triggered by data paths or control signals implemented in the actual control unit of the CPU, which is often subject to partial or complete re-implementation across different families of processing units [Fer19].

Therefore, for this work, we chose the events monitored by HPCs that are available in the architectures studied, and are relevant for the task of attack detection. In order to choose them, we used the first four HPC events in the ranking performed by Al-Zubi [AZW24]. The choice was motivated by a common goal with this work, the availability if the events in both architectures being monitored, and the possible number of HPCs to be monitored concurrently. The four selected HPCs are:

- Number of retired instructions
- Number of mispredicted branch instructions.
- Number of cache misses for the Level-1 Data Cache.
- Total number of CPU cycles.

The complete list of events is available at the "perf" tool documentation [Perb]:

### 2.2.3 HPC Monitoring

As explained in the previous section, and also confirmed by Moore [Moo02], performance monitoring hardware usually consists of a set of registers that record data about the processor's function, often accompanied by a set of control registers that allow the user to configure and control the performance monitoring hardware. In addition, many platforms, such as Intel and ARM, also provide hardware and operating system support for generating an interrupt to the performance monitoring software when a counter overflows a specific threshold [Moo02].

Therefore, hardware performance monitors can be used in one of the two modes, both with their own uses in performance analysis:

1. Sampling: mode to collect aggregate counts of event occurrences at a pre-defined time interval.
2. Overflow: mode to collect profiling data based on counter overflows.

The modes will be further explained in the following sections.

### Sampling

The first mode is the simplest one, where, for each event being monitored, a register is associated that stores the aggregate counts of the event. At a time interval defined by the user, this register (counter) is read by the system, and the value in the register is sampled. Depending on how the user defined it, the counter can either be reset at each time it is read, or the event count number can be accumulated.

This monitoring mode has a wide range of different applications in performance analysis, for example to identify performance bottlenecks or to relate performance problems to program locations [Moo02].

Moreover, this approach for measuring HPCs has been widely used in the detection of cache side-channel attacks [AZW24] [GMES19] [TZW+20] [KHK+24]. It brings the advantage that is simple to implement and safe, given the constant measurements. However, on the other side, the sampling interval is fixed, and, once defined, it cannot be changed.

### Overflow

Another approach for monitoring HPCs is based on the "overflow" feature mentioned by Moore [Moo02]. In this technique, the user defines, for a given event, a threshold value. Then, once the counting is initiated, the system keeps constantly checking the current count value for this given HPC and comparing it to the threshold. Once the value is equal or greater than the threshold, it generates an interrupt and follows a routine defined by the user. Then, the counter is reset.

In a similar way to the sampling mode, the overflow approach has numerous applications in system performance analysis. It has the advantage of not having a fixed sampling interval, which brings more flexibility to the monitoring. However, it is more complex to implement, and the subsequent analysis may be more difficult to perform.

## 2.3 Machine Learning Techniques

As described by Zhou et al. [Zho21], Deep Learning (DL) is one of the Machine Learning (ML) methods that implements Artificial Neural Networks (ANN). A deep learning network is a neural network with multiple layers. Examples of deep learning networks include Deep Neural Networks (DNN), Convolutional Neural Networks (CNN), and Recurrent Neural Networks (RNN), among others [Zho21].

Due to its ability of learning patterns and detect anomalies, Machine Learning (ML) and, more specifically, DL, have a wide range of applications in security, including intrusion detection, malware analysis, or traffic anomalies [AZW24]. Pang et al. [PSCH21], for example, performs a review on how Deep Learning (DL) can be used for anomaly detection.

In this work, we will focus on the use of RNNs, and more specifically, LSTM networks, which will be explained in the following sections.

### 2.3.1 RNNs

Recurrent Neural Networks (RNN) were first proposed and explained in 1990 by Elman [Elm90]. It consists of a type of Artificial Neural Network (ANN), which can be described as a neural network where the connections between the nodes resemble the neurons of a human brain.

As explained by Muhuri et al. [MCY+20], the principle of an ANN is based on the biological brain, where synapses are transmitted among neurons. In a neural network, signals are transmitted through connections from one node to another. Upon receiving a signal, the artificial neuron processes it and then transmits it to the connected nodes. In an ANN, both neurons and connections usually have weights to adjust the learning process. As they differ, the weights adjust the strength of the signal as it travels from the input to the output layer, across the hidden layers [MCY+20].

According to Muhuri et al. [MCY+20], an RNN should have at least three layers. Figure 2.3 below represents a simple RNN architecture with two hidden layers. The basic components of an RNN include input units, hidden units (which perform calculations to adjust the weights and produce the outputs), and output units [MCY+20].

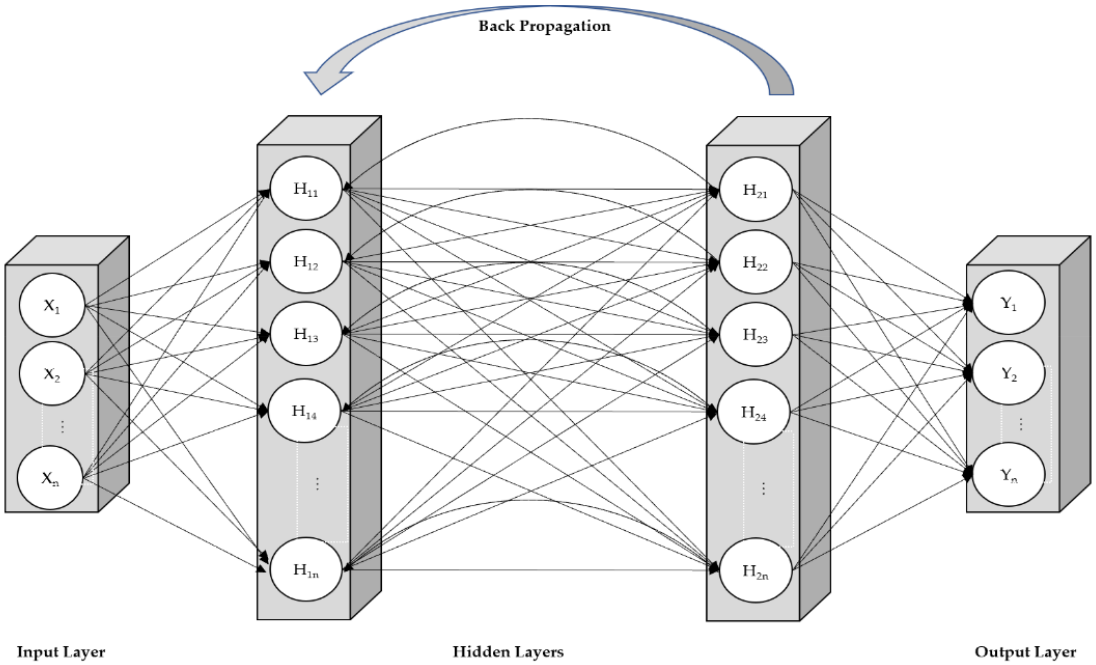



Figure 2.3: A simple RNN [MCY+20]

As portrayed in the Figure 2.3, the one-way information flow in an RNN model flows

from the input units to the hidden units, and a directional loop compares the error of the current hidden layer to that of the previous one, and based on that, adjusts the weights between the hidden layers.

According to Gulmezoglu et al. [GMES19], in a typical RNN structure, the information cycles through a loop. That means that, when the model needs to make a decision, it uses the current input $x_t$ and the hidden state $h_{t-1}$ (where the learned features from the previous data samples are kept). In other words, a RNN algorithm produces output on the previous data samples, and the output is provided as feedback into the network [GMES19]. Figure 2.4 shows a simple diagram of a RNN cell.

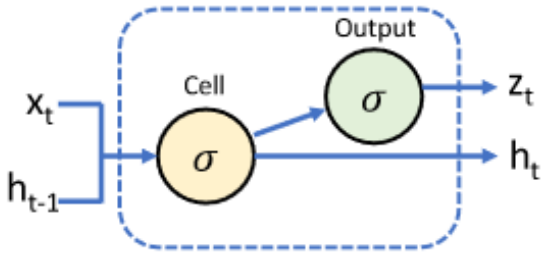



Figure 2.4: RNN Cell [GMES19]

Recurrent Neural Networks can be described as the first algorithm to remember the temporal relations in the input through its internal memory. Therefore, they are widely used for tasks where sequential data is involved [GMES19]. However, conventional RNNs are not effective to learn long-term sequences, because with the increasing time steps, the amount of extracted information converges to zero and the model stops learning [GMES19] [MCY+20].

### 2.3.2 LSTM

Long Short-Term Memory (LSTM) models were first proposed by Hochreiter and Schmidhuber [HS97a] in 1997. They consist of modified RNN models, created in order to address the problem of the vanishing and exploding gradient in traditional RNNs [AP22]. Essentially, they extend the internal memory to learn longer time sequences, being able to bridge more than 1000 discrete time steps [GMES19] [MCY+20].

According to [MCY+20], LSTM networks replace all units in the hidden layer with memory blocks, and each of them has at least one memory cell. Each basic cell on a LSTM network consists of a memory cell, input, forget, and output gates, as represented by Fu et al. [FLM+18] in the Figure 2.5 below.

The basic principle of a LSTM cell is that the information flow is controlled by the three gates (input, forget, and output gates). The memory cells activate with the regulating gates, which control the incoming and outgoing information flow [MCY+20]. At each time step $t$, an LSTM cell receives the current input vector $x_t$, the previous hidden state $h_{t-1}$, and the previous cell state $c_{t-1}$ [Sch15]. The memory cell keeps the learned

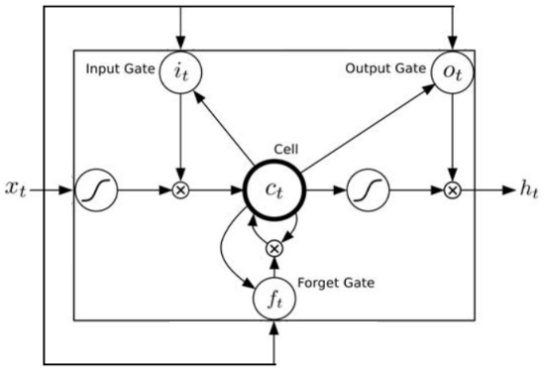



Figure 2.5: LSTM Cell [FLM+18]

information from the previous sequences. Connected to the memory cell, it is placed a forget gate (which are simple sigmoid threshold units), that can reset the state of the unit if the stored information is no longer needed [MCY+20]. The LSTM cell produces two outputs: the current hidden state ($h_t$), and the current cell state ($c_t$) [Sch15]. Finally, the output gate determines whether to pass the output of the memory cell to the next hidden state [FLM+18].

An LSTM layer consists of multiple LSTM cells working in parallel. In practice, multiple LSTM layers are often stacked by feeding the hidden state output of one layer into the next, which allows the network to model more complex temporal patterns [Sch15]. Therefore, LSTM networks can select distinct features in the time sequence data more efficiently than RNNs, which enables it to learn the long-term temporal relations in the input [GMES19].

### 2.3.3 Other Machine Learning Techniques

In a general definition, Machine Learning (ML) techniques are able to improve their performance on a specific task with increasing experience [HTF09]. Therefore, ML techniques have various applications that involve learning a pattern or identifying anomalies. For this reason, traditional Machine Learning (ML) techniques have been widely used for attack detection and other security applications.

The most common ML techniques include linear, non-linear, and non-parametric algorithms (Logistic Regression (LR), Support Vector Classifier (SVC), Perceptron, Decision Tree, Linear Discriminant Analysis LDA), K-Nearest Neighbors (kNN)), probabilistic models (Gaussian Naive Bayes (GaussianNB)), ensemble learning methods (Random Forest (RF) and Gradient Boosting Classifier (GBC)), as well as artificial neural networks (Multilayer Perceptron (MLP)). These algorithms are explained in details by Alpaydin [Alp20].

CHAPTER 3

# Related Works

Due to the importance of assuring information security and privacy, especially with sensitive data, and the developments and increasing variations of cache side-channel attacks, there has been developed works on the detection of these threats. The most recent and better performing approaches with this objective use machine learning or deep learning techniques and can detect attacks with an accuracy of up to 99%.

## 3.1 HPCs Monitoring for Cache Side-Channel Attack Detection

Since the initial implementations of cache side-channel attacks, there has been developed research works that look for ways to detect and identify these types of threats. Given the nature and general behavior of cache side-channel attacks, as explained in Section 2.1, one way to perform this task is by monitoring Hardware Performance Counters.

Al-Zubi and Weissenbacher [AZW24] perform a systematic evaluation of different machine learning techniques to detect Spectre Attack through the statistical profiling of microarchitectural traces. Initially, in order to determine the minimum number of features required for Spectre attack detection, the work samples up to 40 Hardware Performance Counters under different attack scenarios and uses statistical methods to rank them. The selection of HPCs (as well as the optimal number to be monitored) is performed using the Maximum Relevance Minimum Redundancy (mRMR) technique. The best-ranked HPCs that resulted from this experiment reflect execution (total instructions, total cycles) and branch behavior (conditional branches, branch instructions, correctly predicted branches, and branches taken). The datasets for the experiments of this work are generated by sampling the selected HPCs in scenarios, including cache side-channel attacks and benign applications.

In addition, the authors also identify the best-performing machine learning classifiers for this task. They use cross-validation to compare various ML models, including linear, non-linear, and non-parametric algorithms (Logistic Regression (LR), Support Vector Classifier (SVC), Perceptron, Decision Tree, Linear Discriminant Analysis (LDA), K-Nearest Neighbors (kNN)), probabilistic models (Gaussian Naive Bayes (GaussianNB)), ensemble learning methods (Random Forest Classifier (RF) and Gradient Boosting Classifier (GBC)), as well as artificial neural networks (Multilayer Perceptron Classifier (MLP)). The performance of the different classifiers on the same dataset is compared, as well as how the same algorithm performed over all datasets. Finally, a statistical analysis of the features shows that, as a result, the work finds that ensemble learning and Decision Trees outperform other machine learning methods for the detection.

As a result of development on both the attack and Artificial Intelligence techniques, various recent studies use hybrid techniques to improve the task of cache side-channel attacks detection. Tong et al. [TZW+20] developed a method based on the AES algorithm to use Hardware Performance Counters and detect different types of attacks, namely Flush + Reload, Prime + Probe, and Flush + Flush. The dataset used in the experiments is collected through the sampling of 24 HPCs, under 3 different loads, in 6 diverse scenarios (attack and benign applications).

The proposed approach first uses the random forest algorithm to filter the cache features, and select the 4 HPCs that yield the best results (DTLB READ, L1D ACCESS, DTLB ACCESS, and CACHE NODE). Then, the next step is to use Support Vector Machine (SVM) for classification to perform attack detection under different loads. The classification accuracy, precision, recall, and F1 score are measured for the different situations (attacks and loads). The system achieves a detection accuracy of 99.92% without any load, and 96.57% with full load.

