Detection of Micro-Architectural Attacks in Resource-Limited Devices with a Local-Remote Security Mechanism

Nikolaos Foivos Polychronou

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

École doctorale : EEATS - Electronique, Electrotechnique, Automatique, Traitement du Signal (EEATS)
Spécialité : NANO ELECTRONIQUE ET NANO TECHNOLOGIES
Unité de recherche : CEA Grenoble (hors LETI et LITEN)

Détection d'Attaques Microarchitecturales dans des Dispositifs à Ressources Limitées avec un Mécanisme de Sécurité Local-Distant

Detection of Micro-Architectural Attacks in Resource-Limited Devices with a Local-Remote Security Mechanism

Présentée par :

Nikolaos Foivos POLYCHRONOU

Direction de thèse :

Vincent BEROULLE Professeur des Universités, Université Grenoble Alpes	Directeur de thèse
Pierre-Henri THEVENON CEA	Co-encadrant de thèse
Maxime PUYS CEA	Co-encadrant de thèse

Rapporteurs :

Guy GOGNIAT
PROFESSEUR DES UNIVERSITES, University Bretagne Sud
Lilian BOSSUET
PROFESSEUR DES UNIVERSITES, University Jean Monnet

Thèse soutenue publiquement le **16 janvier 2023**, devant le jury composé de :

Vincent BEROULLE PROFESSEUR DES UNIVERSITES, University Grenoble INP-UGA	Directeur de thèse
Guy GOGNIAT PROFESSEUR DES UNIVERSITES, University Bretagne Sud	Rapporteur
Lilian BOSSUET PROFESSEUR DES UNIVERSITES, University Jean Monnet	Rapporteur
Gilles SASSATELLI DIRECTEUR DE RECHERCHE, CNRS/University of Montpellier	Président du jury
Laure GONNORD PROFESSEUR DES UNIVERSITES, University Grenoble INP-UGA	Examinatrice

Invités :

Pierre-henri THEVENON
INGENIEUR DOCTEUR, CEA Grenoble
Maxime PUYS
INGENIEUR DOCTEUR, CEA Grenoble

ACKNOWLEDGEMENT

I would like to thank my parents for their support
whenever it was necessary.

I would also like to thank my supervisors, for their time,
support, and advises, which allowed me to progress in my
thesis and further develop professionally and personally.

I would like to ask the members of the jury and the reviewers
for their presence and evaluation of my work.

ABSTRACT

Internet of Things (IoT) and Industrial Internet of Things (IIoT) have made their way into our daily lives, and with them the dangers associated with their use. These usually inexpensive devices often lack security features or security guarantees because of a desire to reduce development costs. This lack of security comes despite the use of IoT in many critical applications, such as medical devices, industrial sensors, etc. As they remain vulnerable, they are increasingly targeted by attackers, increasing the cost of recovery due to cyberattacks year over year. Traditional security techniques, such as antivirus software, can successfully identify known threats, but they cannot cope with the increasing amount of malware and malware obfuscation techniques.

Malware are typically applications that are covertly inserted to compromise the confidentiality, integrity, or availability of systems. Obfuscation is a technique used by malware authors to modify the behavior and code, but not the actual final goal, of the malicious application to bypass security mechanisms implemented in the system. One of the most well-known IoT malware is Mirai [1], which infected many IoT devices to create a botnet to launch a Distributed Denial-of-Service (DDoS) attack. But due to the complexity of modern systems, a new class of Software Attacks Targeting Hardware Vulnerabilities (SATHV) has emerged. This class of attack, which traditionally targets high-end computers such as desktops, is increasingly being used on IoT devices. This is because these devices are equipped with more complex systems to meet increasing demands on computing and memory resources, such as the emergence of edge-computing and edge-device Machine Learning (ML). This new class of attacks is particularly dangerous because they can extract sensitive information from the system by exploiting hardware vulnerabilities and also remain undetected by traditional security solutions such as antivirus or dynamic analysis of Application Programming Interface (API) calls [2], [3].

This is driving security designers to implement more robust security mechanisms. A hot area of research in recent years has been the use of low-level micro-architectural events to provide us with information about the operation of various hardware components. We are able to measure these micro-architectural events using special registers called Hardware Performance Counters (HPCs), which are part of the Performance

Monitoring Unit (PMU). Using this low-level information, we can profile the operation of the components under normal and malicious conditions. Using ML techniques, researchers are trying to train their systems to recognize the behavior of the system under normal and under attack conditions. But despite the improvements in malware detection and accuracy that these ML-HPCs techniques have shown, these security solutions do not fit the context of IoT devices. IoT are characterized by their low computational and memory resources, and any implementation that imposes significant overhead on the system will not be adopted by IoT vendors. In addition, IoT devices are connected to the Internet and exchange data with a data center, so communication bandwidth is as important as computing resources.

In this thesis, we investigate the ability of ML-HPCs techniques to defend against SATHV and cyber-attacks at runtime. We use low-level information provided by the HPCs to build a robust model against malware and obfuscated malware. We pay particular attention to micro-architectural attacks, which have been a major security threat to systems in recent years due to their ability to leak unprivileged information undetected. We also propose a novel technique to detect malware in low-resource devices such as IoT. The proposed idea aims to accurately detect malicious activities on the edge-device while taking into account all the constraints that modern IoT devices bring, such as overheads in performance, memory, and communication bandwidth. We implement and evaluate the proposed idea and show that it is capable of efficiently detecting malware in modern low-resource devices.

Keywords– IoT, IIoT, security, attacks, hardware vulnerabilities, side-effects, detection, detection mechanisms, edge-computing

ABSTRACT FRANCAIS

Les dispositifs de l'internet des objets (IoT) et de l'IoT industriel (IIoT) sont de plus en plus présents dans le monde, notamment dans applications critiques, telles que les appareils médicaux ou les systèmes industriels. Pour des raisons de coûts et de temps de développements optimisés, la mise en place de sécurité dans ces solutions est souvent mal ou pas prise en compte. Ce manque de sécurité et leur connexion à des réseaux de grande envergure en font une cible de choix pour des attaquants. Les solutions de sécurité traditionnelles, telles que les logiciels antivirus, peuvent identifier avec succès les menaces connues, mais ne peuvent pas faire face au nombre croissant de logiciels malveillants et à leurs techniques d'obscurcissement. Ces logiciels malveillants sont généralement des applications qui sont insérées discrètement pour compromettre la confidentialité, l'intégrité ou la disponibilité d'un système. L'un des malwares IoT les plus connus est le malware Mirai, qui a infecté de nombreux dispositifs IoT pour créer un botnet et lancer une attaque par déni de service distribué (DDoS). La complexité des architectures de processeurs récents a introduit une nouvelle classe d'attaques logicielles ciblant des vulnérabilités matérielles (SATHV). Cette classe d'attaque, qui vise traditionnellement les ordinateurs haut de gamme comme les ordinateurs de bureau, peut également être appliquée aux dispositifs IoT. En effet, ces appareils sont équipés de processeurs plus complexes pour répondre aux demandes croissantes en matière de ressources de calcul et de mémoire, comme l'émergence de l'edge-computing ou du Machine Learning (ML). Ces nouvelles attaques sont particulièrement dangereuses car elles peuvent extraire des informations sensibles du système et ne peuvent être détectées par des solutions de sécurité traditionnelles comme les antivirus ou par l'analyse dynamique des appels système.

Cette situation pousse les concepteurs de sécurité à mettre en œuvre des mécanismes de sécurité plus robustes. Ces dernières années, un état de l'art important s'est constitué sur l'utilisation de données provenant des micro-architectures pour fournir des informations sur le fonctionnement de divers composants matériels afin de détecter des attaques. Ces événements micro-architecturaux peuvent être récupérés à l'aide de registres spéciaux appelés compteurs de performances matérielles (HPC), provenant de l'unité de surveillance des performances (PMU). Grâce à ces informations de bas

niveau, il est possible d'établir un profil de fonctionnement du processeur dans des conditions normales et malveillantes à l'aide de techniques de machine learning (ML). Malgré les améliorations en matière de détection de logiciels malveillants, ces solutions de sécurité ne sont pas toujours adaptées aux contraintes des solutions IoT qui disposent de faibles ressources de calcul et de mémoire et une bande passante de communication restreinte pour la remontée de données.

Dans cette thèse, nous étudions la capacité des techniques ML-HPCs à détecter les attaques de type SATHV en temps réel. Nous utilisons les informations de bas niveau fournies par les HPCs pour construire un modèle robuste contre cette classe de logiciels malicieux. Nous accordons une attention particulière aux attaques micro-architecturales, qui ont constitué une menace majeure pour la sécurité des systèmes ces dernières années en raison de leur capacité à faire fuiter des informations sensibles sans être détectées. Nous proposons également une nouvelle technique pour détecter avec précision ces logiciels malveillants dans des dispositifs embarqués en prenant en compte leurs limitations en terme de performance, de mémoire et de bande passante. Nous mettons en œuvre et évaluons l'idée proposée et montrons qu'elle est capable de détecter efficacement les logiciels malveillants dans les dispositifs modernes à faibles ressources.

Keywords– IoT, IIoT, sécurité, attaques, vulnérabilités matérielles, side-effets, détection, mécanismes de détection, edge-computing

TABLE OF CONTENTS

1

Introduction

1.1 Thesis Context

Internet of Things (IoT) and Industrial Internet of Things (IIoT) devices are growing in popularity and now account for a significant portion of the computing device market. IoT and IIoT range from simple to more complex devices. Their main principle is to interact with the user or the environment and exchange data mainly through a communication interface. IoT and IIoT can intervene in the life of individuals or in the industrial environment. They also handle a large amount of information and data that they either process, store, or transmit over a network. A high percentage of this information is sensitive to the user or industry. A voice controller, for example, processes user requests and consequently patterns that correspond to the user's habits or daily life. A smart lock can store access passwords to a person's home. Taking in mind the development of Cloud and Edge computing, enterprises and common people use these devices to improve the quality of services and life, make proper decisions based on the collected data, track and monitor different aspects of a workload, saving money and resources. Because of their popularity and the amount of sensitive information they can potentially handle, IoT and IIoT are increasingly targeted by attackers. Attackers are taking advantage of the lack of security features in IoT/IIoT to penetrate the system and carry out their malicious activities. Attack campaigns such as Stuxnet [4] and Mirai [1] have demonstrated the high risk of cyberattacks. These campaigns have prompted developers to place more emphasis on IoT/IIoT security.

Despite the developments on the IoT utilization and the data handling, the security of these devices is often not taken into account. Security is considered at a second priority, as IoT vendors try to reduce development costs and time. Taking into account that IoT are low resources devices, connected to the internet, with the lack of security guarantees and robust communication protocols, and further the numerous bugs found in modern software codes, attracts more frequently the attention of attackers. Further, the use of low cost devices without resources dedicated to security plus the availability of the devices for the attacker to apply physical attacks, further increases the attack surface. Attackers benefit from the lack of security features to penetrate the IoT system, extract unprivileged information, use the devices on their own benefit, or perform Distributed Denial-of-Service (DDoS) attacks.

There are two main classes of vulnerabilities in today's systems: Software (SW) and Hardware (HW) vulnerabilities. Software vulnerabilities are flaws or defects in software code that allow attackers to exploit the Operating System (OS) or applications in the system to gain certain privileges. In contrast, hardware vulnerabilities are flaws in the hardware system. The Rowhammer attack [5], for example, exploits flaws that

occur in Dynamic Random Access Memory (DRAM) by repeatedly accessing the same memory locations over a short period of time. Hardware vulnerabilities allow attackers to exploit interactions with the system's electronic components directly, without the need for a software vulnerability, and regardless of the OS.

Traditionally, hardware attacks are used to extract information through leakages such as computing time, power consumption, electromagnetic radiation, or injection of faults into the hardware. Thus, if attackers have physical access to the device, they can employ methods such as laser injection fault attacks [6], [7], the Joint Test Action Group (JTAG) interface [8], [9], or voltage/clock glitch attacks [10], [11] to attack the system.

Due to the complex architectures of modern systems, the attack surface has increased. A new class of Software Attacks Targeting Hardware Vulnerabilities (SATHV) in the various units of the system has emerged. Attacked units include the memory (e.g., cache, DRAM), debugging interfaces, power, and frequency management modules, or solutions used to optimize computation time, such as the out-of-order and speculative execution. Unlike the common hardware attacks mentioned earlier that require physical access, SATHV attacks can be performed remotely. If attackers do not have physical access to the target device, they must access it through communication interfaces such as WiFi or Bluetooth. They can then perform a remote access attack such as clock/voltage fault attacks [12], [13] or a JavaScript Cache Side-Channel Attacks (CacheSCA) [14], [15]. CacheSCA [16], Rowhammer [5], Spectre [17], and Meltdown [18] are among the most serious non-invasive attacks to date that rely on hardware vulnerabilities. SATHV allow attackers to extract sensitive information and induce faults, breaking memory isolation and escalating privileges.

In previous years SATHV were more applicable to high-performance systems, such as desktops and server systems. IoT were mostly low-cost simple devices, equipped with the basic components to perform a series of simple actions. But migrating to the edge-computing era, these low-cost devices start being equipped with more complex processors which allow pre-processing data locally, or allow the implementation of machine learning for voice or image recognition. Since IoT are used more and more and the applicability of SATHV is more possible, attackers apply them to these systems as well. This new class of attacks poses a serious threat to the security of modern devices, and specific countermeasures must be deployed.

It is common to develop patches to address issues related to software vulnerabilities after they have been discovered. In contrast, vulnerabilities in hardware are usually less intensively researched than vulnerabilities in software. This distinction may be justified by the fact that security researchers have limited access to deployed system archi-

tectures. Normally, software tools (e.g., antivirus, firewalls, anti-malware) can ensure that the system is protected against attacks that inject malicious code by exploiting software vulnerabilities. Antivirus software compares information received from the system with known malware information stored in databases. Firewalls can filter information from the Internet. Antimalware systems work like antiviruses, but use other methods such as signature checking, heuristics and sandboxing. Even though these security features are able to detect the software part of an attack, they cannot always detect hardware-related vulnerabilities [2], [3]. In addition, they are unable to detect attacks using obfuscation techniques [19] that modify the software code using techniques such as register reassignment, instruction substitution, etc, to change the signature and bypass signature verification. Thus, if the attacker induces a fault in the hardware, these software tools often do not have access to this information [2].