A recent type of approach for this task involves the use of Recurrent Neural Networks, most specifically of Long-Term Short Memory (LSTM) algorithms [HS97b]. This technique is particularly suitable for sequential time series, due to its ability of learning both short and long-term dependencies.

The system proposed by Gülmezoglu et al. [GMES19] implements a generic model to detect unknown attacks in an unsupervised way, using HPCs. Initially, a training dataset is generated through sampling of HPCs when only benign applications are running. This data is fed to an LSTM model that learns this behavior pattern, using a sliding window technique to learn using smaller sequences. In the testing (online) phase, the data is collected from the same HPCs under both benign applications and attacks, while it dynamically predicts the next value. The Mean Squared Error between the predicted and real values is calculated, and it is used to detect malicious situations. Within a defined decision window, if all the calculated errors are bigger than a fixed threshold, an anomaly flag is set. Therefore, due to the unsupervised learning technique, the system is capable of detecting attacks even in situations where there is an unknown attack. The reported results show that the trained model is capable of detecting attacks such as

Spectre, Meltdown, Zombieload, and Rowhammer, even without observing them during the training, with the highest F-score of 99,7%.

Most of the approaches that aim to detect cache side-channel attacks observe mainly the cache-related events. That is effective for the earlier variants of the attacks. However, in order to bypass this detection technique, recently proposed attacks use different approaches, as is the case with Prime + Abort, for example. Kim et al. [KHK+24] developed a system that can not only detect the conventional cache side-channel attacks, but also the new variants that exploit other hardware events.

The study focuses on the attacks FLUSH + RELOAD, PRIME + PROBE, and PRIME + ABORT, as well as the L3 cache. Initially, the work analyzes the attacks to identify which hardware events most closely relate to them, besides cache events. The patterns of these events is studied under six different conditions (which involve different attacks, different workloads, and different cryptographic applications), and they are selected based on their distribution pattern and temporal behavior. The five selected events are: L3 cache misses, unhalted cycles, retired instructions, and two Intel TSX events. Once the system collects the selected hardware events, the generated datasets are used to train the models. The authors employ Multi-Layer Perceptron, RNN, and LSTM deep-learning models to infer the presence of attacks, achieving an accuracy of over 99% with LSTM.

## 3.2 Deep Learning for Security

Due to their flexibility and ability to identify patterns, RNNs and LSTMs are used in a variety of different applications. As explained in Section 2.3.2, LSTMs are modified RNNs that extend the internal memory to learn longer time sequences [GMES19]. For this reason, they are frequently used with problems involving sequential data and time series, as is often the case for attack detection. Several studies describe applications of deep learning techniques to mitigate security problems.

This is shown in the work developed by Yao et al. [YJDO19]. In this research, the authors propose a system that uses LSTM to detect privacy risks on Named Data Networks, more specifically, timing attacks. This type of attack uses the timing relation between the delivery of data that is cached and not cached to infer if certain contents were recently requested by the user. In order to detect this type of attack, the system observes several statistics of the network packets: requested content names, the interface from which each interest came, the arrival time of each interest packet, and the corresponding cache hit in each time window. This information is used to generate the input features from the supervised LSTM model: cache hit ratio, average request interval, request frequency, and types of requested contents. The performance is compared in terms of classification accuracy, detection ratio, false alarm ratio, and F-measure, achieving better results than previous works.

Another work that uses deep learning methods to detect network attacks is the one proposed by Liu et al. [LLLY19]. Due to the fact that using traditional machine learning

methods for attack detection in payloads depends heavily on feature engineering (which can be time-consuming and complex to implement), the authors propose an end-to-end detection system. They propose two techniques, one using CNN payload classification, and another one with RNN payload classification. Both have the same goal of detecting network attacks, like DOS, probe, U2R, and R2.

The first implemented technique, referred to by the authors as PL-CNN, exploits the ability of CNNs to extract local region features, in order to consider the entire data stream (and not single bytes). Since network traffic is made up of packets, in which payloads appear in the form of data streams, the authors propose another payload classification model based on an RNN. Since RNNs can learn feature representations from data by storing previous states, it is used to identify specific sequences that can distinguish normal from anomalous communications. The experiments resulted in the best accuracy of 99.36% using PL-CNN, and 99.98% using PL-RNN on public datasets.

Similarly, Fu et al. [FLM+18] also proposes the use of RNNs for network attack detection. The technique proposed in this work is based on Long-Short Term Memory (LSTM), and an end-to-end detection is developed, including data preprocessing, feature abstraction, training, and detection. The system focuses on detecting network attacks based on payloads, and the experiments are conducted on the NSL-KDD dataset. The obtained results show that the proposed method outperforms several attack detection techniques based on feature detection and Bayesian or SVM classifiers.

Recently, hybrid approaches have also been used to classify network behavior and identify network attacks, which shows the relevance of this threat. Muhuri et al. [MCY+20] develops an intrusion detection system that combines the use of a genetic algorithm for optimal feature selection and LSTM with a Recurrent Neural Network for identifying attacks. The work also aims for network attacks, and it is trained and tested using the public NSL-KDD dataset. After the data is initially preprocessed, the features are selected using a genetic algorithm (GA), which selects a subset of 99 features from the original 122. Then, the data is input to the LSTM-RNN model, which is trained and subsequently evaluated.

The system performance was measured with accuracy, recall, precision, F-score, and confusion matrix, with both binary and multi-class classification. The results show that the use of the genetic algorithm increases the classification accuracy in both cases. A comparison with other techniques, such as support vector machine and random forest, also shows that the proposed system outperforms them.

## 3.3 Cache Side-Channel Attacks Detection on Different Environments

Despite the development of attack detection systems, which aim to improve performance and cover multiple variants of cache side-channel attacks, it is not always the case that the threats target conventional processors. Improvements on the attacks also include the

adaptation to different environments, which leads to the need of detection systems that focus on these specific situations.

One of the first works to focus on attack detection, by Zhang et al. [ZZL16], proposes a system to detect cache-based side-channel attacks in multi-tenant cloud systems. The approach is composed by two parts: it uses signature-based detection to identify when the machine is executing a cryptographic application, and at the same time uses anomaly-based detection techniques to identify abnormal cache behaviors that are typical from attacks. This technique brings the innovation of focusing on the root causes of cache side-channel attacks, making it difficult to evade, in addition to being able to detect the attack in real-time. Furthermore, it does not require any new hardware support or modifications. The work achieves high detection accuracy, with a performance overhead of at most 5%.

A more recent work, proposed by Bhade et al. [BPSS24], implements a detection system for cache side-channel attacks that is hardware-based. The approach successfully detects multiple cache timing attacks on multiple sensitive locations simultaneously, with minimal performance overhead. The system is evaluated by synthesizing the entire detection algorithm in a block, which is tested in a RISC-V processor, under different workload conditions. The work achieves more than 98% detection accuracy with an overhead of 0.9 to 2.1%, without any impact on its maximum operating frequency.

Even though most studies on cache side-channel attacks (both on generation and detection) have been performed in Intel processors, ARM CPUs are becoming more popular and are being used in more systems. Since ARM processors have a different cache organization and instruction set than Intel processors, the original versions of cache side-channel attacks do not work in this environment. However, there have been developed works that adapt the attacks to ARM CPUs, as is the case of Lipp et al. [LGS+16].

The work initially identifies the challenges for implementing cache side-channel attacks in ARM processors: the fact that the last-level caches are not inclusive, usually there are multiple CPUs that do not share a cache, ARM processors do not support a flush instruction, they use a pseudo-random replacement policy, and cycle-accurate timings require root access. Then, the authors adapt the attacks and demonstrate the applicability of the cache attacks on ARM environments. The adaptations work irrespective of the actual cache organization, and their functionality is demonstrated on Android smartphones.

# CHAPTER 4

# Methodology

## 4.1 System Overview

As described in Chapter 1, the implemented system has the aim of identifying cache side-channel attacks by monitoring HPCs, using an overflow-based approach. In order to evaluate whether the learning could be transferred among different architectures, the experiments were conducted in two different architectures. Figure 4.1 shows the general overview of the system flow.

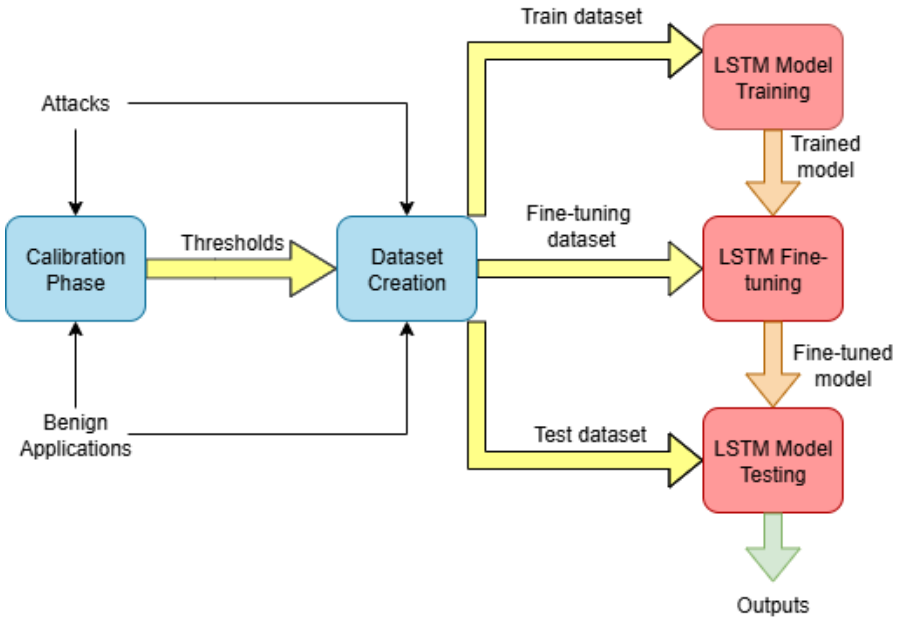



Figure 4.1: Proposed System Overview

Initially, the datasets are created using the overflow technique while monitoring the HPCs running on the machine. In other words, the system is monitoring the values of the selected Hardware Performance Counters and, when the value of either of them exceeds a defined threshold, a sample of the dataset is collected. Therefore, it is first necessary to determine the threshold values for each HPC. This is done in the Calibration Phase, where the system is constantly monitoring the HPC values while benign applications are running. At the same time, it keeps adjusting the values of the thresholds until the overflow events are sampled at around a defined frequency. Once the thresholds are defined, the training dataset itself is created using these fixed values by monitoring the overflows, while attacks and benign applications are also running on the computer. In the next step, after their creation, the datasets are used for training and evaluating the deep learning model. The defined LSTM model was initially trained using the training and validation datasets. Then, in order to assure that the learned knowledge is transferred among different architectures, the model goes through a fine-tuning phase. And finally, the resulting model is evaluated on the test dataset. With the aim of guaranteeing the robustness of the model, the training and evaluation were performed on various scenarios, including different dataset combinations from both platforms.

## 4.2 Data Selection

Prior to the generation of the datasets to train and evaluate the system, it is necessary to define how to create then. That includes defining which HPCs will be monitored to better identify the attacks, ensuring that they are available for monitoring on the two chosen platforms. In addition, both the attacks and the benign applications need to be determined according to their relevance to the attack detection task and the availability on the platforms.

### 4.2.1 HPCs Definition

As mentioned in Section 2.2.2, there is a variety of HPCs available on the different computer architectures. However, in addition to the constraint of the availability of the event counter, each platform has a limitation on how many events can be monitored concurrently. In this work, we perform experiments both on Intel and ARM architectures.

Modern Intel CPUs support three fixed and four programmable counters per core [Int25]. The ARM computer, more specifically a Raspberry Pi in our case, can support up to four HPCs being monitored concurrently [Ras22]. Therefore, in order to guarantee that the datasets created in both platforms can be used interchangeably, the number of HPCs being monitored concurrently needs to be chosen in accordance with the limitations of both of them. In this case, we chose to create the datasets by monitoring four events concurrently.

In the next step, it is necessary to define which HPCs will be monitored for overflows. This has to be done first by observing the list of available events on each platform. The

events available for monitoring in the Intel platform are available in [perc]. For the Raspberry Pi platform (ARM architecture), the list of available events can be found at [perd].

From the list of available HPC events for monitoring, it is necessary to choose the ones that best reflect the behavior of the computer during the cache SCA. In her work, Al-zubi [AZW24] performed an evaluation and ranked the events that are most related to this type of situation. This evaluation was made using the technique of Maximum Relevance Minimum Redundancy (mRMR) [FTS20], which identifies features that are least related to each other and most strongly correlated with the class.

After the described steps, the four HPC events to be monitored concurrently by our system are:

- PERF_COUNT_HW_INSTRUCTIONS = Counts the number of retired instructions.
- PERF_COUNT_HW_BRANCH_MISSES = Counts the number of mispredicted branch instructions.
- PERF_COUNT_HW_CACHE_L1D/RESULT_MISS = Counts the number of cache misses for the Level-1 Data Cache.
- PERF_COUNT_HW_CPU_CYCLES = Counts the total number of CPU cycles.

### 4.2.2 Attacks

Among the list of implemented Cache SCA, as described in Section 2.1, we needed to select the attacks that we aim to detect in this work. Since we are also aiming to perform transfer learning between two different architectures, the attacks involved in the experiments need to have an available implementation that works on both the chosen platforms for our experiments.

Due to its relevance and number of variants, the main attack that we aim to detect in this work was Spectre Attack [KHF+20], explained and described in Section 2.1.4. This attack was extensively studied, both in works that also aim to detect it, as in works that propose different variations for it. The variants chosen to be explored in this work are: Spectre variant 1 and variant 2.

In order to increase the generalization ability of the proposed system, we also included the Rowhammer attack in our experiments [GMM16]. In addition to the most common implementation of this attack, there is also an adapted implementation for the ARM architecture, more specifically, for the Raspberry Pi platform [BYL21].

### 4.2.3 Benign Applications

The aim of this work is to develop a system that is able to detect the presence of cache SCA regardless of the remaining workload running in the computer at the same time.

Therefore, it is necessary to train and evaluate the system in different scenarios and under different workloads, to ensure that the technique is still able to identify it.

In order to cover both situations of light and heavy workloads, the datasets created to train and evaluate the system included applications from the following classes:

- Normal workload: includes internet browsing, file explorer, file compressing, text and spreadsheet editing
- Encoding: includes string, video, and audio encoding
- Reasoning: includes SAT solvers
- Benchmarks: includes computationally heavy benchmarks
- Memory stress tests
- Firewall management

The complete list of benign applications used in the dataset creation is shown in Section 5.2.3.