To protect the system from hardware attacks, various physical protection mechanisms have been developed. These include active shields to protect against invasive probing attacks [20], fault-tolerant redundant hardware systems [21] to prevent fault attacks, dual coils [22], or light sensors [23] to protect against electromagnetic attacks. Based on the extensive research on physical protection measures, we assume that these are already implemented in systems that are physically accessible to prevent physical attacks. Based on this assumption, this work does not consider attacks such as electromagnetic emission analysis, power consumption analysis, leakage power analysis, fault attacks such as electromagnetic, clock, voltage and temperature glitches, laser attacks, cold boot, bus probing, etc.

1.1.1 Introduction to the problem

Unlike physical attacks, it is more challenging to secure the system against SATHV. This is because they use software code to directly target hardware vulnerabilities without using, for example, abnormal Application Programming Interface (API) calls. To secure the system against SATHV, two techniques are proposed: mitigation and detection. Mitigation involves patching the software or hardware to address the vulnerabilities. However, SATHV are difficult to mitigate in software or hardware, either due to the performance/memory cost of patching/updating the software [24] or the applicability only in future CPU architectures. Therefore, detection solutions are preferred. In particular, detection implementations using information from Hardware Performance Counters (HPCs) gain popularity. HPCs are special-core registers that provide information about the operation of the different hardware components in the systems, such as the number of executed instructions, memory accesses, etc. Since SATHV target hard-

ware vulnerabilities, they stress the underlying hardware components in an unusual to the normal operation manner.

However, currently proposed solutions are either proposed for high-performance systems such as desktops or they do not take into account all IoT constraints, making them difficult to be deployed.

IoT Limitations IoT and IIoT are usually limited resource systems. This poses some limitations when considering adding a security solution to protect these devices. We recognize the following limitations:

- Computing power: IoT are usually systems with limited computing resources. This means that these devices are typically not equipped with the high-performance CPU cores available in desktop and server systems. This limits their ability to perform complex operations and/or deliver results quickly. Over the years, more complex CPUs are added, but they are still not comparable to those of desktops.
- Memory: These devices are also equipped with a small amount of memory. Since these devices mostly perform simple tasks, they are only equipped with the necessary memory, which further reduces the cost.
- Communication bandwidth: IoT devices communicate with a control server through a communication interface. With the development of the network, these devices are mostly connected via the Internet to send the extracted data to a control server for further processing and analysis. As the amount of data generated increased, edge-computing was developed to pre-process data locally to filter out data without significant context and transmit only the necessary ones. But with the ever-increasing number of IoT devices, network traffic may soon exceed the capacity of the network. It is important to keep communication bandwidth low through efficient data pre-processing is essential.
- Energy: Many IoT devices are battery-powered. Therefore, it is even more important to conserve as much energy as possible. This requires that the device either remain idle when not in use or operate in low frequencies. In addition, communication over the network should also be reduced, as sending large amounts of data can increase power consumption [25], [26]. Though, minimizing the communication bandwidth can also increase energy efficiency.

These are some of the limitations IoT devices pose when proposing a security mechanism. If one of these is not considered, it can pose limitations in adopting the security solution.

In the State Of The Art (SOTA) we find many detection solutions that can efficiently detect SATHV and/or generic malware. In their efforts to reduce the overheads incurred in the local system, using simple solutions, even if they successfully detect the attacks, they have an increased False Positive Rate (FPR). But when we consider taking appropriate actions in case of system alarms such as reset, returning to a safe state, etc, if these actions are frequent and not due to the existence of true attacks, it can eventually increase the system overheads and make the device unusable. Though, it is important to have high attack coverage and also minimize as possible FPR.

To reduce the FPR while keeping a high attack coverage, SOTA proposed solutions that increase the complexity of the detection models, but this severely increases local overheads. To reduce overheads, when IoT are the target system, remote solutions are proposed, in which resources are *"unlimited"*, but this comes at the cost of increased detection time. This is because the extracted system information to be used for the detection must be transmitted to the remote while waiting it to make a decision. Further, the local system is dependent to the control remote server, unable to run detection offline, and the quantity of data to be transmitted can be big.

Though, considering that IoT pose major limitations, we desire a high attack coverage while keeping the FPR as low as possible, having a quick detection time, increases the complexity of the problem.

1.1.2 Threat Model

In our threat model, we make the following assumptions:

- ***Assumption 1*** is that the attackers can not have physical access to the target system when this is functional. Generally, access to systems in industrial environments is limited, and consumer IoT devices are located in the user's home. If IoT devices can become targets of their own users, we assume there are physical protections in place. Above we mentioned some of the physical protection measures that can be used to protect the system from physical attacks.
- ***Assumption 2*** is that the attackers in our threat model have/had access to the system. During this access, they introduced or implemented a malicious code into the system. This could be done, for example, by loading a corrupted application, inserting a bug into the application code, or the OS. Also, network interfaces, e.g., WiFi, Bluetooth, Ethernet, can be used to gain access. Considering the limited security implementations in IoT and IIoT, this assumption is more than plausible.
- ***Assumption 3*** is that devices equipped with an OS are targeted. Some examples of

OS in our threat model are Linux, embedded Linux, and Android. The attackers' goal is to gain additional privilege levels or extract sensitive information from the system.

- ***Assumption 4*** is that the attackers may already or may not have privileges. Privilege escalation can be achieved through a OS bug. OS Bugs are frequently discovered and allow attackers to gain access to normally protected system information. The attacks investigated could be considered malware and are based on software code that targets a hardware vulnerability in the system microarchitecture.

1.2 Thesis Objectives

In this section, we briefly introduce this thesis objectives, which are the following:

- Study attacks based on Software Attacks Targeting Hardware Vulnerabilities (SATHV).
- Study the side-effects of SATHV on the system.
- Propose a new solution for the detection of SATHV.
- Target to secure resource-limited devices from malware.

As mentioned earlier, SATHV are being used by attackers to attack IoT devices as well, as these systems become more complex. Because SATHV stress the system's hardware components in ways that deviate from normal behavior, they leave traces that we can use to detect them. These traces can be extracted using HPCs because these components measure directly information relative to the hardware components. The difference between the HPC values measured during an attack and during normal operation relates to the side-effects of the attacks. The side-effects are important for several reasons. By selecting a good set of side-effects, we can achieve a high detection rate while having a limited FPR. In addition, selecting an optimal set of side-effects to monitor can enable the implementation of simpler detection mechanisms, which reduces the overhead in the local system.

Current implementations for SOTA attack detection, despite their efficiency, do not consider the limitations of IoT devices such as the performance, memory, communication bandwidth overheads and energy consumption. This can be a problem when applying these solutions on these devices with limited resources. For this reason, we investigate a new security solution that takes into account all the limitations while maintaining a high detection rate and a limited FPR. Finally, since SATHV are only a subset of the available attacks, we investigate the applicability of the proposed solution when the

attack library includes other malware subfamilies.

1.3 Contributions

This thesis focuses on the detection of SATHV in limited resource devices, such as IoT and IIoT. To achieve the goals highlighted in Section 1.2, several milestones were set that led to the contributions summarized here.

- Examine different implementations of SOTA detection and the side-effects that these solutions use to detect SATHV. We show that only theoretical information from other works is not sufficient and that an evaluation should be performed for each attack. Furthermore, in a paper presented at NEWCAS [27], we show that an attacker can attack the system with a small differentiation of the attack and successfully evade implementations with a limited threat model if a security engineer does not consider all attack variants (flush and eviction-based approaches). In addition, in a paper presented at DSD [28], we propose an initial approach for detecting SATHV in an ARMv7 system while considering SATHV using obfuscation techniques.
- Next, we propose a new solution for detecting SATHV in resource-constrained devices such as IoT and IIoT. The new solution is implemented in software and based on a local-remote ML implementation that takes into account all the IoT constraints presented in Section 1.1.1. Since system actions due to false alarms can cause significant overhead in the local system, we aim for an FPR of near 0% while targeting a high attack detection rate.
- Finally, we show that software implementations of the local detection mechanism are vulnerable to software attacks. We demonstrate this using a proof-of-concept software attack that we developed, which shows four cases that the attacker can use to bypass the HPC-based detection mechanism. Furthermore, we show that software detection implementations based on Machine Learning (ML) are vulnerable to Rowhammer attacks [29]. We simulate the reduction in accuracy of a ML-based detection mechanism by inducing a fault in the parameters of the model. To increase the security of the local-remote implementation, we implement the local part in HW. Since the local implementation of SW did not allow us to implement complex ML locally due to performance and energy overheads, we show that this is possible in HW. Finally, we evaluate the proposed implementation on a number of parameters including attack detection rate, FPR, performance, memory, area, and energy consumption.

1.4 Thesis Outline

In addition to this introductory chapter, the rest of this manuscript consists of the following chapters organized as follows:

- Chapter 2: In this chapter, we provide a comprehensive review of the literature on malware detection and SATHV. We discuss in detail the nature of targeted attacks and the detection techniques proposed to solve the detection problem, highlighting their relevant gaps. Finally, we present our methodology steps that allow the reader to become familiar with the terms and techniques used throughout the paper.
- Chapter 3: In this chapter, we analyze the applicability of the theoretical side-effect information extracted from SOTA works for SATHV detection, the development of obfuscated SATHV to circumvent the security mechanism, and the parameters that a designer can use to make this task less feasible. Finally, we analyze and evaluate a detection mechanism that targets SATHV and obfuscated SATHV detection in ARMv7 systems.
- Chapter 4: In this chapter, we propose a novel technique for detecting SATHV in resource-constrained devices. Our approach aims at accurate attack detection, but also takes into account important limitations of modern devices in terms of memory, performance, and communication bandwidth overhead. The proposed idea is implemented in SW. An evaluation with the SOTA in terms of different detection performance metrics and IoT limitations is finally presented.
- Chapter 5: In this chapter, we analyze the security of the SW detection implementations proposed in Chapter 3 and Chapter 4. We demonstrate two attacks that target the side-effect extraction from the HPCs and the ML model, and finally we propose a HW implementation of the local ML to improve security.
- Chapter 6: Finally, in this chapter, we summarize the results of this work and discuss key findings. In the end, we present the limitation of this work and provide perspectives and recommendations for further research.

In the beginning of each chapter, we include a motivation section, which guides the reader behind the problematic driving of our experimentation.

1.5 Publication List

Our Publications

[27] **Polychronou, Nikolaos Foivos**, P.-H. Thevenon, P. Maxime, and V. Beroulle, "Securing iot/iiot from software attacks targeting hardware vulnerabilities", *in 2021 19th IEEE International New Circuits and Systems Conference (NEWCAS)*, IEEE, 2021, pp. 1–4.

[28] **Polychronou, Nikolaos Foivos**, P.-H. Thevenon, M. Puys, and V. Beroulle, "Madman: detection of software attacks targeting hardware vulnerabilities", *in 2021 24th Euromicro Conference on Digital System Design (DSD)*, IEEE, 2021, pp. 355–362.

[119] **Polychronou, Nikolaos-Foivos**, P.-H. Thevenon, M. Puys, and V. Beroulle, "Research for a new solution for the detection of malwares in iot/iiot devices", *Journees C2 2022*, 2022, Available: `https://jc2-2022.inria.fr/files/2022/01/JC2-2022_paper_23.pdf`.

[120] **Polychronou, Nikolaos-Foivos**, P.-H. Thevenon, M. Puys, and V. Beroulle, "A comprehensive survey of attacks without physical access targeting hardware vulnerabilities in iot/iiot devices, and their detection mechanisms", *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, **27**, *1*, pp. 1–35, 2021.

[131] P. Thevenon, S. Riou, D. Tran, *et al.*, "Imrc: integrated monitoring & recovery component, a solution to guarantee the security of embedded systems", *J. Internet Serv. Inf. Secur.*, **12**, *2*, pp. 70–94, 2022. DOI: `10.22667/JISIS.2022.05.31.070`. [Online]. Available: `https://doi.org/10.22667/JISIS.2022.05.31.070`.

[141] **Polychronou, Nikolaos-Foivos**, P.-H. Thevenon, M. Puys, and V. Beroulle, "A system for detecting malwares in a resources constrained device", 2022, patent number DD22030 ST.

1.6 Introduction francais

Contexte de la thèse

Les dispositifs l'internet des objets (IoT) et l'IoT industriel (IIoT) connaissent une popularité croissante et représentent désormais une part importante du marché des dispositifs informatiques. L'IoT et l'IIoT vont de dispositifs simples à des dispositifs plus complexes. Leur principe principal est d'interagir avec l'utilisateur ou l'environnement et d'échanger des données principalement via une interface de communication. L'IoT et l'IIoT peuvent intervenir dans la vie des individus ou dans l'environnement industriel. Ils traitent également une grande quantité d'informations et de données qu'ils traitent, stockent ou transmettent sur un réseau. Un pourcentage élevé de ces informations est sensible pour l'utilisateur ou l'industrie. Un contrôleur vocal, par exemple, traite les demandes des utilisateurs et, par conséquent, les modèles qui correspondent aux habitudes ou à la vie quotidienne de l'utilisateur. Une serrure intelligente peut stocker les mots de passe d'accès au domicile d'une personne. Un actionneur ou une machine automatisée programmée pour produire/concevoir un produit lié à la sécurité. Compte tenu du développement du cloud computing et de l'Edge computing, les entreprises et le grand public utilisent ces dispositifs pour améliorer la qualité des services et de la vie, prendre des décisions appropriées sur la base des données collectées, suivre et surveiller les différents aspects d'une charge de travail, économiser de l'argent et des ressources. En raison de leur popularité et de la quantité d'informations sensibles qu'ils peuvent potentiellement traiter, l'IoT et l'IIoT sont de plus en plus ciblés par les attaquants. Les attaquants profitent de l'absence de fonctions de sécurité dans l'IoT/IIoT pour pénétrer dans le système et mener leurs activités malveillantes. Des campagnes d'attaque telles que Stuxnet [4] et Mirai [1] ont démontré le risque élevé des cyberattaques. Ces campagnes ont incité les développeurs à mettre davantage l'accent sur la sécurité de l'IoT/IIoT.