## 4.3 Dataset Creation

After the selection of the HPC events to be monitored, the attacks and benign applications to be used in the dataset creation, the next step is to effectively create them.

Previous works that use HPC events to detect attacks usually employ the sampling technique to monitor these events [AZW24] [TZW+20] [GMES19] [KHK+24] [BPSS24]. That means that, repeatedly after a pre-defined time interval (for example, 100 ms), the system measures the values of HPCs. Each measurement consists of a sample in the dataset, which is used as an input to a ML model that aims to learn patterns and detect attacks.

However, this technique has some limitations. First, the fact that the time interval is fixed does not allow flexibility on the monitoring. If, on the one hand, the attack is faster than the interval, the system might miss the attack. However, on the other side, if the attack is much slower, the overhead produced will be significantly higher, generating more data than is necessary.

Secondly, the sampling approach results in the model learning the pattern for attacks in a specific platform or architecture. However, different platforms behave differently with respect to HPC values, due to the differences in the architecture, memory, and cache. Therefore, the model learning is not transferred among platforms, which means that a model learned in one platform cannot be used to detect attacks in a different one.

In order to overcome these limitations and generate a model that better performs in the attack detection of different platforms, this work proposes that the datasets be created using an overflow-based approach. As mentioned in Section 4.1, this technique is composed of two parts: the calibration phase and the overflow monitoring.

### 4.3.1 Calibration Phase

The initial step of the dataset creation is to perform the calibration, or, in other words, to determine the thresholds for each HPC to trigger an overflow. This needs to be executed for each platform, taking into account the benign applications.

On this calibration phase, we set initial values for the selected HPCs. In order to select these initial values, we performed the sampling of the event counters every 100 ms, while running the applications. Then, we calculated the average of the sampled values, and subtracted their standard deviation. These calculated results are defined as the initial threshold for each HPC.

Once the initial threshold values are fixed, we run the monitoring process on the machine, simultaneously with the benign applications. The process continuously checks for the HPC counter values, and every time these values reach the threshold, an overflow is triggered, and the counter is reset. At each situation where an overflow occurs, we compare the time since the last overflow with a target time range. In this work, we compared the results from different time intervals, and chose the interval from 20 to 30 ms, so which is smaller than the usual interval in the state-of-the-art (100 ms), but not so small as to create a high overhead.

From this comparison, we adapt the current threshold. If the time between overflows is smaller than the target interval, it means that the threshold is too low and the overflow is being triggered too often. In order to correct that, we increase the current threshold by 10%. On the other hand, if the time between overflows is too big, that means that the threshold is too high, and the overflows are too sparse. In this case, we decrease the current threshold by 10%. However, if the time between overflows is within the target range, we also need to guarantee the robustness of this threshold value. Therefore, we wait for this situation (that the time between overflows is within the target range) to be repeated a certain number of times (in our case, 10 times) before the threshold for this particular HPC is considered calibrated. Only after the thresholds for all four HPCs have been calibrated, we consider the calibration phase to be complete.

Since each platform has different features, like frequency and cache size, for example, the calibrated thresholds will be different for each machine. Consequently, the calibration process needs to be executed at least once for every new device that we use to perform the experiments.

### 4.3.2 Overflow-based Dataset Creation

Once all the thresholds have been defined for each specific platform, the overflow-based datasets must be created, both for training and evaluation, encompassing various scenarios. For this work, in order to also test if the learning is transferred among platforms, multiple datasets were generated for both platforms used in the experiments (Intel and Raspberry Pi).

As described in Section 4.1, the datasets are generated based on overflows while monitoring the selected HPCs. In parallel to this monitoring, both benign and malicious applications are also being executed in the computer. At the beginning of this procedure, we list all the processes currently running on the machine, and attach to each one a monitoring unit. Then, the program keeps constantly checking the HPC values corresponding to each process, and comparing them to the thresholds defined in the calibration phase. Once one of the values reaches the threshold, an overflow is triggered, and a sample of the dataset is collected.

Upon the triggering of the overflow, a handler is called to generate a sample added to the dataset. Each sample is composed of the following parts:

1. Process name: the name of the process being monitored
2. Trigger: identification to the HPC that triggered the overflow
3. "t0": elapsed time since the last overflow triggered by this specific event.
4. "t1": elapsed time since the last overflow.
5. label: label identifying whether the process that generated the overflow is a benign process (label 0) or an attack (label 1).

In order to ensure the diversity of scenarios and guarantee that the trained model is robust, the dataset generation was performed on both platforms, and under different workloads (light, normal, and heavy workload). As explained in the previous section, it involved different attacks (Spectre variant 1 and 2, and Rowhammer), and various benign applications (the list will be detailed in Chapter 5.

## 4.4 LSTM Model Definition

The deep learning technique used to implement the attack detection model is LSTM, explained in Section 2.3.2. This algorithm was chosen because of its ability to store information and to perform well on data series involving time.

### 4.4.1 Dynamic Temporal LSTM

More specifically, the implemented model was based on the one proposed by Baytas et al. [BXZ+17]. This technique is described by the author as "time-aware LSTM network", or T-LSTM, and has the ability to handle irregular time intervals in a series.

The proposed architecture of this model has the same gates as a regular LSTM (forget, input, and output gates), but the memory cell is adapted in a way that, the longer the elapsed time, the smaller the effect of the previous memory to the current output. In other words, previous events that are more recent affect more the network, than if they take a longer time. With this goal, the elapsed time is transformed into a weight in the calculation of the current memory cell value, using a time decay function. A detailed explanation of the model architecture can be found in the paper by the author [BXZ+17].

This adapted technique is effective on the proposed application due to the fact that the generated datasets also present irregular time intervals between samples. The overflow-based approach generates datasets that, in opposition to most related works, are not sampled at a regular time interval. In consequence, it is expected that it will have a good performance in our case.

## 4.5 Model Training and Evaluation

On the next step, the defined T-LSTM model was trained and evaluated on the generated datasets, which went through a pre-processing step before the training. In order to improve the performance and guarantee that a trained model can be applied on different platforms, a fine-tuning step was implemented after the training.

### 4.5.1 Performance Metrics

The training, validation, and testing tasks were evaluated using various ML performance metrics, which will be further explained in Chapter 5. The chosen metrics for performance evaluation are:

- Accuracy
- Precision
- Recall
- F1-Score
- True Negative Rate
- Loss

Moreover, the time overhead was also measured for comparison.

### 4.5.2 Data Pre-processing

With the purpose of ensuring that the predictions by the DL model are more accurate, and that the dataset creation did not generate any accidental errors, all the data goes through a pre-processing phase before being input to the system.

On this step, the collected data is submitted to a normalization procedure, an essential feature scaling technique that maps the defined data to a specific range, in order to provide better results.

On this work, the most important feature provided as input to the T-LSTM model is the timing information. Therefore, this is the feature that we chose to normalize before starting the model training. In order to find the normalization technique that is most appropriate to our situation, we experimented with different approaches and compared their results.

The normalization techniques that we experimented with were: min-max normalization, zero mean (standardization), and quantile normalization. The one that yielded better results, and was therefore chosen, was the zero mean normalization, where the mean is subtracted from the data, and the result is divided by the standard deviation.

An additional step that we included on the training dataset was to group the samples generated by the same process together. The goal of this step is to improve the learning, since the model receives the input data in sequences. With this technique, we can help the model learn better the patterns from the same process on the datasets.

### 4.5.3 Model Training

Once the datasets are clean, normalized, and preprocessed, they are ready to be used as input for the defined T-LSTM model.

The initial step in the model training is to define the training, validation, and testing datasets. With the purpose of testing our hypothesis that the model is able to transfer the learning, we defined different scenarios of combinations of datasets from both platforms (Intel and Raspberry Pi). These scenarios can be classified in 3 different groups:

1. Training, validation, and testing datasets from the same platform.
2. Training, validation, and testing datasets as a combination of both platforms.
3. Cross-validation among platforms: training and validation datasets from one platform and testing dataset from the other one.

After the datasets are determined, the model is defined, and the training starts, with the validation happening every 10 steps. The training is performed for 100 epochs. During the training, the input data is provided to the model in sequences of a defined length,

hence our choice to group samples from the same process. The chosen parameters were manually defined based on the experiments that produced better results.

At each step of the training and validation, the predictions are used to calculate the performance metrics previously cited, and the results are logged. The specific implementation details will be described in Chapter 5.

### 4.5.4 Model Fine-tuning

As previously explained, one hypothesis being evaluated in this work is whether the use of an overflow-based approach and an LSTM model enables the transfer learning between two platforms from two different architectures (Intel and ARM architecture). However, after the performed experiments (whose results will be detailed in Chapter 6), we observed that the behavior from the two platforms was significantly different, which resulted in the transfer learning not working reliably.

For that reason, we included a fine-tuning step after the model training. This step was included as a way of ensuring that a model trained on one platform can be used for detection in another one, but without the need to retrain the whole model.

The fine-tuning procedure has the goal of optimizing a pre-trained ML model for a specific task or dataset, in order to improve its performance. In the case of this work, the fine-tuning will allow a trained model to adapt better to the dataset from the other platform.

### 4.5.5 Model Testing

Finally, after the training and fine-tuning of the model, it can be tested. The evaluation involves the same performance metrics previously defined in Section 4.5.1 (as the training and validation).

The different scenarios for testing are in accordance with what was previously defined in Section 4.5.3, with respect to the different platforms, as well as to the different workloads. The cross-validation is related to the hypothesis that the trained model generalizes well for the different situations, including the data from two different architectures. The results from the performed experiments will be further detailed in Chapter 6.

CHAPTER 5

# System Implementation

The following chapter will describe the technical details of the system implementation. Section 5.1 describes the general details of the machines on which the experiments were developed. Section 5.2 is related to the dataset creation process, and Section 5.3 is focused on the details of training and evaluation of the LSTM model.

## 5.1 Experimental Setup

As described in Chapter 4, the system proposed in this work was implemented in two different platforms: an Intel NUC and a Raspberry Pi, with the aim of evaluating the model's ability to generalize among different architectures.

In order to cover the ARM architecture, it was used a Raspberry Pi 4 Model B [ras]. The processor in this device is a Broadcom BCM2711 SoC, composed of four Cortex-A72 cores compatible with ARM v8 architecture. The device includes an available cache of 320 KB for the L1 cache (128 KB for data and 192 KB for instructions) and 1 MB for the L2 cache. The operating system used is the Raspberry Pi OS (Debian 12 bookworm), which is based on Linux.

On the other hand, related to the Intel architecture, it was used a Intel NUC Board NUC10FNH [nuc24]. This machine is composed of 8 Intel i5-10210U cores with a frequency of 1.5 GHz, compatible with x86_64 architecture. The cache sizes are: 256 KB for the L1 (128 KB for data and 128 KB for instructions), 1 MB for the L2 cache, and 6 MB for the L3 cache. The operating system used is Debian GNU/Linux 11 (bullseye).

For training and test experiments with the T-LSTM model, a Dell Precision laptop with a 12th Gen Intel Core i7-12700H processor was used. The operating system is Ubuntu 20.04.6.

## 5.2 Dataset Creation

The step of dataset creation (which includes both the calibration phase and the overflow-based dataset creation) was implemented using the C language, and the gcc compiler was used to compile it on both platforms. The implementation of the dataset creation is listed on Appendix 7.1.

### 5.2.1 Monitoring Tool

In order to perform the monitoring of the Hardware Performance Counters, it was used the profiling tool Perf Linux [pera]. This lightweight tool was used on multiple previous studies to collect events from HPCs. The monitoring tool is available for Linux operating systems, and enables the overflow-based approach by setting interrupt signals on the selected HPCs. The Perf tool also enables attaching the counters to each running process, which allows local monitoring. Since it is a tool designed for Linux operating systems, it works on both platforms with no need for adaptation.

### 5.2.2 Attacks

The attacks used on the experiments, as explained in Section 4.2.2, are Spectre variants 1 and 2, and Rowhammer.

For the datasets created on the Intel platform, it was used the original implementation of the attacks. The Spectre variant 1 was the one implemented by [Spe18]. The variant 2 from Spectre was the one by [spe]. And the Rowhammer code is available at [rowa].

On the other hand, for the Raspberry Pi platform, we needed to use adapted implementations. That is due to the fact that one of the instructions used in the attacks, particularly the "flush" instruction, is implemented differently on the ARM architecture. Both variants of the Spectre attack were developed by [Jia24]. And the variant of Rowhammer for Raspberry Pi was implemented by [rowb].

### 5.2.3 Benign Applications

The benign applications used for the dataset creation can be classified into different groups, as described in Section 4.2.3. The applications used are the following.

- Normal workload: internet browser, terminal, explorer, text editing, file compression, network connectivity (ping), git
- String encoding
- Image decoding: dcraw
- Video encoding: ffmpeg
- Reasoning: minisat (with different input files)

- Stress test: stress-ng
- Benchmarks: lzbench, mbw (Memory Bandwidth Benchmark), sysbench
- Firewall management: ufw

Since our goal in this work is to generalize among platforms, we chose only applications that are available and work on both the Intel and Raspberry Pi platforms.

## 5.3 Model Training and Evaluation

The module responsible for model training and evaluation was implemented using the Python language, more specifically, the Pytorch library (version 2.4.1). We also used the Pytorch Lightning framework for deep learning-related tasks, and Wandb to plot the performance metrics during training and validation. The implementation for the model training and evaluation is listed on Appendices 7.1 and 7.1.

### 5.3.1 Model Architecture

In order to create the training and validation datasets, they were randomly split using the 80-20 ratio (80% of the dataset for the training and 20% for the validation). For the fine-tuning and testing tasks, different datasets were used. The dataloader used for the training uses a batch size of half of the dataset size.

The proposed supervised T-LSTM model has the following architecture:

1. An embedding layer
2. A set of 4 T-LSTM layers, each one of size 64
3. A set of 4 fully-connected layers, each one of size 64
4. A linear output layer

For training, the Adam optimizer was used. The activation functions for the T-LSTM layer are sigmoid for the gates and hyperbolic tangent for the cell. For the fully-connected layers, a ReLU activation function is used. In addition, the 10-fold cross-validation technique is used to ensure more accurate results.

### 5.3.2 Model Hyperparameters

During the performed experiments, different hyperparameter values were manually evaluated and compared, and the ones that yielded better results were chosen. These hyperparameters are:

- Number of epochs: 100
- Sequence length: 100
- Learning rate: $10^{-3}$
- Initial segment: 50

### 5.3.3 Performance Metrics

During the execution of the experiments, various metrics are calculated to compute the system performance. In order to compute them, we first define the necessary elements in relation to our application:

- True Positives (TP): attack samples correctly classified.
- True Negatives (TN): benign samples correctly classified.
- False Positives (FP): benign samples classified as attacks.
- False Negatives (FN): attacks that were missed and classified as benign samples.

The employed performance metrics are defined as follows.

- Accuracy:

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{5.1}$$

- Precision:

$$Precision = \frac{TP}{TP + FP} \tag{5.2}$$

- Recall:

$$Recall = \frac{TP}{TP + FN} \tag{5.3}$$

- F1-Score:

$$F_1 = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{5.4}$$

- True Negative Rate:

$$TN\_rate = \frac{TN}{TN + FP} \tag{5.5}$$

In addition to the listed metrics, the cross-entropy loss was used for the training, validation, and testing loss. A more detailed explanation of the performance metrics can be found in the book by Zhi-Hua Zhou [Zho21].

CHAPTER 6

# Experimental Evaluation

## 6.1 Calibration Phase

As described in section 5.2, the first implemented step was the calibration phase, which was performed both for the Intel platform and the Raspberry Pi platform. As a result, this phase generated the thresholds for the overflow-based dataset creation.

### 6.1.1 Intel Platform

The first part of the calibration phase includes sampling the HPC values every 100 ms (while the benign applications are also running on the machine) to define the initial thresholds. In our work, these initial values were defined by getting the average of the sampled values, for each HPC. The initial thresholds obtained for the Intel platform were:

- Total instructions (TOT INS): 80000
- Branch mispredictions (BR MSP): 2000
- L1 data cache misses (L1 DCM): 4000
- Total cycles (TOT CYC): 500000

Then, starting from these values, the thresholds were calibrated according to the procedure described in Section 4.3.1. As described previously, three values of target time interval were tested, and the calibration was performed for each one (20, 100, and 200 ms). The time for calibrating in the Intel platform varies between 1 and 2 minutes, depending on the target time interval. The obtained thresholds for each target value are displayed on the table 6.1.

Table 6.1: Thresholds - Intel Platform

| HPC | 20 ms | 100 ms | 200 ms |
|---|---|---|---|
| TOT INS | 2000000 | 8000000 | 20000000 |
| BR MSP | 2000 | 20000 | 200000 |
| L1 DCM | 3000 | 30000 | 300000 |
| TOT CYC | 2000000 | 8000000 | 200000000 |

In order to choose which group of thresholds to use, the datasets created using each one of them were evaluated in the LSTM model, in addition to being analyzed regarding the space and time overhead. Table 6.2 displays the comparison between the datasets created for each target interval, with respect to the mentioned aspects. The first line displays the test accuracy when the respective dataset is used on the model evaluation. The second line shows the timing overhead in the case that the sampling approach was used to generate the dataset (with the related sampling time interval), instead of the overflow approach. Moreover, the space overhead for generating a dataset using the sampling approach, considering the monitoring of the HPC values, is around 10 times more, when compared to the chosen overflow approach (for each generated dataset).

Table 6.2: Comparison Target Intervals - Intel Platform

| Metric | 20 ms | 100 ms | 200 ms |
|---|---|---|---|
| Test Accuracy | 0.98827 | 0.91871 | 0.9774 |
| Sampling Time Overhead | 4.855 | 2.47 | 2.36 |

Given the fact that the compared accuracy values are approximately the same, and the space and time overhead are significantly smaller, the target time interval of 20 ms was chosen to generate the datasets in the Intel platform.

### 6.1.2 Raspberry Pi Platform

Similarly, for the Raspberry Pi platform, the initial thresholds were defined in the same way, by calculating the average on the HPC sampling every 100 ms. The computed initial thresholds for this platform are:

- Total instructions (TOT INS): 200000
- Branch mispredictions (BR MSP): 10000
- L1 data cache misses (L1 DCM): 100000
- Total cycles (TOT CYC): 3000000

The time for calibrating in the Raspberry Pi platform varies between 2 and 3 minutes, depending on the target time interval. The obtained thresholds for each target value are displayed on the table 6.3:

Table 6.3: Thresholds - Raspberry Pi Platform

| HPC | 20 ms | 100 ms | 200 ms |
|---|---|---|---|
| TOT INS | 250000 | 2500000 | 2500000 |
| BR MSP | 35000 | 100000 | 100000 |
| L1 DCM | 40000 | 50000 | 500000 |
| TOT CYC | 1000000 | 10000000 | 20000000 |

Similar to the Intel platform, the Raspberry Pi was also evaluated in the same way regarding the comparison of the target time intervals. Table 6.4 below displays the comparison:

Table 6.4: Comparison Target Intervals - Raspberry Pi Platform

| Metric | 20 ms | 100 ms | 200 ms |
|---|---|---|---|
| Test Accuracy | 0.94474 | 0.91138 | 0.90979 |
| Sampling Time Overhead | 3.636 | 3.076 | 5.38 |

In the same way as to the Intel platform, the target time interval chosen for the dataset creation was 20 ms.

## 6.2 Dataset Creation

The process of dataset creation is detailed on Section 5.2. In order to perform experiments across different platforms, it was necessary to create datasets on both the Intel and Raspberry Pi platforms. The thresholds defined for each target time interval (listed on Section 6.1) were manually experimented with and evaluated, and, for both platforms, it was chosen the set of thresholds related to the target time interval of 20 ms, which generates a smaller overhead with similar performance.

### 6.2.1 Intel Platform

To encompass different scenarios (including benign and malicious applications), we created numerous datasets and sequentially selected those that best represent the combinations of situations. This choice was based on a better proportion of samples from benign processes (which received the label "0") and from attacks (labeled as "1"). The initial goal from this work is to detect Spectre attacks, but we also executed experiments including the Rowhammer attack. Therefore, regarding the attack samples present in the data, the chosen datasets include two different situations:

- Spectre Attack: 7 datasets selected
- Spectre and Rowhammer attacks: 5 datasets selected

The criteria used for the selection are: the number of samples in the dataset, the ratio between benign and malicious samples, and the diversity of the benign applications. The size of the datasets varies between 20000 and 35000 samples, and the ratio that we looked for to use in the experiments is around 50% (meaning that around half of the samples are from attacks, and the other half from benign applications).

#### Data Analysis

The selected datasets were used in different experiments, as it will be detailed in Section 6.3. However, in the interest of conciseness, we will only present here a data analysis from the dataset that yielded the best results in the experiments.

For the Intel platform this dataset included both attacks (Spectre and Rowhammer). It contains 29000 samples, from which 14992 are from attacks and 14008 are from benign applications.

Figure 6.1 displays the distribution of the time since the last overflow ("t0", as defined in section 4.3.2) for each HPC, both for benign and malicious applications. The figure displays the distribution of "t0" after the normalization step, and the values are divided for the situations when each HPC triggered an overflow.

### 6.2.2 Raspberry Pi Platform

The selection of datasets created in the Raspberry Pi platform followed the same principle. The chosen datasets also included two different attack scenarios:

- Spectre attacks: 9 datasets selected
- Spectre and Rowhammer attacks: 5 datasets selected

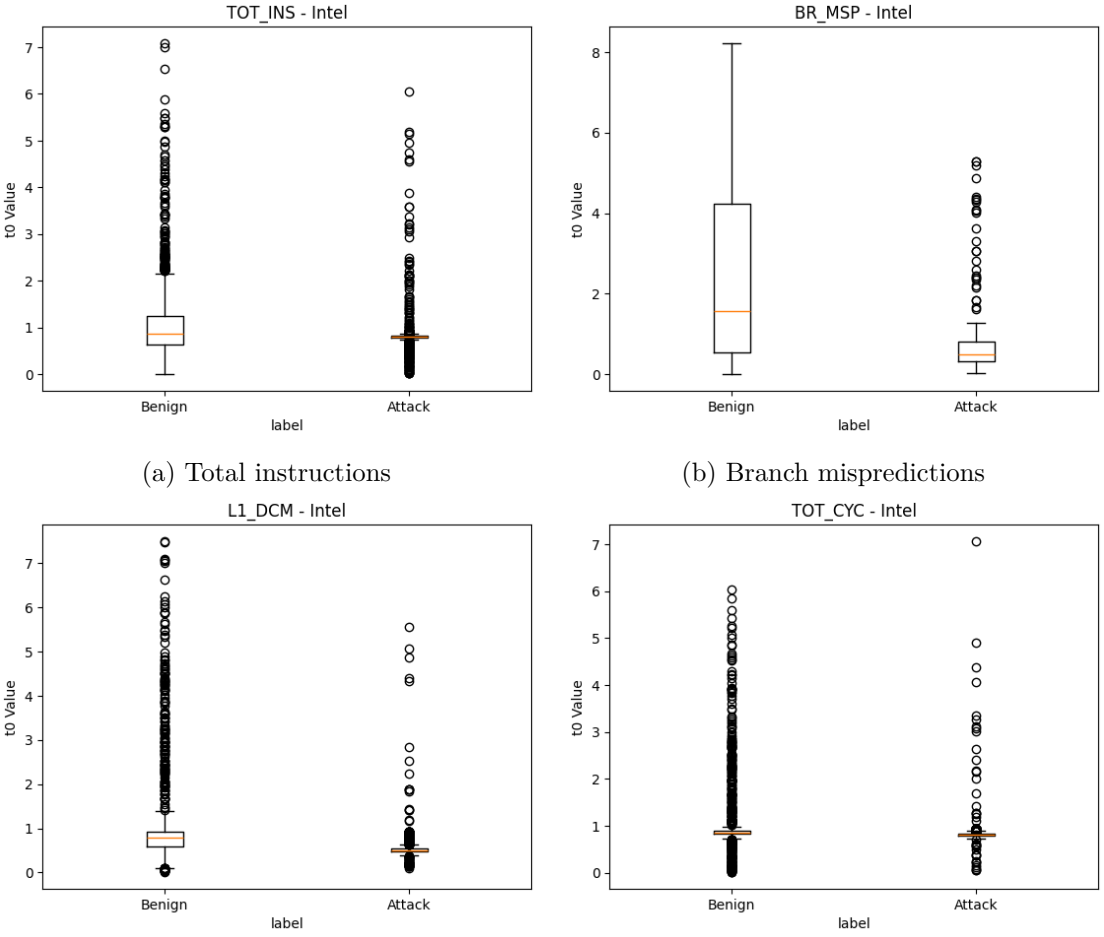
The same criteria were used for the dataset selection (as in the Intel platform). Similarly, the size of the datasets varies between 10000 and 35000 samples, and the aimed ratio between benign and malicious applications remains 50%.

#### Data Analysis

In the Raspberry Pi, the dataset that generated the best results in the experiments is composed of 12000 samples, from which 6601 are attacks (label "1") and 5399 are benign (label "0"). Figure 6.2 displays the distribution of "t0", for each HPC that triggered the overflow, after the normalization step.

## 6.3 LSTM Model Experiments

Once the datasets were generated, we performed the experiments with the implemented T-LSTM model, as described in Section 4.4. Based on the nature of the datasets (comprising




(a) Total instructions

(b) Branch mispredictions

(c) L1 data cache misses

(d) Total cycles

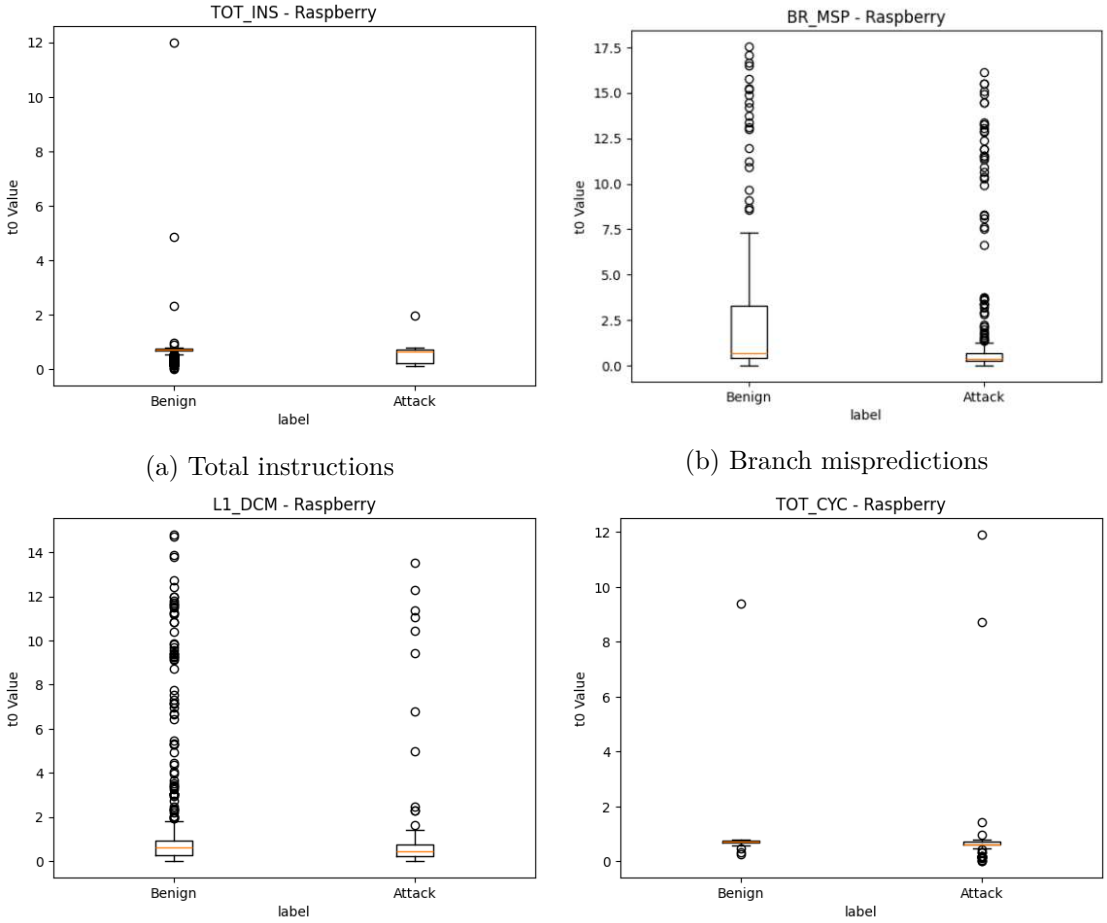
Figure 6.1: Boxplots for distribution of t0 in Intel dataset for different HPCs

time sequences), we created the assumption that the dynamic temporal approach would perform better with the data and learn the timing patterns to detect the attacks.

As described in Section 4.1, one of the aims of this work is to analyze whether the proposed approach is capable of generalization among different platforms, that is, if a model trained with data from one architecture (Intel, for example), would be able to detect an attack on data from another one (ARM, for example). Therefore, we defined different scenarios for our experiments, encompassing various combinations of data from both platforms (Intel and Raspberry Pi platforms), as detailed in Section 4.5.3.