Malgré les développements sur l'utilisation des IoT et le traitement des données, la sécurité de ces appareils n'est souvent pas prise en compte. La sécurité est considérée comme une deuxième priorité, car les fournisseurs d'IoT essaient de réduire les coûts et le temps de développement. Si l'on tient compte du fait que les IoT sont des dispositifs à faibles ressources, connectés à l'Internet, avec le manque de garanties de sécurité et de protocoles de communication robustes, et en outre les nombreux bogues trouvés dans les codes logiciels modernes, cela attire plus fréquemment l'attention des attaquants. De plus, l'utilisation de dispositifs à faible coût sans ressources dédiées à la sécurité, ainsi que la disponibilité des dispositifs pour que l'attaquant puisse appliquer des at-

taques physiques, augmentent encore la surface d'attaque. Les attaquants profitent de l'absence de fonctions de sécurité pour pénétrer dans le système, extraire des informations non privilégiées, utiliser les dispositifs à leur propre avantage ou réaliser des attaques de type déni de service distribué (DDoS).

Il existe deux grandes catégories de vulnérabilités dans les systèmes actuels : les vulnérabilités SW et HW. Les vulnérabilités logicielles sont des failles ou des défauts du code logiciel qui permettent aux attaquants d'exploiter les Système d'Exploitation (OS) ou les applications du système pour obtenir certains privilèges. En revanche, les vulnérabilités matérielles sont des failles dans le système matériel. L'attaque Rowhammer [5], par exemple, exploite des failles présentes dans le mémoire dynamique synchrone à accès aléatoire (DRAM) en accédant de manière répétée aux mêmes emplacements mémoire sur une courte période. Les vulnérabilités matérielles permettent aux attaquants d'exploiter directement les interactions avec les composants électroniques du système, sans avoir besoin d'une vulnérabilité logicielle, et indépendamment du système d'exploitation.

Traditionnellement, les attaques matérielles sont utilisées pour extraire des informations par le biais de fuites telles que le temps de calcul, la consommation d'énergie, le rayonnement électromagnétique ou l'injection de défauts dans le matériel. Ainsi, si les attaquants ont un accès physique au dispositif, ils peuvent employer des méthodes telles que les attaques par injection de fautes de laser [6], [7], l'interface JTAG [8], [9], ou les attaques par glitch de tension/horloge [10], [11] pour attaquer le système.

En raison des architectures complexes des systèmes modernes, la surface d'attaque a augmenté. Une nouvelle classe d'attaques logicielles ciblant des vulnérabilités matérielles (SATHV) dans les différentes unités du système est apparue. Les unités attaquées comprennent la mémoire (cache e.g., , DRAM), les interfaces de débogage, les modules de gestion de l'alimentation et de la fréquence, ou les solutions utilisées pour optimiser le temps de calcul, comme l'exécution hors ordre et spéculative. Contrairement aux attaques matérielles courantes mentionnées précédemment qui nécessitent un accès physique, les attaques SATHV peuvent être réalisées à distance. Si les attaquants n'ont pas d'accès physique au dispositif cible, ils doivent y accéder par le biais d'interfaces de communication telles que WiFi ou Bluetooth. Ils peuvent alors effectuer une attaque d'accès à distance telle que les attaques par défaut d'horloge/de tension [12], [13] ou une attaque JavaScript attaques de cache par canal auxiliaire (CacheSCA). [14], [15]. CacheSCA [16], Rowhammer [5], Spectre [17], et Meltdown [18] sont parmi les attaques non invasives les plus graves à ce jour qui reposent sur des vulnérabilités matérielles. Les SATHV permettent aux attaquants d'extraire des informations sensi-

bles et de provoquer des failles, en rompant l'isolation de la mémoire et en élevant les privilèges.

Au cours des années précédentes, les SATHV s'appliquaient davantage aux systèmes à haute performance, tels que les ordinateurs de bureau et les systèmes de serveurs. Les IoT étaient principalement des appareils simples et peu coûteux, équipés des composants de base pour effectuer une série d'actions simples. Mais en migrant vers l'ère de l'edge-computing, ces appareils à bas coût commencent à être équipés de processeurs plus complexes qui permettent de prétraiter les données localement, ou de mettre en œuvre l'apprentissage automatique pour la reconnaissance de la voix ou des images. Comme l'IoT est de plus en plus utilisé et que l'applicabilité des SATHV est plus possible, les attaquants les appliquent également à ces systèmes. Cette nouvelle classe d'attaques constitue une menace sérieuse pour la sécurité des appareils modernes, et des contre-mesures spécifiques doivent être déployées.

Il est courant de développer des correctifs pour résoudre les problèmes liés aux vulnérabilités logicielles après leur découverte. En revanche, les vulnérabilités du matériel font généralement l'objet de recherches moins intensives que les vulnérabilités du logiciel. Cette distinction peut être justifiée par le fait que les chercheurs en sécurité ont un accès limité aux architectures des systèmes déployés. Normalement, des outils logiciels (antivirus, pare-feu, anti-malware) peuvent garantir que le système est protégé contre les attaques qui injectent du code malveillant en exploitant les vulnérabilités logicielles. Les logiciels antivirus comparent les informations reçues du système avec celles des logiciels malveillants connus stockées dans des bases de données. Les pare-feu peuvent filtrer les informations provenant d'Internet. Les systèmes antimalware fonctionnent comme les antivirus, mais utilisent d'autres méthodes telles que la vérification des signatures, l'heuristique et le sandboxing. Bien que ces dispositifs de sécurité soient capables de détecter la partie logicielle d'une attaque, ils ne peuvent pas toujours déceler les vulnérabilités liées au matériel [2], [3]. En outre, elles ne peuvent pas détecter les attaques utilisant des techniques d'obscurcissement [19] qui modifient le code logiciel en utilisant des techniques telles que la réaffectation de registres, la substitution d'instructions, etc, pour changer la signature et contourner la vérification de la signature. Ainsi, si l'attaquant induit un défaut dans le matériel, ces outils logiciels n'ont souvent pas accès à cette information [2].

Pour protéger le système contre les attaques matérielles, divers mécanismes de protection physique ont été développés. Il s'agit notamment de boucliers actifs pour se protéger contre les attaques par sondage invasif [20], de systèmes matériels redondants tolérants aux pannes [21] pour empêcher les attaques par panne, de bobines doubles

[22], ou de capteurs de lumière [23] pour se protéger contre les attaques électromagnétiques. Sur la base des recherches approfondies sur les mesures de protection physique, nous supposons que celles-ci sont déjà mises en œuvre dans les systèmes accessibles physiquement pour prévenir les attaques physiques. Sur la base de cette hypothèse, ce travail ne prend pas en compte les attaques telles que l'analyse des émissions électromagnétiques, l'analyse de la consommation d'énergie, l'analyse de la puissance de fuite, les attaques par défaut telles que les défaillances électromagnétiques, d'horloge, de tension et de température, les attaques laser, le démarrage à froid, le sondage de bus, etc.

Introduction au problème

Contrairement aux attaques physiques, il est plus difficile de sécuriser le système contre les SATHV. En effet, elles utilisent du code logiciel pour cibler directement les vulnérabilités matérielles sans utiliser, par exemple, des appels Interface de Programmation d'Applications (API) anormaux. Pour sécuriser le système contre les SATHV, deux techniques sont proposées : l'atténuation et la détection. L'atténuation consiste à appliquer des correctifs au logiciel ou au matériel pour corriger les vulnérabilités. Cependant, les SATHV sont difficiles à atténuer dans le logiciel ou le matériel, soit en raison du coût en performance/mémoire des correctifs/mises à jour du logiciel, soit en raison de leur applicabilité uniquement dans les futures architectures de CPU. Par conséquent, les solutions de détection sont préférables. En particulier, les implémentations de détection utilisant les informations des compteurs de performance du matériel (HPCs) gagnent en popularité. Les HPC sont des registres spéciaux qui fournissent des informations sur le fonctionnement des différents composants matériels des systèmes, comme le nombre d'instructions exécutées, les accès à la mémoire, etc. Comme les SATHV ciblent les vulnérabilités matérielles, ils sollicitent les composants matériels sous-jacents d'une manière inhabituelle par rapport à leur fonctionnement normal.

Cependant, les solutions actuellement proposées sont soit destinées aux systèmes à haute performance tels que les ordinateurs de bureau, soit elles ne prennent pas en compte toutes les contraintes de l'IoT, ce qui rend leur déploiement difficile.

IoT Limitations L'IoT et l'IIoT sont généralement des systèmes à ressources limitées. Cela pose certaines limites lorsqu'on envisage d'ajouter une solution de sécurité pour protéger ces appareils. Nous reconnaissons les limitations suivantes :

- Puissance de calcul : les IoT sont généralement des systèmes aux ressources informatiques limitées. Cela signifie que ces appareils ne sont généralement pas

équipés des cœurs d'unité centrale haute performance disponibles dans les systèmes de bureau et de serveur. Cela limite leur capacité à effectuer des opérations complexes et/ou à fournir des résultats rapidement. Au fil des années, des processeurs plus complexes sont ajoutés, mais ils ne sont toujours pas comparables à ceux des ordinateurs de bureau.

- Mémoire de l'appareil : Ces appareils sont également équipés d'une petite quantité de mémoire. Étant donné que ces appareils effectuent principalement des tâches simples, ils ne sont équipés que de la mémoire nécessaire, ce qui réduit encore le coût.

- Bande passante de communication : les appareils IoT communiquent avec un serveur de contrôle via une interface de communication. Avec le développement du réseau, ces appareils sont le plus souvent connectés via Internet pour envoyer les données extraites à un serveur de contrôle pour un traitement et une analyse plus poussés. Avec l'augmentation de la quantité de données générées, l'edge-computing a été développé pour prétraiter les données localement afin de filtrer les données sans contexte significatif et de transmettre uniquement celles qui sont nécessaires. Mais avec le nombre toujours croissant de dispositifs IoT, le trafic réseau peut bientôt dépasser la capacité du réseau. Il est important de maintenir une faible bande passante de communication grâce à un prétraitement efficace des données est essentiel.

- L'énergie : De nombreux appareils IoT sont alimentés par des batteries. Il est donc d'autant plus important de conserver autant d'énergie que possible. Pour cela, l'appareil doit soit rester inactif lorsqu'il n'est pas utilisé, soit fonctionner en basse fréquence. En outre, la communication sur le réseau doit également être réduite, car l'envoi de grandes quantités de données peut augmenter la consommation d'énergie [25], [26]. Bien que la minimisation de la bande passante de communication puisse également augmenter l'efficacité énergétique.

Ce sont là quelques-unes des limites que posent les dispositifs IoT lorsqu'on propose un mécanisme de sécurité. Si l'une d'entre elles n'est pas prise en compte, l'adoption de la solution de sécurité peut être limitée.

Dans le l'Etat de l'Art (EdA), nous trouvons de nombreuses solutions de détection qui peuvent détecter efficacement les SATHV et/ou les malwares génériques. Dans leurs efforts pour réduire les frais généraux encourus dans le système local, en utilisant des solutions simples, même s'ils réussissent à détecter les attaques, ils ont une taux de faux positifs (FPR) accrue. Mais si l'on considère la prise de mesures appropriées en cas

d'alarmes système telles que la réinitialisation, le retour à un état sûr, etc, si ces actions sont fréquentes et non dues à l'existence de véritables attaques, cela peut finalement augmenter les frais généraux du système et rendre le dispositif inutilisable. Cependant, il est important d'avoir une couverture élevée des attaques et de minimiser autant que possible le FPR.

Pour réduire le FPR tout en gardant une couverture élevée des attaques, EdA a proposé des solutions qui augmentent la complexité des modèles de détection, mais cela augmente sévèrement les frais généraux locaux. Pour réduire les frais généraux, lorsque l'IoT est le système cible, des solutions à distance sont proposées, dans lesquelles les ressources sont "illimitées", mais cela se fait au prix d'une augmentation du temps de détection. En effet, les informations extraites du système qui seront utilisées pour la détection doivent être transmises au système distant en attendant qu'il prenne une décision. De plus, le système local est dépendant du serveur distant de contrôle, incapable d'exécuter la détection hors ligne, et la quantité de données à transmettre peut être importante.

Bien que, considérant que l'IoT pose des limitations majeures, nous souhaitons une couverture d'attaque élevée tout en gardant le FPR aussi bas que possible, avoir un temps de détection rapide, augmente la complexité du problème.

Modèle de menace

Dans notre modèle de menace, nous faisons les hypothèses suivantes :

- L'hypothèse 1 est que les attaquants ne peuvent pas avoir un accès physique au système cible lorsque celui-ci est fonctionnel. En général, l'accès aux systèmes dans les environnements industriels est limité, et les dispositifs IoT grand public sont situés au domicile de l'utilisateur. Si les appareils IoT peuvent devenir des cibles de leurs propres utilisateurs, nous supposons que des protections physiques sont en place. Nous avons mentionné ci-dessus certaines des mesures de protection physique qui peuvent être utilisées pour protéger le système contre les attaques physiques.
- L'hypothèse 2 est que les attaquants de notre modèle de menace ont ou avaient accès au système. Au cours de cet accès, ils ont introduit ou implémenté un code malveillant dans le système. Cela peut se faire, par exemple, en chargeant une application corrompue, en insérant un bogue dans le code de l'application ou dans le système d'exploitation. Les interfaces réseau (WiFi, Bluetooth, Ethernet, etc.) peuvent également être utilisées pour obtenir un accès. Compte tenu des

implémentations de sécurité limitées dans l'IoT et l'IIoT, cette hypothèse est plus que plausible.

- L'hypothèse 3 est que les appareils équipés d'un système d'exploitation sont ciblés. Dans notre modèle de menace, Linux, Linux embarqué et Android sont des exemples de systèmes d'exploitation. L'objectif des attaquants est d'obtenir des niveaux de privilèges supplémentaires ou d'extraire des informations sensibles du système.

- L'hypothèse 4 est que les attaquants peuvent déjà ou ne pas avoir de privilèges. L'escalade des privilèges peut être réalisée par le biais d'un bogue de système d'exploitation. Les bogues d'OS sont fréquemment découverts et permettent aux attaquants d'accéder à des informations système normalement protégées. Les attaques étudiées peuvent être considérées comme des logiciels malveillants et sont basées sur un code logiciel qui cible une vulnérabilité matérielle dans la microarchitecture du système.

Objectifs de la thèse

Dans cette section, nous présentons brièvement les objectifs de cette thèse, qui sont les suivants :

- Étudier les attaques basées sur attaques logicielles visant les vulnérabilités matérielles (SATHV).

- Étudier les effets secondaires de SATHV sur le système.

- Proposer une nouvelle solution pour la détection de SATHV.

- Objectif : sécuriser les appareils à ressources limitées contre les logiciels malveillants.

Comme nous l'avons mentionné précédemment, les SATHV sont également utilisés par les attaquants pour attaquer les appareils IoT, car ces systèmes deviennent plus complexes. Comme les SATHV sollicitent les composants matériels du système d'une manière qui s'écarte du comportement normal, ils laissent des traces que nous pouvons utiliser pour les détecter. Ces traces peuvent être extraites à l'aide des HPC car ces composants mesurent directement les informations relatives aux composants matériels. La différence entre les valeurs HPC mesurées pendant une attaque et pendant le fonctionnement normal concerne les effets secondaires des attaques. Ces effets secondaires sont importants pour plusieurs raisons. En sélectionnant un bon ensemble d'effets sec-

ondaires, nous pouvons obtenir un taux de détection élevé tout en ayant un TFP limité. En outre, la sélection d'un ensemble optimal d'effets secondaires à surveiller peut permettre la mise en œuvre de mécanismes de détection plus simples, ce qui réduit les frais généraux dans le système local.

Les implémentations actuelles pour la détection des attaques EdA, malgré leur efficacité, ne tiennent pas compte des limites des dispositifs IoT, telles que les performances, la mémoire, les frais généraux de bande passante de communication et la consommation d'énergie. Cela peut être un problème lors de l'application de ces solutions sur ces appareils aux ressources limitées. Pour cette raison, nous étudions une nouvelle solution de sécurité qui prend en compte toutes les limitations tout en maintenant un taux de détection élevé et un FPR limité. Enfin, comme les SATHV ne sont qu'un sous-ensemble des attaques disponibles, nous étudions l'applicabilité de la solution proposée lorsque la bibliothèque d'attaques comprend d'autres sous-familles de logiciels malveillants.

Contributions

Cette thèse se concentre sur la détection de SATHV dans les dispositifs à ressources limitées, tels que l'IoT et l'IIoT. Pour atteindre les objectifs mis en évidence dans Section 1.2, plusieurs jalons ont été posés qui ont conduit aux contributions résumées ici.

- Examiner différentes implémentations de la détection de EdA et les effets secondaires que ces solutions utilisent pour détecter SATHV. Nous montrons que les seules informations théoriques issues d'autres travaux ne sont pas suffisantes et qu'une évaluation doit être effectuée pour chaque attaque. De plus, dans un article présenté à NEWCAS [27], nous montrons qu'un attaquant peut attaquer le système avec une petite différenciation de l'attaque et réussir à échapper aux implémentations avec un modèle de menace limité si un ingénieur de sécurité ne considère pas toutes les variantes d'attaque (approches basées sur le flush et l'éviction). De plus, dans un article présenté à DSD [28], nous proposons une première approche pour détecter le SATHV dans un système ARMv7 tout en considérant le SATHV utilisant des techniques d'obfuscation.
- Ensuite, nous proposons une nouvelle solution pour détecter les SATHV dans les dispositifs à ressources limitées tels que IoT et IIoT. La nouvelle solution est implémentée en logiciel et basée sur une implémentation ML locale-distante qui prend en compte toutes les contraintes IoT présentées dans Section 1.1.1. Étant donné que les actions du système dues aux fausses alarmes peuvent entraîner une surcharge importante dans le système local, nous visons un FPR proche de

0% tout en ciblant un taux élevé de détection des attaques.

- Enfin, nous montrons que les implémentations logicielles du mécanisme de détection local sont vulnérables aux attaques logicielles. Nous le démontrons à l'aide d'une preuve de concept d'attaque logicielle que nous avons développée, qui présente quatre cas que l'attaquant peut utiliser pour contourner le mécanisme de détection basé sur le HPC. De plus, nous montrons que les implémentations de détection logicielle basées sur apprentissage automatique (ML) sont vulnérables aux attaques Rowhammer [29]. Nous simulons la réduction de la précision d'un mécanisme de détection basé sur ML en induisant une faute dans les paramètres du modèle. Pour augmenter la sécurité de l'implémentation locale-distante, nous implémentons la partie locale dans HW. Comme l'implémentation locale de SW ne nous permettait pas d'implémenter localement des ML complexes en raison des surcharges de performance et d'énergie, nous montrons que cela est possible en HW. Enfin, nous évaluons l'implémentation proposée sur un certain nombre de paramètres, notamment le taux de détection des attaques, le FPR, les performances, la mémoire, la surface et la consommation d'énergie.

Cadre de la thèse

En plus de ce chapitre d'introduction, le reste de ce manuscrit se compose des chapitres suivants organisés comme suit :

- Chapter 2 : Dans ce chapitre, nous fournissons une revue complète de la littérature sur la détection des logiciels malveillants et le SATHV. Nous discutons en détail de la nature des attaques ciblées et des techniques de détection proposées pour résoudre le problème de la détection, en soulignant leurs lacunes pertinentes. Enfin, nous présentons les étapes de notre méthodologie qui permettent au lecteur de se familiariser avec les termes et les techniques utilisés tout au long de l'article.
- Chapter 3 : Dans ce chapitre, nous analysons l'applicabilité des informations théoriques sur les effets secondaires extraites des travaux de EdA pour la détection des SATHV, le développement de SATHV obfusqués pour contourner le mécanisme de sécurité, et les paramètres qu'un concepteur peut utiliser pour rendre cette tâche moins réalisable. Enfin, nous analysons et évaluons un mécanisme de détection qui cible le SATHV et la détection du SATHV obfusqué dans les systèmes ARMv7.
- Chapter 4 : Dans ce chapitre, nous proposons une nouvelle technique de détection des SATHV dans les dispositifs à ressources limitées. Notre approche vise une détection précise des attaques, mais prend également en compte les limitations

importantes des dispositifs modernes en termes de mémoire, de performances et de bande passante de communication. L'idée proposée est mise en œuvre dans un logiciel. Une évaluation avec le EdA en termes de différentes métriques de performance de détection et de limitations de l'IoT est finalement présentée.

- Chapter 5 : Dans ce chapitre, nous analysons la sécurité des implémentations de détection SW proposées dans Chapter 3 et Chapter 4. Nous démontrons deux attaques qui ciblent l'extraction d'effets secondaires des HPC et le modèle ML, et enfin nous proposons une implémentation HW du ML local pour améliorer la sécurité.
- Chapter 6 : Enfin, dans ce chapitre, nous résumons les résultats de ce travail et discutons des principales conclusions. Enfin, nous présentons les limites de ce travail et fournissons des perspectives et des recommandations pour de futures recherches.

Au début de chaque chapitre, nous incluons une section de motivation, qui guide le lecteur derrière la problématique de notre expérimentation.

Contexte et état de l'art

Ce chapitre fournit les connaissances de base nécessaires pour comprendre les différentes sous-familles de logiciels malveillants, la gravité des attaques micro-architecturales et les méthodes proposées pour sécuriser le système. Tout d'abord, le chapitre définit les logiciels malveillants et explique les étapes et les objectifs des attaques micro-architecturales. Ensuite, nous analysons les méthodes proposées dans l'état de l'art pour détecter les malwares. L'apprentissage automatique étant un outil de plus en plus utilisé dans la sécurité, nous analysons ses avantages et présentons les différents modèles utilisés dans ce travail. Enfin, le chapitre se termine par une discussion sur les informations obtenues à partir de l'état de l'art.

Attaques de Logiciels Malveillants et de Logiciels Ciblant des Vulnérabilités Matérielles

Au cours des dernières décennies, le monde a connu une transition rapide vers l'ère numérique, où la quasi-totalité de nos données sont stockées et traitées par des systèmes informatiques. Cette accélération de la transition vers le monde numérique a été rendue possible par le développement rapide des ordinateurs, des mémoires, des communications, des protocoles de sécurité, etc. L'industrie a fait des progrès en matière de matériel et de logiciels, développant des systèmes de plus en plus compliqués pour

répondre aux demandes croissantes. Cependant, le passage à l'ère numérique a entraîné une augmentation des attaques contre les systèmes informatiques. Les attaquants effectuent des opérations malveillantes qui perturbent le fonctionnement normal des systèmes informatiques. Nous appelons ces applications malveillantes des malwares. Il existe de nombreuses définitions du terme "malware", par exemple :

- Programme inséré dans un système, généralement de manière clandestine, dans le but de compromettre la confidentialité, l'intégrité ou la disponibilité des données, des applications ou du système d'exploitation de la victime, ou de l'importuner ou de la perturber d'une autre manière [30].

- Un logiciel malveillant est un type de programme informatique conçu pour infecter l'ordinateur d'un utilisateur légitime et lui infliger des dommages de plusieurs manières. Les logiciels malveillants peuvent infecter les ordinateurs et les appareils de plusieurs façons et se présentent sous différentes formes, dont les virus, les vers, les chevaux de Troie, les logiciels espions, etc [31].

- Malware est une forme abrégée de "logiciel malveillant". Il s'agit d'un logiciel spécifiquement conçu pour accéder à un ordinateur ou l'endommager, généralement à l'insu de son propriétaire [32].

Toutes les définitions ci-dessus sont très similaires et comprennent au moins la malveillance ou la nocivité et la capacité d'effectuer des actions sans la permission ou la conscience de l'utilisateur. À cette fin, Or-Meir et al. [33] proposent une définition différente des logiciels malveillants qui ne se concentre pas sur les intentions de l'attaquant, car il est impossible de déterminer les intentions derrière un code binaire inconnu. Cependant, ils proposent la définition suivante :

Un logiciel malveillant est un code exécuté sur un système informatique dont les administrateurs du système ignorent la présence ou le comportement ; si les administrateurs du système avaient connaissance du code et de son comportement, ils n'en autoriseraient pas l'exécution. Les logiciels malveillants compromettent la confidentialité, l'intégrité ou la disponibilité du système en exploitant les vulnérabilités existantes d'un système ou en en créant de nouvelles.

La définition ci-dessus des logiciels malveillants fait référence au point de vue de l'administrateur système et aux trois principes de sécurité : confidentialité, intégrité et disponibilité, qu'un logiciel malveillant tente de compromettre. Nous adoptons la définition susmentionnée d'Or-Meir et al. et nous y ajoutons les éléments suivants :

Définition étendue des logiciels malveillants

Malware est l'abréviation d'actions malveillantes logicielles ou matérielles dont les administrateurs système ignorent la présence ou le comportement ; si les administrateurs système étaient au courant du code et de son comportement, ils n'en autoriseraient pas l'exécution. Les logiciels malveillants compromettent la confidentialité, l'intégrité ou la disponibilité du système en exploitant les vulnérabilités existantes d'un système ou en en créant de nouvelles.

La définition étendue ci-dessus inclut le code malveillant lié au matériel qui tente de compromettre la confidentialité, l'intégrité ou la disponibilité des systèmes. C'est le cas des chevaux de Troie matériels, que de nombreux travaux SOTA qualifient de malwares [34], [35].

Dans les sections suivantes, nous distinguerons les attaques en fonction de leur mode de réalisation, c'est-à-dire logiciel ou matériel.

Attaques Logicielles

Les attaques visent à exploiter une vulnérabilité pour compromettre la confidentialité, l'intégrité ou la disponibilité du système. Une vulnérabilité peut être une faille ou un bug qui permet aux attaquants d'effectuer leurs actions malveillantes dans le système cible. À mesure que le code logiciel devient plus complexe, les vulnérabilités dans le code logiciel se multiplient. Selon le livre Code Complete de Steve McConnell [36], la moyenne de l'industrie est d'environ 15 à 50 erreurs pour 1000 lignes de code livrées. L'augmentation des erreurs dans les logiciels accroît la surface d'attaque que les attaquants peuvent utiliser pour pénétrer dans le système. L'une des vulnérabilités logicielles les plus visibles est le dépassement de tampon, qui permet aux attaquants d'écrire plus de données dans un tampon d'entrée que ne le permet ce dernier. Un attaquant peut exploiter cette situation pour placer une charge utile malveillante dans le tampon débordé, qui est ensuite exécutée. Un débordement de tas est un type de débordement de tampon qui se produit dans la zone de tas, où l'attaque est réalisée en corrompant les structures de tas telles que les pointeurs de listes liées. Parmi les autres vulnérabilités logicielles, citons l'injection SQL, les vulnérabilités des bibliothèques et API tierces, le contrôle d'accès défectueux, l'injection de code, les échecs d'identification et d'authentification, les échecs d'intégrité des logiciels et des données, etc.

Lorsque les attaquants découvrent une vulnérabilité logicielle dans un système cible, ils effectuent leurs actions malveillantes. La vulnérabilité peut être connue du public ou être une nouvelle vulnérabilité qui n'a jamais été vue auparavant, que nous appelons vulnérabilité "zero-day". Après avoir exploité une vulnérabilité logicielle, les attaquants peuvent voler ou manipuler des données sensibles, intégrer le système cible dans un botnet, installer une porte dérobée ou d'autres types de logiciels malveillants tels qu'un rootkit. En outre, l'attaquant peut exploiter un hôte vulnérable pour pénétrer d'autres hôtes sur le même réseau. En revanche, les attaques visant les vulnérabilités logicielles permettent d'effectuer d'autres actions malveillantes après avoir pris le contrôle du système ciblé.