### 6.3.1 Scenario 1: Separate Platforms

The first set of experiments was performed on both platforms separately, meaning that one model was trained on a dataset generated on one platform, and then applied to




(a) Total instructions

(b) Branch mispredictions

(c) L1 data cache misses

(d) Total cycles

Figure 6.2: Boxplots for distribution of t0 in Raspberry Pi dataset for different HPCs

a test dataset created on the same platform. That was repeated for both available platforms. The goal of these experiments was to establish a baseline and determine whether the model is able to detect attacks, achieving a performance comparable to the state-of-the-art.

For each of the platforms, the procedure was the same: a dataset was used for training and validation (split with the ratio 80-20), and another one (generated on the same platform) was used for testing. The preprocessing techniques are described in section 4.5.2. The implemented model is trained and validated using the generated datasets, and in sequence, evaluated with the test dataset. The implementation details and hyperparameters used are detailed on Section 5.3.

**Results**

The results of the experiments were evaluated using the performance metrics detailed in 5.3.3, for all three phases: training, validation, and test. Table 6.5 below shows the performance results for the present scenarios for both platforms.

Table 6.5: Results - Experiments on separate platforms

| Metric | Intel Platform | Raspberry Pi Platform |
|---|---|---|
| Train Accuracy | 1 | 0.97939 |
| Train Precision | 1 | 1 |
| Train Recall | 1 | 0.94637 |
| Train F1 | 1 | 0.97245 |
| Train TN Rate | 1 | 1 |
| Train Loss | 0 | 0.05335 |
| Val Accuracy | 1 | 0.95636 |
| Val Precision | 1 | 0.9397 |
| Val Recall | 1 | 1 |
| Val F1 | 1 | 0.96791 |
| Val TN Rate | 1 | 0.98 |
| Val Loss | 0.00002 | 0.12945 |
| Test Accuracy | 0.99941 | 1 |
| Test Precision | 0.99875 | 1 |
| Test Recall | 1 | 1 |
| Test F1 | 0.99937 | 1 |
| Test TN Rate | 0.99937 | 1 |
| Test Loss | 0.00902 | 0.00257 |

From the obtained results, we can observe that, in this scenario, the model is able to learn and detect attacks with a comparable performance as reported by the state-of-the-art. That establishes the baseline for the experiments executed in this work. The execution time for the experiments (on the available environment), which will be used for further comparison, varies between 2 minutes and 2.5 minutes.

### 6.3.2 Scenario 2: Combined Datasets

The next set of experiments has the goal of observing whether the proposed system is able to detect attacks among the combined data from both platforms. In order to do so, we performed experiments where the datasets used to train the model (training and validation datasets), were composed of a combination of the data collected in the Intel and Raspberry Pi platforms.

With the intention of guaranteeing that the model will not be biased towards one of the platforms, the combined datasets were created in accordance with the same proportion as before. More specifically, the dataset collected in the Intel platform was first randomly

split (80% for training, and 20% for validation), and the same was done for the dataset collected in the Raspberry Pi. Then, they were combined so that the new training and validation datasets used for this set of experiments had half of their samples collected on each platform.

The model trained on this scenario was then tested in different situations. In the first one, the test dataset was also created by combining the ones generated on both platforms. However, we also tested the model on separate datasets (from both platforms) in order to guarantee that the data from one platform does not outperform the other. Apart from the differences in the datasets, the experiments were performed in the same way as those in the previous scenario.

### Results

The results for training, validation, and test obtained in the experiments using combined training datasets are displayed on Table 6.6. That includes the situation with the combined test dataset, and also the individual test datasets from both platforms.

Table 6.6: Results - Experiments on combined datasets

| Metric | Combined test datasets | Intel test dataset | Raspberry Pi test dataset |
|---|---|---|---|
| Train Accuracy | 0.99152 | 0.99958 | 0.988 |
| Train Precision | 0.96609 | 0.99929 | 0.97569 |
| Train Recall | 0.99914 | 1 | 0.99315 |
| Train F1 | 0.98234 | 0.99965 | 0.98434 |
| Train TN Rate | 0.98916 | 0.9972 | 0.98484 |
| Train Loss | 0.02304 | 0.00217 | 0.03563 |
| Val Accuracy | 0.99966 | 0.935 | 0.94425 |
| Val Precision | 0.9975 | 0.98114 | 0.97543 |
| Val Recall | 1 | 0.92679 | 0.92031 |
| Val F1 | 0.99875 | 0.95261 | 0.94615 |
| Val TN Rate | 0.9996 | 0.95582 | 0.97304 |
| Val Loss | 0.00339 | 0.15494 | 0.13775 |
| Test Accuracy | 0.98781 | 0.997 | 0.94474 |
| Test Precision | 0.97771 | 0.99944 | 0.90965 |
| Test Recall | 0.991 | 0.99498 | 0.97771 |
| Test F1 | 0.98426 | 0.9972 | 0.94245 |
| Test TN Rate | 0.98426 | 0.9972 | 0.94245 |
| Test Loss | 0.04854 | 0.02356 | 0.13143 |

As can be observed in the results, the model behaved as expected and achieved a good performance when trained with the combined datasets. This is in accordance to our predictions, because the use of the combined dataset in the training enables the model to learn the behavior of the attack on both platforms. The fact that the test performance

metrics are slightly lower when tested on the dataset created on the Raspberry Pi platform is a possible initial indicator that the behavior of the microarchitectural traces might be different in both platforms.

One of the limitations related to this scenario is the fact that only the platforms encompassed in the training dataset are expected to perform well on the trained model. It does not guarantee that the model generalizes to unknown architectures. Therefore, the next set of experiments will cover this scenario.

### 6.3.3 Scenario 3: Cross-validation Across Platforms

The third set of experiments aims to evaluate whether the developed model is robust enough to generalize across different platforms, even if the training did not include any data collected on this platform. Our initial assumption is that the LSTM model, combined with the dynamic temporal approach, will enable the system to generalize and detect attacks on data collected on an unseen platform.

In order to perform the described evaluation, we created cross-validation experiments. This new set of experiments involved training the model on the dataset collected in one platform (which is split for training and validation, as before), and then performing the test on a dataset collected in another platform. The goal in this experiment is to determine whether the learning can be transferred across the two different platforms.

#### Initial Results

The experiments performed on this scenario included cross-validation in both directions: when the training dataset was collected on the Intel platform, and the test dataset generated on the Raspberry Pi platform, and the opposite, when the Raspberry Pi is used for training and the Intel dataset is used for testing.

Initially, we performed the experiments in this scenario using the same model as the previous experiments, without any additional features, in order to see how the system behaved. The initial results in both directions are displayed on Table 6.7.

As can be observed in the results, the current system was not able to transfer the learning among platforms, or, in other words, it does not generalize well. Given the obtained values, we decided to analyze the data collected in both platforms, and attempt to improve the performance using machine learning optimization techniques.

#### Normalization Results

As a possible way to improve the results, we decided to apply normalization to the timing values of the datasets. The reason for this choice is related to the fact that, due to the different architectures, the timing measurements on both datasets present significant differences.

Table 6.7: Results - Cross-validation Experiments

| Metric | Training Intel Test Rasp. Pi | Training Raspberry Pi Test Intel |
|---|---|---|
| Train Accuracy | 0.99912 | 1 |
| Train Precision | 0.99812 | 1 |
| Train Recall | 1 | 1 |
| Train F1 | 0.99906 | 1 |
| Train TN Rate | 0.99834 | 1 |
| Train Loss | 0.00419 | 0.00008 |
| Val Accuracy | 1 | 0.95727 |
| Val Precision | 1 | 0.96875 |
| Val Recall | 1 | 0.90107 |
| Val F1 | 1 | 0.93228 |
| Val TN Rate | 1 | 0.98623 |
| Val Loss | 0.00058 | 0.34896 |
| Test Accuracy | 0.78418 | 0.27671 |
| Test Precision | 0.5 | 0.49389 |
| Test Recall | 0.13745 | 0.11582 |
| Test F1 | 0.21562 | 0.18763 |
| Test TN Rate | 0.21562 | 0.18763 |
| Test Loss | 2.27906 | 6.60135 |

Therefore, we applied normalization techniques to both datasets and repeated the experiments. As mentioned in Section 4.5.2, the type of normalization chosen for our work was the zero-mean normalization (also called standardization).

The results of the experiments after the use of normalization techniques are shown in Table 6.8.

The displayed results indicate that, with the use of the current model, the learning was only transferred in one direction (when it is trained on the dataset created on the Raspberry Pi and tested on the dataset from the Intel platform). Therefore, we cannot state that, in this situation, the model is robust to generalization.

Our assumption for this result is related to the fact that both architectures behave differently on a microarchitectural scale, and, therefore, the HPC overflow patterns exhibit substantial differences. This can be observed in the boxplots from Figures 6.1 and 6.2, where we can see that the time intervals on the Raspberry Pi are distributed on a larger scale than the Intel platform. This is also in accordance with the observed fact that the model trained in the Raspberry Pi performs well on the Intel platform, but the opposite is not true. In order to try to overcome this challenge, we decided to incorporate the fine-tuning technique.

Table 6.8: Results - Normalization Experiments

| Metric | Training Intel Test Rasp. Pi | Training Raspberry Pi Test Intel |
|---|---|---|
| Train Accuracy | 0.99972 | 0.99971 |
| Train Precision | 1 | 0.99857 |
| Train Recall | 0.99964 | 1 |
| Train F1 | 0.99982 | 0.99929 |
| Train TN Rate | 1 | 0.99963 |
| Train Loss | 0.00036 | 0.00141 |
| Val Accuracy | 0.98444 | 0.96529 |
| Val Precision | 0.99938 | 0.76638 |
| Val Recall | 0.982 | 0.96667 |
| Val F1 | 0.99052 | 0.85148 |
| Val TN Rate | 0.99505 | 0.96696 |
| Val Loss | 0.12328 | 0.16475 |
| Test Accuracy | 0.79111 | 0.98827 |
| Test Precision | 0.80925 | 0.97075 |
| Test Recall | 0.95206 | 0.9948 |
| Test F1 | 0.87486 | 0.98263 |
| Test TN Rate | 0.87486 | 0.98263 |
| Test Loss | 2.42958 | 0.05853 |

**Fine-tuning Results**

As a possible optimization approach, the fine-tuning technique was integrated into the implemented system. Described in Section 4.5.4, this strategy involves starting from a pre-trained model and retraining only the relevant layers on a more specific dataset, so that the model specializes in a desired task.

In our case, we implemented this strategy on the cross-validation scenario, as a way for a model trained in one platform to execute the detection in a dataset from another one. This is expected to enable the learning to be transferred across platforms, but without the necessity of retraining the whole model.

The fine-tuning method was implemented and evaluated in both directions. In order to perform the retraining, a newly collected dataset was used, generated on the same platform as the test dataset, and (initially) with a similar size. The results comparing both directions of fine-tuning experiments are displayed in Table 6.9.

The results from the fine-tuning experiments (presented on Table 6.9) show that the addition of this technique enables the model to transfer the learning across platforms, without requiring a complete retraining of the deep-learning model.

In addition to observing the performance metrics, we also analyzed the time overhead resulting from this addition. In the experiments performed in this work, the distribution

Table 6.9: Results - Fine-Tuning Experiments

| Metric | Training Intel Test Rasp. Pi | Training Raspberry Pi Test Intel |
|---|---|---|
| Train Accuracy | 0.9388 | 0.99621 |
| Train Precision | 0.95152 | 0.99934 |
| Train Recall | 0.90752 | 0.99472 |
| Train F1 | 0.929 | 0.99702 |
| Train TN Rate | 0.96349 | 0.99884 |
| Train Loss | 0.15221 | 0.01869 |
| Val Accuracy | 1 | 0.995 |
| Val Precision | 1 | 0.99241 |
| Val Recall | 1 | 1 |
| Val F1 | 1 | 0.99618 |
| Val TN Rate | 1 | 0.98575 |
| Val Loss | 0.00017 | 0.0379 |
| Test Accuracy | 0.98358 | 0.9925 |
| Test Precision | 0.94692 | 0.99962 |
| Test Recall | 0.96634 | 0.98642 |
| Test F1 | 0.9527 | 0.99298 |
| Test TN Rate | 0.9527 | 0.99298 |
| Test Loss | 0.15326 | 0.02119 |

was observed: 70% of the experiment time is used for training and validation, while 30% is used for fine-tuning. This shows that this approach generates less time overhead in comparison with the situation where complete retraining is necessary.

In order to further reduce the overhead in this approach, we experimented with fine-tuning datasets of different sizes to determine which is the smallest dataset that can still perform successfully in our scenario. The evaluated sizes for the fine-tuning datasets were 1000 and 5000 samples (in addition to the initial size of 20000 samples). For the sake of conciseness, we will display the results in one direction only. Since our experiments show that the transfer from a model trained in a dataset collected on the Intel platform to a test dataset collected on the Raspberry Pi did not perform well in the previous experiment, we will observe this direction. In this way, the difference is more visible and a better evaluation is possible. The test results for the experiments on the fine-tuning dataset size are displayed in Table 6.10.

The results displayed on the table show that even though there is still room for improvement, the fine-tuning dataset containing 5000 samples can already reach performance metrics with values around 90%, which is a significant improvement compared to the initial situation.

This set of experiments shows that, with the use of the fine-tuning technique, it is possible for the model trained in one platform to perform the attack detection in another one

Table 6.10: Results - Fine-tuning Size Experiments

| Metric | 20000 samples | 1000 samples | 5000 samples |
|---|---|---|---|
| Test Accuracy | 0.98358 | 0.8743 | 0.91526 |
| Test Precision | 0.94692 | 0.87019 | 0.8937 |
| Test Recall | 0.96634 | 0.81179 | 0.95497 |
| Test F1 | 0.9527 | 0.8446 | 0.91253 |
| Test TN Rate | 0.9527 | 0.8446 | 0.91253 |
| Test Loss | 0.15326 | 0.37579 | 0.22692 |

without the need to completely retrain the model, and using a fine-tuning dataset with a smaller size (around 25% of the original number of samples). This leads to a smaller overhead, both in the size of the dataset and in the time necessary to generate it.

CHAPTER 7

# Conclusion

Recent advances in computer architecture make access to information faster and more efficient, but also make the computer prone to attacks and information leakage. One type of attack that makes use of this and poses a threat to modern computer architectures is cache-based side-channel attacks, such as the Spectre Attack, for example. Even though this type of attacks, which have an effect on the microarchitectural traces of the machine, has been studied by several works that aim to detect it, there are still challenges to overcome. Among them, it is the fact that current works use the sampling technique at a defined interval to monitor HPC events, not being flexible to adapt and detect attacks that may occur between the sampling intervals. In addition, state-of-the-art studies are usually limited to a single target architecture, which does not guarantee the robustness of a model that generalizes well. Therefore, this thesis aims to develop an approach to detect Spectre attacks by observing microarchitectural traces of the computer and using machine learning techniques.