Attaques matérielles

Les attaques matérielles exploitent les vulnérabilités des composants matériels eux-mêmes. Comme les composants matériels sont responsables de l'exécution du code logiciel, le comportement du matériel peut révéler des informations sur les informations traitées. En outre, la modification du comportement du matériel peut également modifier le comportement du logiciel. Les attaquants ont trouvé des moyens d'exploiter cette interaction pour extraire des informations sensibles du système, comme des clés cryptographiques, ou modifier le flux de contrôle normal d'un code logiciel. Ces attaques matérielles sont l'analyse de la puissance différentielle et de la corrélation, l'horloge, les attaques par défaut de tension et les attaques par défaut de laser. L'analyse différentielle et par corrélation de la puissance des canaux latéraux extrait les mesures des canaux latéraux en utilisant des traces de la consommation d'énergie ou du rayonnement électromagnétique, et tente de les corréler à une variable sensible traitée par le système ciblé. Les attaques d'horloge, de tension ou de défaut laser induisent un défaut pendant l'exécution normale du système ciblé afin d'extraire des informations en utilisant l'analyse différentielle des défauts, ou de faire alterner l'exécution normale avec une exécution inattendue. D'autres types d'attaques matérielles incluent les attaques utilisant l'interface de débogage JTAG [9]. Les attaques susmentionnées ciblent les vulnérabilités du matériel et nécessitent un accès physique au dispositif. Cela limite la capacité de l'attaquant à exploiter les vulnérabilités du système dans les cas où l'accès physique n'est pas possible. En revanche, les attaques logicielles peuvent être réalisées à distance, ce qui permet à l'attaquant d'effectuer les actions malveillantes depuis presque n'importe où.

Les attaques matérielles avec accès physique aux dispositifs ne font pas partie de ce travail. Nous supposons que les attaquants n'ont pas d'accès physique au système cible.

En général, l'accès aux systèmes dans les environnements industriels est limité, et les dispositifs IoT grand public sont situés au domicile de l'utilisateur. En outre, si les dispositifs IoT peuvent devenir les cibles de leurs propres utilisateurs, nous supposons que des protections physiques sont en place. Des exemples de telles protections incluent un élément sécurisé intégré ou la détection des tentatives de démantèlement de l'appareil. Ce sont là quelques-unes des protections physiques qui peuvent être utilisées pour protéger le système contre les attaques physiques. Nous avons mentionné les attaques matérielles nécessitant un accès physique car elles nous aident à démontrer la gravité d'une nouvelle classe d'attaques que nous présenterons dans la prochaine Section 2.1.3.

Attaques logicielles visant les vulnérabilités matérielles

En raison des exigences croissantes des systèmes modernes en matière de puissance de calcul, de mémoire et d'utilitaires, les architectures modernes deviennent plus complexes. De plus en plus de composants dotés de capacités complexes sont mis en œuvre dans le matériel pour répondre aux demandes croissantes. Cela conduit à une surface d'attaque plus grande. Les attaquants explorent le matériel pour trouver des vulnérabilités qui leur permettent d'explorer le système. Ainsi, une nouvelle classe de SATHV dans les différentes unités du système est apparue. Les unités matérielles ciblées comprennent la mémoire (par exemple, cache, DRAM), les modules de gestion de l'énergie et de la fréquence, les interfaces de débogage ou les composants proposés pour optimiser le temps de calcul, comme l'exécution hors ordre et spéculative.

Contrairement aux attaques matérielles courantes mentionnées dans la Section 2.1.2, où l'attaquant doit avoir un accès physique au système cible, le SATHV peut être réalisé à distance. Puisque les attaquants n'ont pas d'accès physique à leurs systèmes cibles, ils y accèdent par le biais d'une interface de communication telle que Wi-Fi, Bluetooth, par exemple. Cela leur permet d'effectuer une attaque d'accès à distance telle que les attaques par défaut d'horloge/tension [12], [13] ou un CacheSCA JavaScript [14], [15]. CacheSCA [16], Rowhammer [37], Spectre [17], et Meltdown [18] comptent parmi les attaques non invasives les plus graves à ce jour, qui reposent sur des vulnérabilités matérielles et peuvent être exécutées à distance. Dans les sous-sections suivantes, nous expliquerons comment ces attaques fonctionnent et quelles vulnérabilités elles ciblent.

Attaques de Cache par Canal Auxiliaire

Architecture des caches Les CacheSCA sont un ensemble d'attaques qui ciblent la mémoire cache. La mémoire cache est un composant matériel dont le but est de réduire

le temps moyen et le coût énergétique de l'accès aux données stockées dans la mémoire principale. Les caches sont plus petits en taille que la mémoire principale, plus rapides et situés plus près de l'unité centrale. Leur but est de fournir un accès plus rapide aux données ou aux instructions qui sont fréquemment utilisées pendant l'exécution du programme. Cela permet de gagner un temps précieux en accélérant l'exécution. Les mémoires caches sont souvent organisées selon une hiérarchie allant du plus proche du CPU au plus proche de la mémoire principale. Plus on est proche de l'unité centrale, plus les caches sont petits et plus ils sont grands à mesure que l'on se rapproche de la mémoire principale. Les caches de niveau 1 (L1) sont également divisés en cache de données L1 (L1D) et cache d'instructions L1 (L1I). Le cache d'instructions est chargé de stocker les instructions exécutées, tandis que le cache de données stocke les données à traiter.

En outre, il existe des caches TLB (Translation Look-aside Buffer). Le cache TLB est une mémoire qui stocke les traductions des adresses virtuelles en adresses physiques. Le TLB fait partie de l'unité de gestion de la mémoire (MMU), qui est responsable de la gestion de la mémoire virtuelle. La MMU est un composant matériel qui gère toutes les opérations de mémoire et de cache associées au CPU, dans le but de séparer et de protéger efficacement la mémoire utilisée entre les processus. La TLB contient les traductions les plus fréquemment utilisées des adresses virtuelles aux adresses physiques. Ainsi, lorsque le CPU demande à accéder à une adresse virtuelle d'instruction ou de données, la TLB traduit cette demande en une adresse physique que le matériel peut utiliser pour rechercher les données demandées dans les caches d'instructions ou de données. Les caches TLB peuvent également être divisés en TLB d'instructions (iTLB) et TLB de données (dTLB), et il peut également y avoir plus d'un niveau de caches TLB.

2

Background and SOTA

This chapter provides the background knowledge necessary to understand the various subfamilies of malware, the severity of microarchitectural attacks and the proposed methods for securing the system. First, the chapter defines malware and explains the steps and objectives of microarchitectural attacks. Then, we analyze the methods proposed in the state of the art to detect malware. Since machine learning is a tool that is increasingly used in the security, we analyze its benefits and present the different models used in this work. Finally, the chapter concludes with a discussion of the information obtained from the state of the art.

2.1 Malware and Software Attacks Targeting Hardware Vulnerabilities

In recent decades, the world has been rapidly transitioning to the digital era, where almost all of our data is stored and processed by computer systems. This acceleration of the transition to the digital world was made possible by the rapid development of computers, memories, communications, security protocols, etc. Industry made advances in hardware and software, developing increasingly complicated systems to meet growing demands. However, the transition to the digital age led to an increase in attacks on computer systems. Attackers perform malicious operations that disrupt the normal operation of computer systems. We refer to these malicious applications as malwares. There are many definitions for the term malware, for example:

- *A program that is inserted into a system, usually covertly, with the intent of compromising the confidentiality, integrity, or availability of the victim's data, applications, or operating system or of otherwise annoying or disrupting the victim* ***NIST***.[30]
- *Malware, short for "malicious software," refers to a type of computer program designed to infect a legitimate user's computer and inflict harm on it in multiple ways. Malware can infect computers and devices in several ways and comes in a number of forms, just a few of which include viruses, worms, Trojans, spyware and more* ***Kaspersky***[31].
- *Malware is an abbreviated form of "malicious software." This is software that is specifically designed to gain access to or damage a computer, usually without the knowledge of the owner* ***Norton***[32].

All of the above definitions are very similar and include at least the maliciousness or harmfulness and the ability to perform actions without the user's permission or awareness. To this end, Or-Meir et al. [33] proposes a different definition of malware that does not focus on the intentions of the attacker, as it is impossible to determine the intentions behind an unknown binary code. Though, they propose the following definition:

Malware is code running on a computer system whose presence or behavior the system administrators are unaware of; were the system administrators aware of the code and its behavior, they would not permit it to run. Malware compromises the confidentiality, integrity or the availability of the system by exploiting existing vulnerabilities in a system or by creating new ones.

The above definition of malware refers to the system administrator's point of view and to the three principles of security: confidentiality, integrity, and availability, which a malware attempts to compromise. We embrace the aforementioned definition by Or-

Meir et al. and we extended adding the following:

Extended Malware Definition

Malware is the abbreviation of software or hardware malicious actions whose presence or behavior the system administrators are unaware of; were the system administrators aware of the code and its behavior, they would not permit it to run. Malware compromises the confidentiality, integrity or the availability of the system by exploiting existing vulnerabilities in a system or by creating new ones.

The above extended definition includes hardware related malicious code that tries to compromise systems confidentiality, integrity or the availability. Such cases are the case hardware Trojans, which numerous SOTA works refer to as malwares [34], [35].

In the following sections, we will distinguish attacks according to the means of performing them i.e., software or hardware.

2.1.1 Software Attacks

Attacks aim to exploit a vulnerability to compromise the confidentiality, integrity, or availability of the system. A vulnerability can be a flaw or bug that allows attackers to perform their malicious actions in the target system. As software code becomes more complex, vulnerabilities in the software code multiply. According to Steve McConnell's book Code Complete [36], the industry average is about 15-50 errors per 1000 lines of delivered code. Increasing errors in software increase the attack surface that attackers can use to penetrate the system. One of the most noticeable software vulnerabilities are buffer overflows, which allow attackers to write more data to an input buffer than the buffer allows. An attacker can exploit this to place malicious payload in the overflowed buffer, which is then executed. A heap overflow is a type of buffer overflow that occurs in the heap area, where the attack is performed by corrupting the heap structures such as linked list pointers. Other software vulnerabilities include SQL injection, third-party library and API vulnerabilities, faulty access control, code injection, identification and authentication failures, software and data integrity failures, etc.

When attackers discover a software vulnerability in a target system, they perform their malicious actions. The vulnerability may be publicly known or it may be a new vulnerability that has never been seen before which we refer to as zero-day vulnerabilities. After

attackers exploit a software vulnerability, they can steal or manipulate sensitive data, join the target system in a botnet, install a backdoor, or install other types of malware such as a rootkit. In addition, the attacker can exploit a vulnerable host to penetrate other hosts on the same network. Though, attacks targeting software vulnerabilities are the means to perform other malicious actions after gaining control of the targeted system.

2.1.2 Hardware Attacks

Hardware attacks exploit vulnerabilities of the hardware components itself. Since the hardware components are responsible for executing the software code, the hardware behavior can reveal information about the information processed. Further, modifying the behavior of the hardware can also change the software behavior. Attackers found ways to exploit this interaction to extract sensitive information from the system, such as cryptographic keys, or change the normal control flow of a software code. Such hardware attacks are differential and correlation power analysis, clock, voltage glitch attacks, laser-fault attacks. Differential and correlation power side channel analysis extract side channel measurements using traces from the power consumption or electromagnetic radiation, and try to correlate it to a sensitive variable processed by the targeted system. Clock, voltage glitch, or laser-fault attacks induce a fault during the normal execution of the targeted system to extract information using differential fault analysis, or alternate the normal execution to an unexpected. Other types of hardware attacks include attacks using the JTAG debug interface [9]. The aforementioned attacks target vulnerabilities in the hardware and require physical access to the device. This limits the ability of the attacker to exploit system vulnerabilities in cases where physical access is not possible. In contrast, software attacks can be performed remotely, which allows an attacker to perform the malicious actions from almost anywhere.

Hardware attacks with physical access to the devices are outside the scope of this work. We assume that the attackers do not have physical access to the target system. In general, access to systems in industrial environments is limited, and consumer IoT devices are located in the user's home. Furthermore, if IoT devices can become targets of their own users, we assume that physical protections are in place. Examples of such protections include an integrated secure element or detection of attempts to dismantle the device. These are some of the physical protections that can be used to protect the system against physical attacks. We mentioned hardware attacks requiring physical access because they help us demonstrate the severity of a new class of attacks that we will introduce in the next Section 2.1.3.

2.1.3 Software Attacks Targeting Hardware Vulnerabilities

Due to the increasing demands of modern systems on computing power, memory, and utilities, modern architectures are becoming more complex. More and more components with complex capabilities are implemented in hardware to meet the increasing demands. This leads to a larger attack surface. Attackers are exploring the hardware to find vulnerabilities that allow them to explore the system. Thus, a new class of SATHV in the various units of the system has emerged. The targeted hardware units include the memory (e.g., cache, DRAM), power and frequency management modules, debugging interfaces, or components proposed to optimize computation time, such as the out-of-order and speculative execution.

Unlike the common hardware attacks mentioned in Section 2.1.2, where the attacker must have physical access to the target system, SATHV can be performed remotely. Since the attackers do not have physical access to their target systems, they access them through a communication interface such as Wi-Fi, Bluetooth, e.g., . This allows them to perform a remote access attack such as clock/voltage fault attacks [12], [13] or a JavaScript CacheSCA [14], [15]. CacheSCA [16], Rowhammer [37], Spectre [17], and Meltdown [18] are among the most serious non-invasive attacks to date that rely on hardware vulnerabilities and can be performed remotely. In the following subsections we will explain how these attacks work and which vulnerabilities they target.