To this end, this work makes the use of an overflow-based approach and T-LSTM technique with the goal of attempting to generalize the model and transfer learning between different architectures. The developed system includes an initial step to calibrate to a new platform and define threshold values for each HPC that will be monitored. The values are used in the dataset creation as overflow thresholds, where a sample is collected for the dataset each time an overflow is triggered by one of the HPCs. The dataset composed by the sequence of triggered overflows, along with the time interval between overflows is submitted as input to a T-LSTM model, which is trained to detect cache side-channel attacks. Optimization techniques, as normalization and fine-tuning, are used to improve the model's performance.

The experiments carried out in this work were divided into different scenarios, in order to evaluate different aspects of the system. Initially, separate models were trained, one for each evaluated platform. The trained models were then tested with samples from the same platform on which they were trained, achieving results of over 99% in both

cases. The datasets generated on both platforms were then combined in a new training dataset, and this trained model was evaluated both on a combination and with separate test datasets. The obtained outcomes were also satisfactory, achieving the test accuracy between 95% and 99.7%. Finally, the cross-validation scenario was tested, in which a model trained with data from one platform was tested on another. The results of these experiments revealed that the model was not generalized across platforms, and our initial hypothesis was not confirmed.

As a way of improving the performance in this scenario, additional techniques were used, namely normalization and fine-tuning. This enabled the system to enhance the detection on a new platform, without the need to completely retrain the model. The test results showed that, with the use of these techniques, the system achieved 98% accuracy on the Raspberry Pi platform and 99% accuracy on the Intel platform. Further experiments also indicated that a fine-tuning dataset of 5000 samples is enough to achieve all the performance metrics above 90%. This suggests that the overhead created is significantly small, in comparison with the need to generate a complete dataset and train the model again.

One of the contributions of this work, as explained in Chapter 1, is the development of an overflow-based dataset through the monitoring of microarchitectural traces (Hardware Performance Counters). The used approach enables the system to calibrate the thresholds to the target machine, offering flexibility for the application in unseen platforms. In addition, the adaptability of the overflow technique ensures a constant data flow, even in different workloads, and reduces the space and time overhead. The developed system can identify the presence of Spectre and Rowhammer attacks in two architectures (Intel and ARM), and the use of a fine-tuning technique enables the use of the model for unknown platforms, while requiring less data and time than a complete model retraining would.

## 7.1 Future Works

The challenges and limitations related to the proposed approach are outlined on Chapter 6. Therefore, the work described on this thesis raises several improvement possibilities.

One of the most important aspects to improve in the proposed system is the ability to generalize for different platforms. Possible approaches that could be experimented with include testing different preprocessing and normalization techniques, different model hyperparameters, or a different model architecture. In addition, a different calibration algorithm may enable the generation of datasets that improve the machine learning model training.

An important matter to consider would be to also guarantee that the system is safe to data poisoning and other security threats. Especially when dealing with safety risks, as is the case with this work, an essential future work would be to ensure that the model training is performed in a safe environment and is not at risk of being poisoned by an attacker.

A technique that could be tested is the use of an attention mechanism in the deep learning model. The idea behind this approach is to enable the neural network model to focus on the selective and most relevant parts of the input data. In our case, it may be the case that this feature helps the model to better identify the attack patterns, even with data from an unknown platform.

Moreover, future works also include further ways of assessing the robustness of the model. This includes the use of different attacks or different classes of benign applications, as well as the presence of noise or randomization. These would make it more difficult for the system to detect attacks, and it would guarantee that the model is robust in difficult environments.

# Overview of Generative AI Tools Used

No Artificial Intelligence tools were used in the development of this thesis, except for the initial translation of the Abstract. For this task, it was used Chat-GPT Model 5.

# List of Figures

# List of Tables

# Acronyms

**ANN** Artificial Neural Network. 15, 16

**ARM** Advanced RISC Machines. 12

**BTB** Branch Target Buffer. 11

**CNN** Convolutional Neural Network. 15, 22

**CPU** Central Processing Unit. 1, 5, 9–14, 27

**DL** Deep Learning. 15, 32

**DNN** Deep Neural Network. 15

**DRAM** Dynamic Random-Access Memory. 12

**FN** False Negatives. 38

**FP** False Positives. 38

**GA** Genetic Algorithm. 22

**GBC** Gradient Boosting Classifier. 18, 20

**HPC** Hardware Performance Counter. 1, 2, 4, 13–15, 20, 25–30, 36, 41, 42, 44, 50, 55

**kNN** K-Nearest Neighbors. 18, 20

**LDA** Linear Discriminant Analysis. 18, 20

**LLC** Last-Level Cache. 5

**LR** Logistic Regression. 18, 20

**LSTM** Long Short-Term Memory. 3, 4, 16–18, 20–22, 26, 30, 31, 33, 35, 42, 49, 61

**ML** Machine Learning. 1, 15, 18, 28, 31, 33

**MLP** Multilayer Perceptron. 18, 20

**mRMR** Maximum Relevance Minimum Redundance. 27

**PHT** Pattern History Table. 10

**PMU** Performance Monitor Unit. 13

**RAM** Random-Access Memory. 5

**RF** Random Forest. 18, 20

**RNN** Recurrent Neural Network. 4, 15–18, 21, 22, 61

**RSB** Return Stack Buffer. 11

**SCA** Side-channel Attack. 6, 27

**SGX** Software Guard Extensions. 13

**STL** Store To Load. 12

**SVC** Support Vector Classifier. 18, 20

**SVM** Support Vector Machine. 20

**T-LSTM** Time-Aware Long Short-Term Memory. 31, 32, 35, 37, 44, 55

**TN** True Negatives. 38

**TP** True Positives. 38

# Bibliography


[Alp20] Ethem Alpaydin. *Introduction to machine learning*. MIT press, 2020.

[AP22] Hossein Abbasimehr and Reza Paki. Improving time series forecasting using lstm and attention models. *Journal of Ambient Intelligence and Humanized Computing*, 13:1–19, 01 2022.

[AZW24] Mai AL-Zu'bi and Georg Weissenbacher. Statistical profiling of micro-architectural traces and machine learning for spectre detection: A systematic evaluation. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6, 2024.

[BMW+18] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient Out-of-Order execution. In *27th USENIX Security Symposium (USENIX Security 18)*, page 991–1008, Baltimore, MD, August 2018. USENIX Association.

[BPSS24] Pavitra Bhade, Joseph Paturel, Olivier Sentieys, and Sharad Sinha. Lightweight hardware-based cache side-channel attack detection for edge devices (edge-cascade). *ACM Trans. Embed. Comput. Syst.*, 23(4), June 2024.

[BRN24] Swapnil Baviskar, Sanoj R, and Hiran V Nath. Cache based side-channel attacks: A survey. In *2024 IEEE Recent Advances in Intelligent Computational Systems (RAICS)*, pages 1–8, 2024.

[BXZ+17] Inci M. Baytas, Cao Xiao, Xi Zhang, Fei Wang, Anil K. Jain, and Jiayu Zhou. Patient subtyping via time-aware lstm networks. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, page 65–74, New York, NY, USA, 2017. Association for Computing Machinery.

[BYL21] Yohannes Bekele, Ahmed Yiwere, and Daniel B Limbrick. Rowhammer attacks on the raspberry pi 3b+. In *Government Microcircuit Applications & Critical Technologies Conference*, 2021.

[CBS+19] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 249–266, Santa Clara, CA, August 2019. USENIX Association.

[Elm90] Jeffrey L. Elman. Finding structure in time. *Cognitive Science*, 14(2):179–211, 1990.

[Fer19] Serena Ferracci. Detecting cache-based side channel attacks using hardware performance counters. mathesis, Sapienza, University of Rome, 2019.

[FLM+18] Yunsheng Fu, Fang Lou, Fangzhi Meng, Zhihong Tian, Hua Zhang, and Feng Jiang. An intelligent network attack detection method based on rnn. In *2018 IEEE Third International Conference on Data Science in Cyberspace (DSC)*, pages 483–489, 2018.

[FTS20] Hongqing Fang, Pei Tang, and Hao Si. Feature selections using minimal redundancy maximal relevance algorithm for human activity recognition in smart home environments. *Journal of Healthcare Engineering*, 2020(1):8876782, 2020.

[GMES19] Berk Gülmezoglu, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. Fortuneteller: Predicting microarchitectural attacks via unsupervised deep learning. *ArXiv*, abs/1907.03651, 2019.

[GMM16] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A remote software-induced fault attack in javascript. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, DIMVA, Germany, July 2016. Springer Vieweg.

[GMWM16] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+flush: A fast and stealthy cache attack. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9721:279 – 299, 2016. Cited by: 378.

[Gre14] Brendan Gregg. *Systems performance: enterprise and the cloud*. Pearson Education, 2014.

[HL17] Zecheng He and Ruby B. Lee. How secure is your cache against side-channel attacks? In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, page 341–353, New York, NY, USA, 2017. Association for Computing Machinery.

[Hor18] Horn. Speculative execution, variant 4: Speculative store bypass, 2018. `https://project-zero.issues.chromium.org/issues/42450580` [Accessed: 2025-08-14].

[HS97a] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[HS97b] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[HTF09] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition (Springer Series in Statistics)*. 02 2009.

[Int25] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3 : System Programming Guide*, 2025. Available at `https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html`.

[Jia24] Yuchen Jiang. Implemention and analysis of various spectre attacks. Bachelor's thesis, Technische Universität Wien, 2024.

[KHF+20] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: exploiting speculative execution. *Commun. ACM*, 63(7):93–101, June 2020.

[KHK+24] Hodong Kim, Changhee Hahn, Hyunwoo J. Kim, Youngjoo Shin, and Junbeom Hur. Deep learning-based detection for multiple cache side-channel attacks. *IEEE Transactions on Information Forensics and Security*, 19:1672–1686, 2024.

[KKSAG18] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *Proceedings of the 12th USENIX Conference on Offensive Technologies*, WOOT'18, page 3, USA, 2018. USENIX Association.

[LGS+16] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache attacks on mobile devices. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 549–564, Austin, TX, August 2016. USENIX Association.

[LLLY19] Hongyu Liu, Bo Lang, Ming Liu, and Hanbing Yan. Cnn and rnn based payload classification methods for attack detection. *Knowledge-Based Systems*, 163:332–341, 2019.

[LM18] Yangdi Lyu and Prabhat Mishra. A survey of side-channel attacks on caches and countermeasures. *Journal of Hardware and Systems Security*, 2, 03 2018.

[LSG+20] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, Mike Hamburg, and Raoul Strackx. Meltdown: Reading kernel memory from user space. *Communications of the ACM*, 63(6):46–56, May 2020.

[MAB+18] Maria Mushtaq, Ayaz Akram, Muhammad Khurram Bhatti, Naveed Bin Raees Rao, Vianney Lapotre, and Guy Gogniat. Run-time Detection of Prime+Probe Side-Channel Attack on AES Encryption Algorithm. In *Global Information Infrastructure and Networking Symposium (GIIS)*, Thessaloniki, Greece, October 2018.

[MCY+20] Pramita Muhuri, Prosenjit Chatterjee, Xiaohong Yuan, Kaushik Roy, and Albert Esterline. Using a long short-term memory recurrent neural network (lstm-rnn) to classify network attacks. *Information*, 11:243, 05 2020.

[MMB+20] M. Asim Mukhtar, Maria Mushtaq, M. Khurram Bhatti, Vianney Lapotre, and Guy Gogniat. Flush + prefetch: A countermeasure against access-driven cache-based side-channel attacks. *Journal of Systems Architecture*, 104:101698, 2020.

[Moo02] Shirley V. Moore. A comparison of counting and sampling modes of using performance monitoring hardware. In *Proceedings of the International Conference on Computational Science-Part II*, ICCS '02, page 904–912, Berlin, Heidelberg, 2002. Springer-Verlag.

[MR18] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using return stack buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 2109–2122, New York, NY, USA, 2018. Association for Computing Machinery.

[nuc24] Intel® nuc products nuc10i3fn/nuc10i5fn/ nuc10i7fn - technical product specification, 2024. `https://www.intel.com/content/www/us/en/content-details/841263/intel-nuc-products-nuc10i3fn-nuc10i5fn-nuc10i7fn-technical-produc` `html` [Accessed: 2025-08-27].

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology*, CT-RSA'06, page 1–20, Berlin, Heidelberg, 2006. Springer-Verlag.

[pera] perf: Linux profiling with performance counters. `https://perfwiki.github.io/main/` [Accessed: 2025-08-26].

[Perb] perf_event_open(2) — linux manual page. `https://man7.org/linux/man-pages/man2/perf_event_open.2.html` [Accessed: 2025-08-18].

[perc] Perfmon events - intel processors. https://perfmon-events.intel.com/ [Accessed: 2025-08-25].

[perd] Pmu events - arm cortex a72. https://developer.arm.com/documentation/100095/0002/performance-monitor-unit/events?lang=en [Accessed: 2025-08-25].

[PSCH21] Guansong Pang, Chunhua Shen, Longbing Cao, and Anton Van Den Hengel. Deep learning for anomaly detection: A review. *ACM Comput. Surv.*, 54(2), March 2021.

[ras] Raspberry pi 4 tech specs. https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/ [Accessed: 2025-08-26].

[Ras22] Raspberry Pi Ltd. *Datasheet Raspberry Pi BCM2711 ARM Peripherals*, 2022. Available at https://datasheets.raspberrypi.com/bcm2711/bcm2711-peripherals.pdf.

[rowa] Program for testing for the dram "rowhammer" problem. https://github.com/google/rowhammer-test [Accessed: 2025-08-26].

[rowb] rowhammer_rpi3. https://github.com/developedby/rowhammer_rpi3 [Accessed: 2025-08-26].

[Sch15] Pia Schwarzinger. Integrating causal discovery and machine learning for enhanced financial forecasting. dithesis, Technische Universität Wien, 2015.

[SCZ21] Chaoqun Shen, Congcong Chen, and Jiliang Zhang. Micro-architectural cache side-channel attacks and countermeasures. In *2021 26th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 441–448, 2021.

[SDO23] Ishu Sharma, Rajat Dubey, and Sharad Shyam Ojha. Exploit detection and mitigation technique of cache side-channel attacks using artificial intelligence. In *2023 2nd International Conference on Automation, Computing and Renewable Systems (ICACRS)*, pages 995–1001, 2023.

[spe] Spectre variant 2 poc. https://github.com/Anton-Cao/spectrev2-poc [Accessed: 2025-08-26].

[Spe18] Spectre attack implementation, 2018. https://github.com/Eugnis/spectre-attack [Accessed: 2025-08-14].

[TZW+20] Zhongkai Tong, Ziyuan Zhu, Zhanpeng Wang, Limin Wang, Yusha Zhang, and Yuxin Liu. Cache side-channel attacks detection based on machine learning. In *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 919–926, 2020.