Cache Side Channel Attacks

Cache Architecture CacheSCA are a set of attacks that target cache memory. Cache memory is a hardware component whose goal is to reduce the average time and energy cost of accessing data stored in main memory. Caches are smaller in size than main memory, faster, and located closer to CPU. Their purpose is to provide faster access to data or instructions that are frequently used during program execution. This saves valuable time by speeding up execution. Cache memories are often organized in a hierarchy ranging from closer to CPU to closer to main memory. The closer we are to CPU, the smaller the caches are and the larger they get as we get closer to main memory. The Level1 (L1) caches are also divided into L1 Data Cache (L1D) and L1 Instruction Cache (L1I). The instruction cache is responsible for storing the executed instructions, while the data cache stores the data to be processed.

In addition, there are the Translation Look-aside Buffer (TLB) caches. The TLB cache is a memory that stores the translations from virtual to physical addresses. The TLB is part of the Memory Management Unit (MMU), which is responsible for managing virtual memory. The MMU is a hardware component that handles all memory and cache

operations associated with the CPU, with the goal of effectively separating and protecting the memory used between processes. The TLB contains the most frequently used translations from virtual to physical addresses. Thus, when the CPU demands to access a virtual instruction or data address, the TLB translates that request into a physical address that the hardware can use to look up the requested data in the instruction or data caches. The TLB caches can also be divided into instruction TLB (iTLB) and data TLB (dTLB), and there can also be more than one level of TLB caches.

A typical two-level cache memory can be seen in Figure 2.1. Cache memories can also have more levels, as is typical for systems with high-processing power such as Intel or ARM CPUs targeting desktop systems.

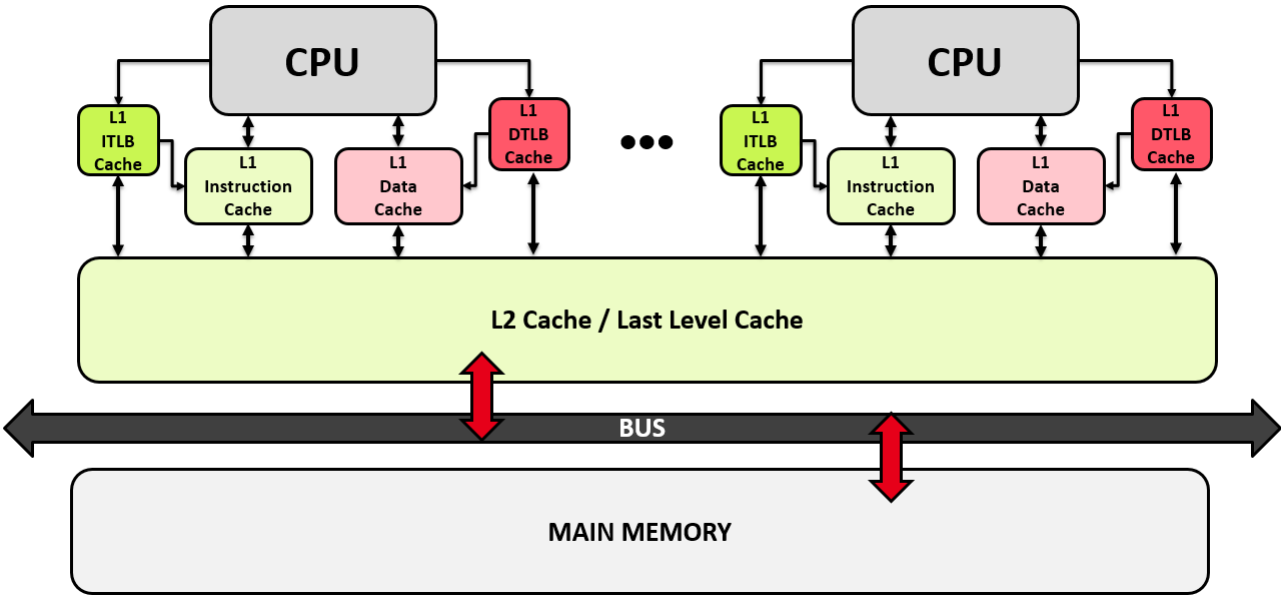


Figure 2.1: Memory hierarchy in a modern-CPU system.

CacheCSA Goals CacheCSA aims to extract information from caches using side-channel information. The reason for this is that the attackers have no knowledge of a secret parameter, which they want to extract via the cache as a side channel. The information extracted via the side channel is the timing for accessing the cache or main memory. If the requested data is stored in one of the cache levels, the request will be processed faster than if the data has to be transferred from the main memory.

Cache Side Channel Attacks targeted Hardware Vulnerability

The main vulnerability that attackers exploit with CacheCSA is the timing of a memory access. When data or address translations are cached, the request is processed faster than waiting for the request to be processed from main memory.

To exploit this vulnerability, there exist many works in the SOTA that propose methodologies to succeed. The different works target to exploit the vulnerability in the different levels of cache available in the system. For example, [38] attack the L1 Data Cache, [39]–[42] attack the L1 Instruction Cache, [43] target the TLB Cache, and [16], [44]–[49] attack the Last Level Cache (LLC), which in our examples in Figure 2.1 is the L2 Cache. Most, but not all, CacheCSA attack methods target the LLC. The reason for choosing this level is that the LLC is the cache shared by all CPU cores. This means that attackers can execute the attack in one core while running the victim in a separate core. If a CacheCSA is not running in the LLC, e.g., the L1D cache, the attacker must execute both the victim application and attacker code in the same physical core, which might reduce the attacks' applicability.

In the following paragraphs, we will explain some of the techniques used to extract sensitive information using the cache as a side channel. The techniques presented are not exhaustive, but they better demonstrate, with less complexity, the CacheCSA.

Flush+Reload The Flush+Reload (F+R) CacheCSA was introduced in [47]. The Flush+Reload in this work targeted the LLC. This attack uses the *flush* instruction, which removes the requested address from all levels of the cache. The *flush* instruction is available from userspace in Intel and ARMv8 architectures, but not in ARMv7 and RISCV. The attacker must perform several steps to succeed with the attack. In our example, the attacker and victim share memory addresses, which means that the attacker knows which address the victim will use to process the sensitive information. In Figure 2.2a we can see an example of Flush+Reload. The attacker first hypothesizes which virtual address the victim will use and to which physical address it translates to in the LLC (orange box). Then the attackers flush the address and invalidate it at all levels of the cache. Then they allow the victims to execute so that they can process the sensitive data. Finally, the attacker reloads the shared address and measures the time it takes to complete the request. If the access time is less than a threshold, the attacker can assume that the victim has accessed the hypothesized address. If the access time is longer than a threshold, the attacker assumes that the requested data is from

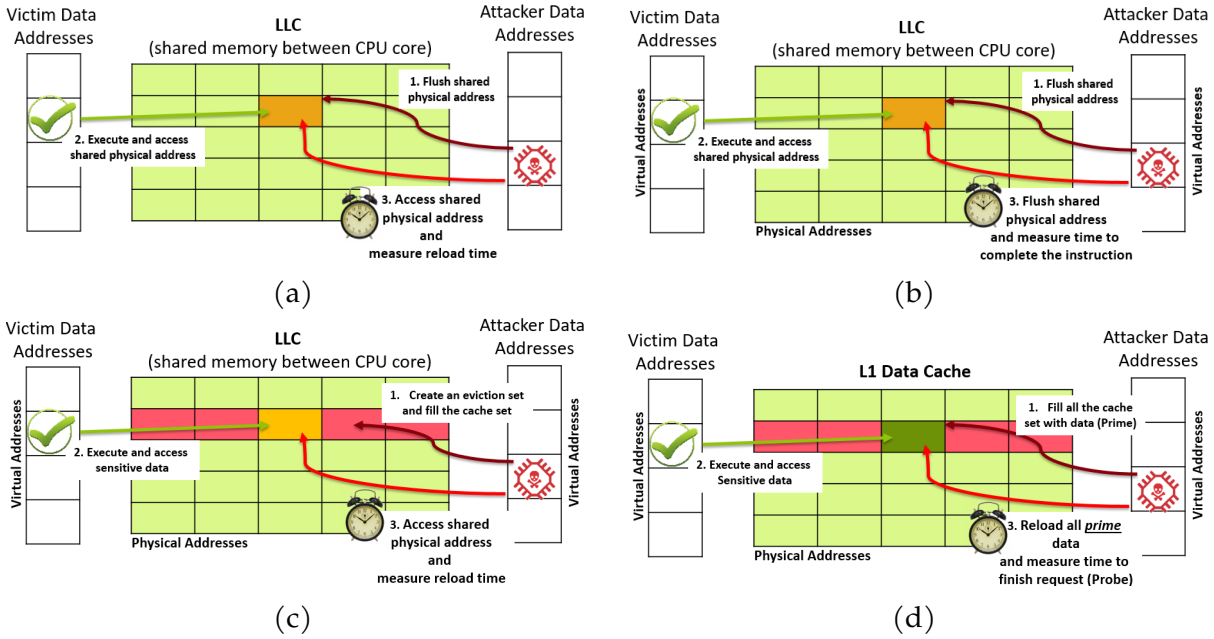


Figure 2.2: (a) Flush+Reload on shared address, (b) Flush+Flush on shared address, (c) Evict+Reload, (d) Prime+Probe.

main memory and that the victim did not process the sensitive data using the hypothesized address. When performing the attack for multiple hypotheses, the attacker can associate the sensitive value processed by the victim with the address with the shortest access time.

Flush+Flush The Flush+Flush (F+F) was introduced in [46]. The Flush+Flush in this work targeted the LLC and also used a shared memory between victim and attacker. This attack is similar to the Flush+Reload presented in Section 2.1.3, but the attackers now measure not the time taken to access the requested data, but the time taken for the *flush* instruction to invalidate the targeted address. This is because the execution time of the *flush* instruction depends on whether the requested data is cached or not. If the requested data is cached, the request will take longer than if it is not. Since the attackers first flush the targeted address, if the victim has accessed the data bringing them to the cache, the attackers can then measure the time it takes for the following *flush* instruction to complete. If the request takes longer than a certain threshold, it means that the victim has used the address, otherwise the victim has used a different address to process the sensitive data. An example of the Flush+Flush an the different steps can be seen in Figure 2.2b.

Evict+Reload The Evict+Reload (E+R) attack, which we will present, was introduced in [48]. This attack does not necessarily require the existence of the *flush* instruction, so

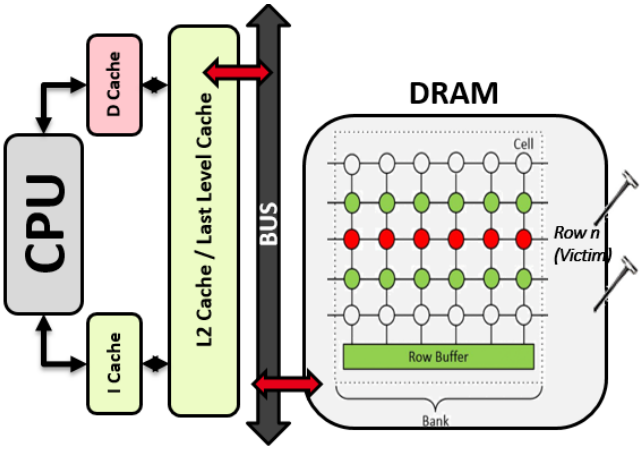
it can be used on all architectures. This attack is similar to Flush+Reload presented in Section 2.1.3, but now the attacker cannot use the *flush instruction*. This may be the case if there are restrictions on the use of the instruction from the user space or it is not implemented in the instruction set. To successfully perform the attack, the attacker must find a set of addresses that map in the same cache set. If the attacker successfully fills the set with its own data, it removes all other data from other processes stored in the same cache set, performing the equivalent of flushing the target address from the cache. With Flush+Reload, only the target address to be removed needs to be specified, which is then removed from the cache by the hardware in subsequent clock cycles. On the other hand using Evict+Reload, the attacker must request multiple data addresses to be stored in the cache in order to successfully remove the same target address. This increases the time required to execute the attack and increases noise from other processes using the same cache. The next step is to execute the victim, and as with Flush+Reload, the access time of the reload performed by the attacker will be faster than a threshold if the victim has used the target address Figure 2.2c.

Prime+Probe The Prime+Probe (P+P) attack we will present was introduced in [49]. This attack does not necessarily require the existence of the *flush* instruction, so it can also be used on all architectures. In this example, the attacker and the victim do not share the same addresses, so the attacker cannot request the targeted address directly because the MMU prohibits it. A common step used in many CacheCSA is to find the virtual to physical mapping of the target address. Since the MMU manages memory, each process has its own virtual address space that translates differently into physical addresses. The attackers must figure out which virtual address in their own virtual space translates to the targeted physical address, which in turn translates to the victim's virtual address. After this step, the attackers can carry out the attack using their different methods. In Prime+Probe, the attacker assumes that the victim will access a targeted address. After learning what physical address it translates, it tries to find addresses stored in the same cache set in its virtual space. We can think of the cache set as a full row in the cache, as seen in Figure 2.2. The attackers try to fill the cache set with their own data, as seen in Figure 2.2d. Then the victim executes and accesses the sensitive data. In the third step, the attacker reloads all the data it used to fill the cache set and measures the time it takes to retrieve it. If the request takes longer than a certain threshold, it means that the victim has accessed the sensitive data in that cache set removing the attacker's data. This means that when the attacker reloads the data, the data mapped in the address removed by the victim will need to be brought from memory, taking more time to finish the request.

Cache side-channel attacks require a lot of attention because attackers can perform them with or without using the *flush* instruction, and with or without shared memory, just by observing the timing information of accessing this fast memory compared to the slower main memory.