[YF14] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, l3 cache Side-Channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, San Diego, CA, August 2014. USENIX Association.

[YJDO19] Lin Yao, Binyao Jiang, Jing Deng, and Mohammad S. Obaidat. Lstm-based detection for timing attacks in named data network. In *2019 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6, 2019.

[Zho21] Zhi-Hua Zhou. *Machine Learning*. 01 2021.

[ZZL16] Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee. Cloudradar: A real-time side-channel attack detection system in clouds. In Fabian Monrose, Marc Dacier, Gregory Blanc, and Joaquin Garcia-Alfaro, editors, *Research in Attacks, Intrusions, and Defenses*, pages 118–140, Cham, 2016. Springer International Publishing.

# Appendix

## Implementation of Dataset Creation

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <fcntl.h>
#include <signal.h>
#include <sys/syscall.h>
#include <linux/perf_event.h>
#include <sys/ioctl.h>
#include <time.h>
#include <dirent.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <time.h>
#include <dirent.h>

#define MAX_PROCESSES 100
#define MAX_SAMPLES 100000
#define NUM_COUNTERS 4

int thresholds[NUM_COUNTERS];

// Store the info on the PID and 4 HPCs
typedef struct {
    pid_t pid;
    int fds[NUM_COUNTERS];
} monitored_proc_t;

// List of processes that are being monitored
monitored_proc_t procs[MAX_PROCESSES];
int num_procs = 0;
int list_pids[100000];
int list_pids_attached[1000];
int list_pids_seen[100000] = {0};
```

```
int list_eventsets[1000];
long long int count_pid = 0;
int count_pid_attached = 0;
int count_pid_seen = 0;

// Save last value of each HPC for each process
long long last_value_tot_ins[MAX_PROCESSES] = {0};
long long last_value_br_msp[MAX_PROCESSES] = {0};
long long last_value_l1_dcm[MAX_PROCESSES] = {0};
long long last_value_tot_cyc[MAX_PROCESSES] = {0};

// Save the new value of each HPC (with difference from the last one)
long long value_trigger;

// Store HPC values
long long values[NUM_COUNTERS];
long long last_values[NUM_COUNTERS][MAX_PROCESSES] = {0};

// Timing info
clock_t start_time, end_time;
long long t0;
long long last_usec_HPC[NUM_COUNTERS] = {0};
long long cur_usec_HPC[NUM_COUNTERS];

// Overflow counter
int counter = 0;
long long overflow_log[100000][9] = {0};
char overflow_name_log[100000][20];
char list_names_attached[100000][1024];
int label = 0;

int isProcessDir(const struct dirent *entry) {
    // Check if the entry name is a number (PID)
    for (int i = 0; i < strlen(entry->d_name); i++) {
        if (!isdigit(entry->d_name[i]))
            return 0;
    }
}

const char* get_process_name_by_pid(const int pid)
{
    char* name_pid = (char*)calloc(1024,sizeof(char));
    char* name_cmd = (char*)calloc(1024,sizeof(char));

    if(name_pid){
        sprintf(name_pid, "/proc/%d/comm",pid);
        FILE* f = fopen(name_pid,"r");
        if(f){
            size_t size;
            size = fread(name_cmd, sizeof(char), 1024, f);
      if(size>0){
                if('\n'==name_cmd[size-1])
                    name_cmd[size-1]='\0';
```

```
92              }
93              fclose(f);
94          }
95      }
96      return name_cmd;
97  }
98
99  // Handler overflow
100 void handle_event(int signo, siginfo_t *info, void *context) {
101
102     // Get t1: Time since last overflow
103     clock_t end_time = clock();
104     long long t1 = llabs((long long)(((double)(end_time - start_time) /
        CLOCKS_PER_SEC)*1000000));
105
106     // Get overflow trigger
107     int fd = info->si_fd;
108     for (int i = 0; i < num_procs; i++) {
109
110         monitored_proc_t *p = &procs[i];
111         long long val;
112         char name_overflow[1024];
113         int pid_index;
114
115         // Get the PID and HPC that caused the overflow (1 HPC at a time)
116         if (fd == p->fds[0]) { // TOT_INS
117
118             // Read the trigger value
119             read(fd, &val, sizeof(val));
120             value_trigger = val - last_value_tot_ins[i];
121             last_value_tot_ins[i] = val;
122             values[0] = llabs(val);
123             last_values[0][i] = llabs(val);
124
125             // Read other values
126             int k;
127             for(k=0; k < NUM_COUNTERS; k++) {
128                 if(k==0) continue;
129                 long long other_val;
130                 read(p->fds[k], &other_val, sizeof(other_val));
131                 values[k] = llabs(other_val);
132                 last_values[k][i] = llabs(other_val);
133             }
134
135             // Get the time and calculate t0
136             cur_usec_HPC[0] = (long long)end_time;
137             t0 = llabs((long long)(((double)(cur_usec_HPC[0] - last_usec_HPC
        [0]) / CLOCKS_PER_SEC)*1000000));
138
139             // Get process name
140             int j;
141             for(j=0; j< count_pid_attached; j++) {
142                 if(p->pid == list_pids_attached[j]) {
```

```
                    pid_index = j;
                    strcpy(name_overflow, list_names_attached[j]);
                }
            }

            // Get label (attack or benign)
            if(name_overflow) {
                if (strstr(name_overflow, "spectre") != NULL) {
                    label = 1;
                } else if(strstr(name_overflow, "pht_flush") != NULL) {
                    label = 1;
                } else if(strstr(name_overflow, "hprh") != NULL) {
                    label = 1;
                } else {
                    label = 0;
                }
            }

            strcpy(overflow_name_log[counter], name_overflow);
            overflow_log[counter][0] = 1;
            overflow_log[counter][1] = value_trigger;
            overflow_log[counter][2] = values[1];
            overflow_log[counter][3] = values[2];
            overflow_log[counter][4] = values[3];
            overflow_log[counter][5] = t0;
            overflow_log[counter][6] = t1;
            overflow_log[counter][7] = label;
            overflow_log[counter][8] = counter;
            last_usec_HPC[0] = cur_usec_HPC[0];

        } // Repeated for other HPC events
    }
    counter++;
    start_time = clock();
}

// Function to setup the overflow signal
void setup_signal(int signum) {
    struct sigaction sa = {0};
    sa.sa_sigaction = handle_event;
    sa.sa_flags = SA_SIGINFO;
    sigaction(signum, &sa, NULL);
}

// Attach the overflow signal to each process and PID
void attach_signal(int fd, int signo) {
    fcntl(fd, F_SETFL, O_ASYNC);
    fcntl(fd, F_SETSIG, signo);
    fcntl(fd, F_SETOWN, getpid());
}

// Perf function for HPC monitoring
long perf_event_open(struct perf_event_attr *attr, pid_t pid,
```

```
                    int cpu, int group_fd, unsigned long flags) {
    return syscall(__NR_perf_event_open, attr, pid, cpu, group_fd, flags);
}

// Create and open perf counter given the name and threshold
int open_counter(pid_t pid, int type, int config, long period, int signo) {
    struct perf_event_attr pe = {0};
    pe.type = type;
    pe.size = sizeof(pe);
    pe.config = config;
    pe.sample_period = period;
    pe.exclude_kernel = 1;
    pe.exclude_hv = 1;
    pe.disabled = 0;

    int fd = perf_event_open(&pe, pid, -1, -1, 0);
    if (fd == -1) return -1;
    attach_signal(fd, signo);
    return fd;
}

// Create and open perf counter for the cache HPC
int open_cache_event(pid_t pid, int cache, int op, int result, long period,
    int signo) {
    struct perf_event_attr pe = {0};
    pe.type = PERF_TYPE_HW_CACHE;
    pe.size = sizeof(pe);
    pe.config = cache | (op << 8) | (result << 16);
    pe.sample_period = period;
    pe.exclude_kernel = 1;
    pe.exclude_hv = 1;

    int fd = perf_event_open(&pe, pid, -1, -1, 0);
    if (fd == -1) return -1;
    attach_signal(fd, signo);
    return fd;
}

// Function to setup PID and counters for each process to be monitored
int setup_proc(pid_t pid, monitored_proc_t *proc) {
    proc->pid = pid;

    // TOT_INS
    proc->fds[0] = open_counter(pid, PERF_TYPE_HARDWARE,
    PERF_COUNT_HW_INSTRUCTIONS, thresholds[0], SIGRTMIN);
    if (proc->fds[0] == -1) return -1;

    // BR_MSP
    proc->fds[1] = open_counter(pid, PERF_TYPE_HARDWARE,
    PERF_COUNT_HW_BRANCH_MISSES, thresholds[1], SIGRTMIN + 1);
    if (proc->fds[1] == -1) return -1;

    // L1_DCM
```

```
246     proc->fds[2] = open_cache_event(pid, PERF_COUNT_HW_CACHE_L1D,
247                                     PERF_COUNT_HW_CACHE_OP_READ,
248                                     PERF_COUNT_HW_CACHE_RESULT_MISS,
249                                     thresholds[2], SIGRTMIN + 2);
250     if (proc->fds[2] == -1) return -1;
251
252     // TOT_CYC
253     proc->fds[3] = open_counter(pid, PERF_TYPE_HARDWARE,
        PERF_COUNT_HW_CPU_CYCLES, thresholds[3], SIGRTMIN+3);
254     if (proc->fds[3] == -1) return -1;
255
256     return 0;
257 }
258
259 int main(int argc, char **argv) {
260
261     if (argc - 1 > MAX_PROCESSES) {
262         fprintf(stderr, "Max %d processes allowed\n", MAX_PROCESSES);
263         return 1;
264     }
265
266     /* Variables needed for monitoring the processes */
267     DIR *dir;
268     struct dirent *entry;
269
270     // Setting interrupt signals for overflow of 4 HPCs
271     setup_signal(SIGRTMIN);
272     setup_signal(SIGRTMIN + 1);
273     setup_signal(SIGRTMIN + 2);
274     setup_signal(SIGRTMIN + 3);
275
276     /*Open and read threshold file */
277     FILE *fptr;
278     fptr = fopen("thresholds.txt", "r");
279
280
281     char myString[100];
282     while(fgets(myString, 100, fptr)) {
283       //printf("%s\n", myString);
284     }
285
286     char *token = strtok(myString, ",");
287     int counter_HPC = 0;
288     while (token != NULL)
289     {
290         thresholds[counter_HPC] = atoi(token);
291         if(thresholds[counter_HPC] == 0) thresholds[counter_HPC] = 100;
292         counter_HPC++;
293         token = strtok(NULL, ",");
294     }
295
296     fclose(fptr);
297
```

```
298     FILE *file_dataset;
299     file_dataset = fopen("dataset.csv", "w");
300 
301 
302     fprintf(file_dataset, "process_name,trigger,TOT_INS,BR_MSP,L1_DCM,TOT_CYC
        ,t0,t1,label,sample\n");
303 
304     // Iterate until dataset is done (MAX_SAMPLES reached)
305     int iterations = 0;
306 
307     while (counter < MAX_SAMPLES) {
308 
309         iterations++;
310 
311         // Open the /proc directory
312         dir = opendir("/proc");
313         if (dir == NULL) {
314             perror("opendir");
315             return 1;
316         }
317 
318         // List all pids
319         count_pid = 0;
320         while ((entry = readdir(dir)) != NULL) {
321 
322             if (isProcessDir(entry)) {
323                 int pid_num = atoi(entry->d_name);
324 
325                 list_pids[count_pid] = pid_num;
326                 count_pid++;
327             }
328         }
329 
330         closedir(dir);
331 
332         int i;
333 
334         // Attach pids and eventsets
335         for(i=0; i< count_pid; i++) {
336 
337             int j;
338             int attached = 0;
339             int seen = 0;
340 
341             // Check if process is attached
342             for (j=0; j<count_pid_attached; j++) {
343                 if(list_pids_attached[j] == list_pids[i]) {
344                     attached = 1;
345                 }
346             }
347 
348             // Check if process was seen (tried to attach)
349             if(list_pids_seen[i] == 1) seen = 1;
```

```
350
351             // If it is the first time that I am seeing this process
352             if ((attached == 0)&&(seen==0)) {
353
354                 char path[64];
355                 snprintf(path, sizeof(path), "/proc/%d", list_pids[i]);
356                 if (access(path, F_OK) != 0) continue;
357
358                 pid_t pid = list_pids[i];
359
360                 // Setup the HPC counters and overflows for the process
361                 if (setup_proc(pid, &procs[i]) < 0) {
362                     printf("Error %d\n", pid);
363                     perror("setup_proc failed");
364                     if(seen == 0) {
365                         list_pids_seen[i] = 1;
366                         count_pid_seen++;
367                     }
368                 } else {
369
370                     // Start counting for timer
371                     start_time = clock();
372
373                     // Get process name
374                     char* name = get_process_name_by_pid(list_pids[i]);
375
376                     // Save name and PID of process being monitored
377                     list_pids_attached[count_pid_attached] = list_pids[i];
378               strcpy(list_names_attached[count_pid_attached],name);
379                   count_pid_attached++;
380                 }
381             }
382         }
383     }
384
385     int i;
386     for(i=0; i<counter;i++) {
387         if(overflow_log[i][0] != 0) {
388             fprintf(file_dataset, "%s,",overflow_name_log[i]);
389             fprintf(file_dataset, "%d,%lld,%lld,%lld,%lld,%lld,%lld,%d,%d\n",
     overflow_log[i][0], overflow_log[i][1], overflow_log[i][2], overflow_log
    [i][3], overflow_log[i][4], overflow_log[i][5], overflow_log[i][6],
    overflow_log[i][7], overflow_log[i][8]);
390         }
391     }
392     fclose(file_dataset);
393     return 0;
394 }
```

Listing 1: Code for dataset creation

```

from tlstm import *
import torch
import numpy as np
import pytorch_lightning as pl
from pytorch_lightning.loggers import WandbLogger
import wandb
from collections import Counter

torch.set_float32_matmul_precision('medium')
seq_len = 100

# Scenario and datasets to be used
eval_scenario = 1
train_NUC = 1
train_Rasp = 0
test_NUC = 0
test_Rasp = 1

# SCENARIO 3: CROSS-VALIDATION

# Getting training dataset from file
if(train_NUC):
    name_train_data = './datasets/train_data_intel.csv'
    print('Train NUC: '+ name_train_data)
elif (train_Rasp):
    name_train_data = './datasets/train_data_rasp.csv'
    print('Train Rasp: '+ name_train_data)

data = torch.tensor(np.loadtxt(name_train_data, delimiter=','), dtype=torch.
    long)
data = data[data[:, 7]<=1]
data = torch.stack([
        data[:, 0] - 1, data[:, 5], data[:, 7]
]).T
data = data[len(data)%seq_len:].reshape(-1, seq_len, 3)