Rowhammer

The Rowhmammer attack was first presented by [5]. The vulnerability exploited in this attack is the side effects of repeated access to the same memory cells in the DRAM. Repeated access to a memory cell or pair of memory cells within a short period of time causes the memory cells to electrically interact with each other by leaking their charges. This can cause the contents of the memory cells in nearby memory rows that were not accessed during the original memory access to be altered. The attack is highly dependent on one particular parameter, namely the DRAM refresh interval. At each refresh interval, the DRAM recharges the memory cells because DRAM is essentially a set of capacitors whose charge must be refreshed periodically or the stored data will be lost. Attackers who want to successfully cause memory faults must access the targeted memory cells as often as possible, increasing electrical interaction. At each refresh interval, the DRAM recharges the memory cell charges, and the attacker's efforts are in vain as the cells are reset to the nominal charge.



(a)

```
input  : row(n) = Address of targeted row
         n
/* Address to induce memory errors     */
while True do
    read address at row(n+1);
    read address at row(n-1);
    remove from cache address at
     row(n+1);
    remove from cache address at row(n-1);
    memory barrier;
end
```

(b)

Figure 2.3: (a) Rowhammer, (b) Rowhammer pseudo-code.

Since the DRAM is after the cache memories and frequently used data is stored in the cache, accessing the same memory address causes the memory contents to be stored in the cache. A subsequent access will load the data from the cache and not from DRAM. Since the attackers want to avoid this situation, they flush the cache each time before requesting the same data address from memory to ensure that the subsequent memory

access brings the data from main memory. An example of the attack and its pseudocode can be seen in Figure 2.3. The authors in [5] have shown that double-sided rowhammer can be more effective than one-sided rowhammer. When using double-sided rowhammer as in Figure 2.3, we use the pseudocode in Figure 2.3b. The attackers need to find the addresses of the two adjustment rows to the target address they want to induce faults. After that, they start continuously flushing and accessing i.e., *hammering*, the adjustment rows. In the one-sided rowhammer, the attacker only *hammers* one row adjustment to the target row.

Rowhammer targeted Hardware Vulnerability

The main vulnerability that attackers exploit with Rowhammer is the electrical interaction of the DRAM memory cells. If the memory cells are accessed multiple times within a short period of time, this can cause the contexts of the DRAM memory cells adjacent to the accessed cells to be altered. This leads to targeted memory errors caused by the attacker. The electrical interaction between cells is becoming more significant with the modern high-density DRAMs.

Apart from using the *flush* instruction, [50] has shown that it is possible to induce memory errors using eviction techniques instead of the *flush* instruction. Using eviction techniques means that the attack is performed at a lower frequency than when using the *flush* instruction, which can affect the success rate in corrupting the memory cell. As explained earlier in Section 2.1.3, "flushing" the cache address using eviction-based techniques is slower compared to flush-based techniques. This as a result means we create less interactions between the DRAM cells per attacking period. With Rowhammer, an attacker can cause memory corruption that can affect the behavior of programs or gain privileges in the target system. Researchers in [51]–[53] have used Rowhammer to induce errors in a machine learning algorithm with the goal of forcing machine learning to misclassify the input. In addition, researchers in [54] used rowhammer to escalate privileges in the system. The authors use a malicious VM that exploits the bitflips triggered by rowhammer to crack the memory isolation enforced by virtualization.

Spectre

Spectre [17], [55] and Meltdown [18], which we will present in Section 2.1.3, are two of the most recent micro-architectural attacks. The two attacks were made public on 3 January 2018, after the affected hardware vendors had already been made aware of

the issue on 1 June 2017 [56]. Spectre is a micro-architectural attack affecting many CPUs with a branch prediction or other speculation forms, for examples Intel Skylake-Haswell, ARM-Cortex A72-A9 [57], AMD Naples-Toronto etc, . Spectre has concern the most the CPU vendors, as it can extract privileged information from userspace/unprivileged application. Spectre exploits a vulnerability in the branch speculation, which is a result of the demand for increasing CPU performance. A branch-predictor is a hardware unit that tries to predict the outcome of a branch destination address, when this address depends on a previous calculation not available yet to the CPU. This allows the CPU to continue executing subsequent code by speculating the result of a branch destination address when it is not sure of the real outcome. When the result of the previous computation is finally ready, the CPU verifies if its prediction was correct or not. If not, the CPU restores the execution back to the previous point and continues execution using the correct result. If the prediction was correct, the CPU can continue executing normally which effectively means we increase performance as we execute more code instead of staying idle waiting for a result.

Attackers can use speculation to force the CPU execute an attacker controlled branch destination address, accessing unprivileged code, before the CPU realize the fault. As the CPU will eventually realize the fault in the prediction, restoring the execution back to the correct result, the attackers will loose the unprivileged data accessed just before. One main idea attackers use to extract the unprivileged data is to use the cache as a side-channel. The trick is that attackers will force the CPU to speculative execute in order to access to a memory value at an attacker-chosen address. Then, it will perform a memory operation that modifies the cache state in a way that exposes this value [55].

Spectre targeted Hardware Vulnerability

The main vulnerability that attackers exploit with Spectre is accessing arbitrary memory locations during speculative execution due to the execution of subsequent instructions prior to verifying the correctness of the branch prediction result. The attacker can then affect the state of a side channel to extract the sensitive information.

Since the original Spectre whitepaper, many variants have appeared, targeting various forms of speculation.

Meltdown

Meltdown [18], is another hardware vulnerability presented together with Spectre. Meltdown can be used to read privileged memory in a process's address space that even the process cannot normally access. In some unprotected OS this can include data owned by the kernel or other processes. Meltdown affected most of Intel's CPUs [18], [58], while other vendors were not so affected. For example some ARM core affected are ARM Cortex A57, A72, A75 etc.

The main cause of Meltdown is the side effects caused by out-of-order execution. Out-of-order execution is another strategy introduced to speed up code execution. Instead of waiting for slow operations to finish and not remaining idle, CPU can execute out-of-order operations by scheduling subsequent tasks to execute on the idle components. The vulnerability exploited by attackers is the access to privileged data owned by the kernel or other processes as a result of the out-of-order execution. This is because when out-of-order accesses to privileged data are executed, they are temporarily stored in the system registers. An attacker can then use the value stored in the register to read an address from an array directly associated with the privileged data, as demonstrated with spectre. But since an attacker try to access the privileged data, the CPU will check the access privileges of the requested address when translating from virtual to physical address in the MMU. Since this operation can take some time, the CPU will access the illegal memory storing the result in the system register instead of waiting it.

Meltdown targeted Hardware Vulnerability

Meltdown exploits the side-effects of out-of-order execution to leak privileged data. This is succeeded by accessing the privileged address in the small timing window between the illegal memory access and the raising of the exception by the CPU.

Researchers in [18] shown that is is possible to succeed reading rates of privileged data at 503 KB/s with an error rate as low as 0.02 % when using exception suppression and with exception handling, they achieved average reading speeds of 123 KB/s with an error rate of 0.03 %. As the CPU raises an exception when running Meltdown due to the illegal access, the attackers can use in-code exception handling techniques to surpass the exception. This only requires operating system support to catch segmentation faults and continue operation afterwards. This demonstrates the high severity of Meltdown, which breaks the memory isolation barriers between processes enforced by the OS.

In summary, SATHV pose a serious threat to modern systems because they do not re-

quire physical access, can target vulnerabilities in the hardware itself, and can damage the system or extract non-privileged information without being detected by modern software tools. Because of their severity, a number of solutions have been proposed in SOTA to secure the system or detect them. In Section 2.3.2, we will present SOTA solutions that deal with SATHV and their relevant gaps.

2.1.4 Malware Classification

To summarize we will present a taxonomy of the different classes of malware in Table 2.1. We filled the table with information found from the SOTA [59]–[61], that will help better illustrate the different malware families and behaviors.

We divide the table into three columns. The first column indicates whether the attack was from software or hardware. The second column contains the malware class and the third column describes the goals of the malware. In the upper part of the table, we list buffer and heap overflows. Often these attacks are used to subsequently install other forms of malware. We also see that some malware occurs in both hardware and software, such as rootkits and Trojans. In addition, we list SATHV, which are of most interest in this work because they are able to extract sensitive information by exploiting vulnerabilities in the hardware itself. Moreover, patching hardware vulnerabilities is not as easy as software bugs, since patches can only be applied in new architectures, which can be costly and time-consuming. This makes it necessary to find ways to protect against SATHV. Last, we list hardware attacks, which, as mentioned earlier, are beyond the scope of this work, as we assume that the IIoT/IoT environment restricts physical access and/or physical protections exist.

2.2 Malware Detection

When it comes to protecting the system from malicious applications, we can distinguish between two approaches: Mitigation and Detection. Mitigation often involves developing patches to address issues related to software vulnerabilities or redesigning hardware. More complex modifications or redesign can be also necessary for software (for example if it requires some changes in the communication protocol). A major drawback of mitigation is that we apply security patches after vulnerabilities have been discovered and are likely to be exploited by attackers, and in some cases they cannot be applied directly to a vulnerable system. This is the case with attacks targeting hardware vulnerabilities, as the required fixes can only be applied to future architectures, which increases development costs. Another drawback is that the application of a security fix may run

SW or HW	Malware Class	Description
SW	Buffer Overflow	Attempts to insert more data than possible into a buffer, with the goal of overwriting system variables and redirecting program execution to execute arbitrary code. For example, install other types of malware.
SW	Heap Overflow	Attempts to insert more data than possible into the allocated buffer. This could lead to corruption of heap metadata or other heap objects, which in turn could allow arbitrary code execution.
SW	Virus	Malware that attaches to running applications and spreads through user interactions. A virus can be evasive using polymorphism, meaning that when it replicates to attach to a new target, it changes its payload to avoid detection. It can also be metamorphic, i.e., when it replicates to attach to a new target, it changes both payload and functionality, plus the means of changing form in the future.
SW	Worm	Malware that spreads from an infected host to other hosts via exploits in the OS interfaces, for example using system calls.
SW/HW	Trojan	Malware that tries to behave as a normal application and acts maliciously after installation, for example creating backdoors.
SW	Distributed Denial of Service (DDoS)	A DDoS attack aims to flood the devices, services, and network of the intended target with spoofed Internet traffic, rendering them inaccessible or useless to legitimate users.
SW	SpyWare	Malware that hides and monitors and reports on the user's computer usage and personal information.
SW	Ransomware	Ransomware is a malware that aims to deny legitimate users access to files on their computer by encrypting them and demanding payment for decryption.
SW	Keylogger	A keylogger is a form of malware or hardware that records and tracks legitimate users keystrokes as they type.
SW	Botnet	Malware that uses a legitimate user's computer to create a network of infected computers controlled by a central malicious organization.
SW/HW	Rootkit	A malware created to provide access to a computer that is otherwise not allowed, and which usually disguises its existence or the existence of other software.
SW	CacheCSA	A malware trying to extract sensitive information using the cache and its access timing as a side channel.
SW	Spectre	A malware that exploits speculative execution to read sensitive information and uses the cache as a side channel to extract the sensitive values.
SW	Meltdown	Same as Spectre, but exploiting the out-of-order execution.
SW	Rowhammer	A malware that causes memory errors in a targeted memory cell by continuously accessing its adjustment cells. The error is a result of the electrical interaction between memory cells and the increasing cell density in modern DRAMs.
HW	Fault Attacks (Laser, Clock, Voltage Glitch)	Hardware attacks aiming to redirect program flow or change the status of system registers

Table 2.1

counter to the original intended use of the component. For example, in speculative execution not allowing the CPU to execute part of the future branches, it contradicts the original purpose of the unit. This is to predict whether the branch will execute even if that case does not occur, in order to speed up execution. In [62], the authors discuss mitigations for Spectre and Meltdown and measure the induced overheads. They show that the proposed mitigations can increase the energy and performance overhead by up to 72%.

Based on the above arguments and because mitigation does not solve the problem, monitoring, detection, and response are a more definitive solution that helps us secure the system in the short and possibly long term. Detection allows us to find malicious applications before they can exploit the system, so we can take security measures. But unlike mitigation, which attempts to eliminate the vulnerability, detection only informs the system in the event of an attack, and appropriate action should then be taken to eliminate the threat. This means that there is a possibility that the attack will affect the system before it is detected, and that it will succeed in performing some or all of the malicious actions.

Detection is preferable to mitigation when mitigation techniques can take time to apply or require considerable modifications. Also, it depends heavily on the ability of detection implementations to quickly and effectively detect malware, and though reducing or minimizing the impact of the attack. In the next sections, we present the SOTA detection approaches that address our problem.

To detect malware, there are two main approaches: Static or Dynamic analysis.

2.2.1 Static Analysis

Static analysis checks parts of the application without actually executing it. A widely used static analysis technique is signature-based detection, which can detect known malware families. Antivirus programs use signature-based techniques to compare application signatures with known malware signatures stored in a database. A signature is a unique pattern of the applications executable. Signatures are created by scanning files for code strings responsible for the malware execution, permission requests, etc, [63]. The scanning of these information necessary to create the signatures can be extracted using decompilers, disassemblers, or source code analyzers. This allows examining with a high code coverage the executable offline. Since the signatures for each executable are unique, an antivirus can offer high detection rate and quick detection latency. This allows the antivirus to detect the malicious executable even before this

succeeds executing locally. The main drawback of static analysis is the inability to detect new malware classes because a malware signature is added to the database only after the malware is detected. This means that static analysis detection tools leave the system vulnerable to the malware until the database is updated with the new signature. Moreover, malware obfuscation [19] is a technique attackers use to modify the malware's signature to evade detection. Since minor changes to the malware file can result in different signatures, static analysis is not enough to fully protect the system. Other drawbacks of static analysis include cases where the malware is encrypted and decrypted only during execution, or cases where there is dynamic code loading or self-modifying code. In such cases, it is necessary to execute the malware to better understand its true behavior.

2.2.2 Dynamic Analysis

Dynamic analysis is an approach in which the malware is executed in a virtual or physical machine to extract information during its execution. This allows us to examine the behavior of the application during runtime, which provides a less abstract perspective on the application compared to static analysis. By executing the application, we can also observe the execution paths taken, which are a subset of all possible execution paths, since the execution path often depends on the application's inputs. The goal of dynamic analysis is to reveal the malicious activities performed by the executable while it is running without compromising the security of the analysis platform [33].