# Getting test dataset from file
if(test_NUC):
    name_test_data = './datasets/test_data_intel.csv'
    print('Test NUC: '+ name_test_data)
elif (test_Rasp):
    name_test_data = './datasets/test_data_rasp.csv'
    print('Test Rasp: '+ name_test_data)

data_test = torch.tensor(np.loadtxt(name_test_data, delimiter=','), dtype=
    torch.long)
data_test = data_test[data_test[:, 7]<=1]

data_test = torch.stack([
        data_test[:, 0] - 1, data_test[:, 5], data_test[:, 7]
```

```
]).T
data_test = data_test[len(data_test)%seq_len:].reshape(-1, seq_len, 3)

# Train Dataset
class Dataset(torch.utils.data.Dataset):
    def __init__(self):
        self.signal = data[:, :, 0]
        self.timing = data[:, :, 1].to(torch.float)
        self.y = data[:, :, 2]
        self.n_labels = self.y.max().item() + 1
    def __len__(self):
        return len(self.signal)
    def __getitem__(self, idx):
        return self.signal[idx], self.timing[idx], self.y[idx]

data = Dataset()

# Data Normalization
mean, std, var = torch.mean(data.timing), torch.std(data.timing), torch.var(
    data.timing)
data.timing = abs((data.timing-mean)/std)

# Divide into train + val subsets
train_ratio = 0.8
train_samples = int(train_ratio * len(data))
train, val  = torch.utils.data.random_split(data, [train_samples, len(data) -
     train_samples])

train_loader = torch.utils.data.DataLoader(train, batch_size=len(train)//2,
    num_workers=15, shuffle=True)
val_loader = torch.utils.data.DataLoader(val, batch_size=len(val)//2,
    num_workers=15, shuffle=False)

# Test Dataset
class Test_Dataset(torch.utils.data.Dataset):
    def __init__(self):
        self.signal = data_test[:, :, 0]
        self.timing = data_test[:, :, 1].to(torch.float)
        self.y = data_test[:, :, 2]
        self.n_labels = self.y.max().item() + 1
    def __len__(self):
        return len(self.signal)
    def __getitem__(self, idx):
        return self.signal[idx], self.timing[idx], self.y[idx]

data_test = Test_Dataset()

# Test data normalization
mean, std, var = torch.mean(data_test.timing), torch.std(data_test.timing),
    torch.var(data_test.timing)
data_test.timing = abs((data_test.timing-mean)/std)

test_loader = torch.utils.data.DataLoader(data_test, batch_size=len(data_test
```

```
    ), num_workers=15, shuffle=False)

# Define model
model = TLSTM(2, 4, 64, 50, lr=1e-3, freeze_lstm=False)

# Define logger and trainer
logger = WandbLogger(project="spectre", group="test")
trainer = pl.Trainer(
    max_epochs=100,
    logger=logger,
    check_val_every_n_epoch=10,
    devices=1
)

# Train model
trainer.fit(
    model,
    train_loader,
    val_loader
)

# Save model trained
trainer.save_checkpoint("tlstm.ckpt")

# Define dataset for fine-tuning
if(test_NUC):
    name_ft_data = './datasets/fine_tune_data_intel.csv'
    print('Fine-tune NUC: ' + name_ft_data)
elif (test_Rasp):
    name_ft_data = './datasets/fine_tune_data_rasp.csv'
    print('Fine-tune Rasp: ' + name_ft_data)

# Load dataset and get relevant columns
ft_data = torch.tensor(np.loadtxt(name_ft_data, delimiter=','), dtype=torch.
    long)
ft_data = ft_data[ft_data[:, 7]<=1]
ft_data = torch.stack([
        ft_data[:, 0] - 1, ft_data[:, 5], ft_data[:, 7]
]).T
ft_data = ft_data[len(ft_data)%seq_len:].reshape(-1, seq_len, 3)

# Fine-tune Dataset
class Fine_Tune_Dataset(torch.utils.data.Dataset):
    def __init__(self):
        self.signal = ft_data[:, :, 0]
        self.timing = ft_data[:, :, 1].to(torch.float)
        self.y = ft_data[:, :, 2]
        self.n_labels = self.y.max().item() + 1
    def __len__(self):
        return len(self.signal)
    def __getitem__(self, idx):
        return self.signal[idx], self.timing[idx], self.y[idx]

```

```
ft_data = Fine_Tune_Dataset()

mean, std, var = torch.mean(ft_data.timing), torch.std(ft_data.timing), torch.var(ft_data.timing)
ft_data.timing = abs((ft_data.timing-mean)/std)

ft_loader = torch.utils.data.DataLoader(ft_data, batch_size=len(ft_data)//2, num_workers=15, shuffle=True)

# Fine-tune only the final layer on the new dataset
ft_model = TLSTM(2, 4, 64, 50, lr=1e-4, freeze_lstm=True)
ft_model.load_state_dict(model.state_dict(), strict=False)

trainer_ft = pl.Trainer(
    max_epochs=100,
    logger=logger,
    accelerator="auto",
    devices=1
)

trainer_ft.fit(ft_model, ft_loader)
trainer_ft.save_checkpoint("tlstm_ft.ckpt")

# Perform test
trainer_ft.test(dataloaders=test_loader)

# Close logger
wandb.finish()
```

Listing 2: Code for model training and evaluation

# Implementation of T-LSTM Model

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from pytorch_lightning import LightningModule
import numpy as np
import torchmetrics
import itertools
from itertools import chain
import sklearn
from sklearn import metrics
import matplotlib.pyplot as plt

class TLSTM_Layer(nn.Module):

    def __init__(self, dim):
        super().__init__()
        self.dim = dim
        self.W_all = nn.Linear(dim, dim * 4)
        self.emb = nn.Linear(dim, dim * 4)
        self.W_d = nn.Linear(dim, dim)
        self.out = nn.Linear(dim, dim)
        self.h_initial = nn.Parameter(torch.zeros(1, dim)).to(self.emb.weight
    )
        self.c_initial = nn.Parameter(torch.zeros(1, dim)).to(self.emb.weight
    )
        self.norm = nn.LayerNorm(dim)

    def forward(self, signal, delta):
        batch, seq_len, _ = signal.shape
        h = self.h_initial.expand(batch, -1)
        c = self.c_initial.expand(batch, -1)
        signal = self.norm(signal)
        outputs = torch.zeros(*signal.shape).to(signal)

        for step in range(seq_len):
            c_s = torch.tanh(self.W_d(c))
            c_ds = c_s / (1 + torch.log(delta[:, step:step + 1]).expand_as(
    c_s))
            c_t = c - c_s
            c_star = c_t + c_ds
            outs = self.W_all(h) + self.emb(signal[:, step])
            f, i, o, c_candiate = torch.chunk(outs, 4, 1)
            f = torch.sigmoid(f)
            i = torch.sigmoid(i)
            o = torch.sigmoid(o)
            c_candiate = torch.tanh(c_candiate)
            c = f * c_star + i * c_candiate
            h = o * torch.tanh(c)
            if(~torch.isnan(self.out(h)).any()):
                outputs[:, step] = self.out(h)

```

```
49          return outputs
50
51  class FullyConnected(nn.Module):
52      def __init__(self, dim):
53          super().__init__()
54          self.norm = nn.LayerNorm(dim)
55          self.linear1 = nn.Linear(dim, 4*dim)
56          self.linear2 = nn.Linear(4*dim, dim)
57          self.activation = nn.ReLU()
58
59      def forward(self, x):
60          x = self.linear1(self.norm(x))
61          x = self.activation(x)
62          x = self.linear2(x)
63          return x
64
65  class TLSTM(LightningModule):
66      def __init__(self, n_labels=2, n_layer=4, hidden_size=64, initial_segment
        =50, lr=1e-3, freeze_lstm=False):
67          super().__init__()
68          self.n_labels = n_labels
69          self.emb = nn.Embedding(4, hidden_size)
70          self.tlstm_layers = nn.ModuleList([
71              TLSTM_Layer(hidden_size)
72              for _ in range(n_layer)
73          ])
74          self.fc_layers = nn.ModuleList([
75              FullyConnected(hidden_size)
76              for _ in range(n_layer)
77          ])
78          self.out = nn.Linear(hidden_size, n_labels)
79          self.initial_segment = initial_segment
80          self.save_hyperparameters()
81
82          if freeze_lstm:
83              for layer in self.tlstm_layers:
84                  for param in layer.parameters():
85                      param.requires_grad = False
86
87          self.y_train_true = []
88          self.y_train_pred = []
89          self.y_test_true = []
90          self.y_test_pred = []
91
92      def forward(self, signal, timing):
93          signal = self.emb(signal)
94
95          for tlstm, fc in zip(self.tlstm_layers, self.fc_layers):
96              signal = signal + tlstm(signal, timing)
97              signal = signal + fc(signal)
98
99          return self.out(signal).squeeze()
100
```

```
    def training_step(self, batch, batch_idx):
        signal, timing, y = batch
        counter = 0

        out_t = (signal, timing)
        out = self(signal, timing)

        # Calculate loss
        loss = F.cross_entropy(
            out[:, self.initial_segment:].reshape(-1, self.n_labels),
            y[:, self.initial_segment:].reshape(-1)
        )
        self.log('train_loss', loss, prog_bar=True, sync_dist=True)

        # Calculate metrics
        accuracy = (out.argmax(-1) == y)[:, self.initial_segment:].float().
    mean()
        self.log('train_accuracy', accuracy, prog_bar=True, sync_dist=True)

        y_true_tensor = y[:, self.initial_segment:].reshape(-1)
        y_pred_tensor = out[:, self.initial_segment:].reshape(-1, self.
    n_labels).argmax(-1)

        self.y_train_true.append(y_true_tensor.tolist())
        self.y_train_pred.append(y_pred_tensor.tolist())

        TP = ((y_pred_tensor == 1) & (y_true_tensor == 1)).sum().item()
        FP = ((y_pred_tensor == 1) & (y_true_tensor == 0)).sum().item()
        FN = ((y_pred_tensor == 0) & (y_true_tensor == 1)).sum().item()
        TN = ((y_pred_tensor == 0) & (y_true_tensor == 0)).sum().item()

        precision = TP / (TP + FP) if TP + FP > 0 else 0
        recall = TP / (TP + FN) if TP + FN > 0 else 0
        f1 = 2 * (precision * recall) / (precision + recall) if (precision +
    recall) > 0 else 0
        tn_rate = TN / (TN + FP) if TN + FP > 0 else 0

        self.log('train_precision', precision, prog_bar=True, sync_dist=True)
        self.log('train_recall', recall, prog_bar=True, sync_dist=True)
        self.log('train_f1', f1, prog_bar=True, sync_dist=True)
        self.log('train_tn_rate', tn_rate, prog_bar=True, sync_dist=True)

        return loss

    def validation_step(self, batch, batch_idx):
        signal, timing, y = batch
        out = self(signal, timing)
        input = out[:, self.initial_segment:].reshape(-1, self.n_labels)

        # Calculate loss
        loss = F.cross_entropy(
            out[:, self.initial_segment:].reshape(-1, self.n_labels),
            y[:, self.initial_segment:].reshape(-1)
```

```
        )
        self.log('val_loss', loss, sync_dist=True)

        # Calculate metrics
        accuracy = (out.argmax(-1) == y)[:, self.initial_segment:].float().
    mean()
        self.log('val_accuracy', accuracy, sync_dist=True)

        y_true_tensor = y[:, self.initial_segment:].reshape(-1)
        y_pred_tensor = out[:, self.initial_segment:].reshape(-1, self.
    n_labels).argmax(-1)

        TP = ((y_pred_tensor == 1) & (y_true_tensor == 1)).sum().item()
        FP = ((y_pred_tensor == 1) & (y_true_tensor == 0)).sum().item()
        FN = ((y_pred_tensor == 0) & (y_true_tensor == 1)).sum().item()
        TN = ((y_pred_tensor == 0) & (y_true_tensor == 0)).sum().item()

        precision = TP / (TP + FP) if TP + FP > 0 else 0
        recall = TP / (TP + FN) if TP + FN > 0 else 0
        f1 = 2 * (precision * recall) / (precision + recall) if (precision +
    recall) > 0 else 0
        tn_rate = TN / (TN + FP) if TN + FP > 0 else 0

        self.log('val_precision', precision, prog_bar=True, sync_dist=True)
        self.log('val_recall', recall, prog_bar=True, sync_dist=True)
        self.log('val_f1', f1, prog_bar=True, sync_dist=True)
        self.log('val_tn_rate', tn_rate, prog_bar=True, sync_dist=True)

        return loss

    def test_step(self, batch, batch_idx):

        signal, timing, y = batch
        print(len(y))
        out = self(signal, timing)
        input = out[:, self.initial_segment:].reshape(-1, self.n_labels)

        # Calculate loss
        loss = F.cross_entropy(
            out[:, self.initial_segment:].reshape(-1, self.n_labels),
            y[:, self.initial_segment:].reshape(-1)
        )
        self.log('test_loss', loss, sync_dist=True)

        # Calculate metrics
        accuracy = (out.argmax(-1) == y)[:, self.initial_segment:].float().
    mean()
        self.log('test_accuracy', accuracy, sync_dist=True)

        y_true_tensor = y[:, self.initial_segment:].reshape(-1)
        y_pred_tensor = out[:, self.initial_segment:].reshape(-1, self.
    n_labels).argmax(-1)

```

```
        self.y_test_true.append(y_true_tensor.tolist())
        self.y_test_pred.append(y_pred_tensor.tolist())

        TP = ((y_pred_tensor == 1) & (y_true_tensor == 1)).sum().item()
        FP = ((y_pred_tensor == 1) & (y_true_tensor == 0)).sum().item()
        FN = ((y_pred_tensor == 0) & (y_true_tensor == 1)).sum().item()
        TN = ((y_pred_tensor == 0) & (y_true_tensor == 0)).sum().item()

        precision = TP / (TP + FP) if TP + FP > 0 else 0
        recall = TP / (TP + FN) if TP + FN > 0 else 0
        f1 = 2 * (precision * recall) / (precision + recall) if (precision +
    recall) > 0 else 0
        tn_rate = TN / (TN + FP) if TN + FP > 0 else 0

        self.log('test_precision', precision, prog_bar=True, sync_dist=True)
        self.log('test_recall', recall, prog_bar=True, sync_dist=True)
        self.log('test_f1', f1, prog_bar=True, sync_dist=True)
        self.log('test_tn_rate', f1, prog_bar=True, sync_dist=True)

        return loss

    def configure_optimizers(self):
        return torch.optim.Adam(filter(lambda p: p.requires_grad, self.
    parameters()), lr=self.hparams.lr)
```

Listing 3: Code for T-LSTM model