In order to successfully analyze a malware using a testbed during runtime, some properties must be respected [33]:

1. The extracted data must be trusted. The extracted information must not be compromised by the malware.

2. The malware must not detect the presence of the analysis tool. If it does, the malware could hide its true malicious activities. Also, the malware should not be able to impact the monitoring tool.

3. The analysis tools should extract as much relevant information about the malware's execution as possible, e.g., system calls, networking, modifications in local files.

4. The local system should be properly installed to allow the malware to execute as intended. Library dependencies, operating system version, etc. should all be properly installed to allow the malware to reveal its true functionality.

The above properties apply to the analysis of malware in a testbed environment. In a real system our goal is to restrict the execution of malware or minimize as possible their impact.

Dynamic analysis is more robust than static analysis when there is obfuscated or encrypted malware. This is because the actual malicious behavior does not completely change even if the malware is obfuscated or encrypted, because malware must perform a series of actions to achieve its malicious goals. Obfuscation only changes the sequence of malicious actions, and if the analysis tool is able to extract enough relative information during runtime, detection is still feasible.

In the next paragraphs, we describe the most promising techniques used in the literature for malware detection. Each technique has its advantages and limitations, which we will analyze in the context of our current problem of malware detection on resource-constrained devices, where memory and performance overheads are critical to adapting a solution.

Control/Data Flow Integrity Control Flow Integrity (CFI) and Data Flow Integrity (DFI) are two solutions proposed to protect the system from control-hijacking attacks, or malicious programs that attempt to subvert a program's intended data flow by exploiting memory corruption vulnerabilities. The CFI security policy requires software execution to follow a predefined path of an offline computed control flow graph (CFG) [64]. CFI secures the system in cases where the attackers corrupt control data, such as a function pointer, to redirect control flow. It compares at runtime the target of each indirect control flow transfer instruction with a set of allowed targets. If there is a mismatch between the calculated destination and the path taken, it reports this as a control hijacking attack. CFI protects against the second stage of memory-based attacks. This means that the attackers have already corrupted control or non-control flow data, allowing them to execute arbitrary code in a second stage. The CFI tries to detect or make it more difficult to execute this arbitrary code.

Unlike CFI, DFI attempts to protect the system against the first stage of the attack, i.e., the corruption of sensitive data such as control flow data, for example, the return addresses [65]. If the attackers manage to corrupt this sensitive data, they can hijack the execution flow. DFI ensures the trustworthiness of the data used at runtime, i.e., DFI ensures that the data used is not corrupted by an attack targeting a memory corruption vulnerability.

Both techniques have shown promising results in detecting attacks. However, their main drawback is the high memory and performance overhead on the local system.

The performance overhead is due to the runtime verification of the control or data flow at each occurrence, while the memory overhead is due to the additional instrumentation that these two techniques require, e.g., storing the statically computed control and data flow graphs. For example, [66] employs CFI to detect with 100% accuracy, but introduces a 10% performance overhead, 4% code overhead, and 0.81% memory overhead. Another approach, [67], uses DFI but also introduces a 7-15% performance overhead, a 5.42% code overhead, and a 1.9% memory overhead. Another DFI approach presented in [68] also resulted in a high performance overhead of 10-25% and a code size overhead of 12.5%. When we consider IoT systems where these resources are valuable, the performance and memory overhead caused by these techniques are not negligible. Backguard is another work in [65] where the authors considered all the IoT constraints, but their final solution has an average of 5.06% execution time overhead and requires 9% additional memory. Since these solutions add significant overheads to already resource-constrained systems, their adoption may be limited.

Another major disadvantage of DFI and CFI is the code instrumentation required. For example, Backguard [65] is implemented within the Clang/LLVM compiler infrastructure. This directly implies changes to the software vendor IPs, which in some cases are not even allowed. In the cases where CFI /DFI are implemented in hardware [69], [70], this implies direct changes in the CPU core and execution stages, and/or the addition of new instructions that verify the legitimate control flow.

On the other hand, there are techniques that dynamically monitor the system without requiring changes to the software. However, they may require the development of a custom application to extract system information periodically. Another benefit of dynamically monitoring the system using a custom application, is that we do not modify other IPs. Modifying IPs from other vendors can pose restrictions, as vendors might not allow access to their products so we can apply instrumentation.

Dynamic Analysis of API Calls Dynamic analysis can also use the API calls, system calls, native Windows API calls, file creation, process creation, or registry activities to model the behavior of normal and malicious applications. The API calls are used by a program to access system resources. Every application uses APIs to access resources such as networks, file systems and other software. Malicious applications regularly use certain APIs, which can be a good indicator of abnormal actions, such as modifying registry files to allow the malware to run even after the system reboots. The fingerprint of the executed system calls provides a lot of information that represents the raw interaction between the program and the host system, allowing a good characterization of the executed application. Dynamic analysis can also detect obfuscated malware, as

shown in [71], because despite the obfuscation techniques, the malware must perform a number of necessary actions to achieve its malicious goals.

Despite the promising accuracy in malware detection, a drawback of dynamic analysis of API calls is that SATHV do not rely on abnormal API calls to perform their malicious actions [72]. Moreover, monitoring the system at the system call granularity, provides us with less information about the underlying activities, which may reduce the detection rate. Another drawback we identify, is that API calls are extracted from software sources. Since these software sources are vulnerable to software attacks, we cannot be completely sure of their legitimacy, as attackers may find ways to compromise these software sources. This goes against property:1 introduced in the dynamic analysis essential properties.

Hardware Performance Counters Based Malware Detection Since monitoring API calls is not sufficient to detect SATHV, the researchers proposed the use of HPCs. HPCs are special-core registers available in most modern architectures such as Intel [73], ARM [74], RISC-V [75]. HPCs are available as part of the Performance Monitoring Unit (PMU) and allow monitoring of lower level micro-architectural parameters such as the total number of instructions executed, branch miss predictions, memory accesses, etc. This allows us to gain information about how the various hardware components behave during the execution of an application. Since SATHV and other malware tend to treat the system differently, this affects the behavior of the hardware components. Demme et al. [59] were the first to investigate the feasibility of detecting malware using HPCs. Since then, a number of works have been proposed using HPC information [76]–[80]. Considering that HPCs provide low-level information about hardware component behavior and SATHV tend to stress the hardware components to exploit the hardware vulnerability and extract the sensitive information, this source of information becomes interesting for detecting all subfamilies of malware i.e., classical malware and SATHV alltogether. Also, since these counters are implemented as hardware components, it is more difficult for attackers to manipulate the counted information. This increases their reliability.

On the downside, HPCs suffer from nondeterminism [59], [81]. Non-deterministic results mean that between two identical runs of the same application with the same inputs and parameters, it is very possible to have different results for the hardware events counted. [81] showed that HPCs can produce deterministic results in cases where the test environment is tightly controlled. Contributing to this non-determinism is the hardware itself, as modern systems execute instructions out-of-order, are speculative, or experience interference due to the OS. In addition, reading the HPCs means stopping the

normal execution of the CPU, which involves handling an interrupt and saving the CPU registers, which adds noise to the extracted values. Finally, since the HPCs are special-core registers and implementing registers within the CPU is expensive, these registers are limited. This means that we can only measure a limited number of hardware events at a time, equal to the number of HPC registers available.

Despite their drawbacks, researchers have widely used HPCs for detection. Further, the combination of HPCs and ML has become a standard tool for malware detection. In the next section Section 2.3, we will analyze the use of HPC-ML solutions that researchers have proposed in SOTA.

2.3 HPC-based Malware Detection, State of the Art

In this section, we present SOTA HPC-ML solutions proposed for SATHV and malware detection. Since HPCs provide us with the necessary information to distinguish between malicious and normal applications, the next step is to define rules that allow us to detect and categorize the different system behaviors during attack or normal operation. However, defining rules for the ever-increasing complexity of modern applications requires a high level of expertise and may still not be enough efficient to model most system behaviors resulting in missing malicious actions or raising false alerts. This is where machine learning comes in, a tool that can automatically learn rules and speed up the modeling of systems. In the next Section 2.3.1, we introduce the concept of ML in the area of security, while more background information can be found later in Section 2.4. After introducing the concept of ML and security, we present SOTA proposed solutions for SATHV and genral malware detection.

2.3.1 Machine Learning and Security

ML is a widely deployed tool for malware and intrusion detection. The increase in computing power and memory have accelerated the development of machine learning and its use in various fields.

Considering the field of security, a machine learning algorithm try to learn from a set of information extracted from the system, such as the HPCs or API calls, with the task to efficiently label normal and malicious applications, or the properly separate the different samples in the different applications.

Machine learning models can help us in many fields, but in the context of security two of these are most interesting, *classification* and *anomaly detection*. Classification is the task

of predicting in which class a given input belongs to. Given a set of input values, for example the HPC data, the ML algorithm will decide if this input belongs to the normal or malicious class. On the other hand, anomaly detection is used to discover anomalies in the dataset. In the general case, anomaly detection algorithms seek the input samples that differentiate the most among the rest of the samples. Both of these fields have been deployed in numerous SOTA works targeting malware and SATHV detection [76], [82]–[88]. The reason why researchers start to increasingly use ML for malware detection is its ability to learn complex patterns. Malware detection has moved from rule-based approaches [85], [89] to using ML, as modern applications start to become more and more complex, and also malware writers find more ways to penetrate the system and bypass simple rules by obfuscating their code. Since defining and developing rules is complex and can take time, the automated learning of patterns using ML help developers accelerate the deployment of new strategies, and quickly redefine the strategies in the presence of new threats.

In a following Section 2.4 we will provide more information of the ML algorithms used throughout this work.

2.3.2 State of the Art, SATHV Detection

As mentioned earlier, SATHV is a subclass of malware that is becoming increasingly popular and poses a growing threat to system security. Because SATHV targets vulnerabilities in hardware, mitigation efforts are currently insufficient to protect systems. This is because hardware mitigations can only be applied in future CPU architectures when software mitigations cause a large performance overhead [62]. For this reason, researchers have proposed detection techniques. In addition, since SATHV do not use abnormal API calls, researchers are investigating the use of HPCs that directly inform us about the hardware behavior of the system. However, most SOTA works either target a limited number of SATHV families possible in the system, or the target platforms are desktops or cloud systems where resources allow the implementation of complex solutions that can detect most SATHV families.

However, as attackers using SATHV also target IoT devices, proposed detection solutions should consider the limitations of these systems. As the IoT/IIoT become more complex to meet the increasing demand for computing power and memory, more and more of these devices are becoming vulnerable to SATHV. This is because vendors are adding more features to the systems, especially with the rise of edge-computing, where data is pre-processed in the edge-device and/or MLs are used locally for image or voice recognition. According to a survey [90], there were 27.7 IoT connections per 100 inhab-

itants in the European Union in 2017, a number that is rapidly increasing every year. The increasing use of IoT devices in many aspects of the daily life, the limited security guarantees that come with them, and their increasing complexity should lead to the proposal of malware detection implementations that take into account their limitations presented in Section 1.1.1.

In the following we present proposed works for SATHV detection.

Mushtaq et al. Nights-Watch [91] and WHISPER [92] Mushtaq et al. proposed a local CacheCSA detector targeting Intel-based systems in [91]. Their CacheCSA attack library includes Flush+Flush and Flush+Reload. The authors used only hardware events that provide cache state information, such as cache misses, cache hits, and cache accesses, etc. The proposed idea was successful in detecting Flush+Reload with 99.51% accuracy, 0% FPs, and 1.63% overhead using logistic regression in a noiseless environment. They also tested their solution in a noisy environment with 99.47% accuracy and 7.72% FPs.

A noiseless environment is an execution state in which each application starts executing only when the previous one finishes. A noisy environment is when two or more applications share the resources of the local system at the same time. This means that the OS schedules each application to run before the previous one finishes, giving each application the *"same"* execution time. The above is true when all applications have the same priority. If a task has higher priority, then the OS will give it more resources that the rest of the applications.

As we can see from the solution of Mushtaq et al. [91], testing in the noisy environment increased FPs. This can be attributed to the context switch increasing the noise in the cache. When a new application is scheduled by the OS, we observe increased cache misses as the new data should be loaded into the cache. Moreover, the proposed solution detects Flush+Flush with 91.73% accuracy, 0% FPs and 1.103% performance overhead in the noiseless environment. When the same model is tested in a noisy environment, the accuracy drops to 63.16% and the FPs increase to 1.86%. The lower accuracy in Flush+Flush can be attributed to the use of cache-relevant hardware events. The proposed method relies on hardware events that provide information about the cache state, such as cache hits, cache references, etc, but most importantly cache misses. This poses a problem when considering SATHV that do not cause many cache misses. One such case is Flush+Flush as mentioned in Section 2.1.3. In the original whitepaper [46], the authors refer to it as a stealthy CacheCSA. This is because the proposed detection mechanisms rely on the increased cache misses observed during Flush+Reload

and Prime+Probe. However, Flush+Flush does not increase cache misses because the receiver (attacker) of the covert side-channel does not access shared memory, resulting in no cache hits or cache misses.

The proposed solution successfully detects both attacks and also has a low FPR. However, the authors suggest using a specific ML for each malware class. This might not be a problem in the high resource systems they are targeting, but could cause difficulties in more limited devices such as the IoT.

In another work, Mushtaq et al. proposed WHISPER [92]. In this work, the authors' attack library included Prime+Probe, Flush+Flush, Flush+Reload, Spectre, and Meltdown. Also in this proposal, the authors suggest using a different ML model for each malware. They also note that each ML model has good attack detection accuracy, but has a high FPR. To reduce the FPR, they use an ensemble ML, where multiple models (decision trees, svm, random forest) are used for decision making. The final decision is made by a majority vote. Again, the proposed mechanism successfully detects malware while having a low FPR ($< 1\%$), but using multiple MLs in an ensemble technique and for each malware may increase the implementation cost and be restrictive for devices with limited resources.

Cho et al. [83] The authors in [83] proposed a mechanism for detecting local CacheCSA in real time in Intel server-based systems. The attack libraries include Flush+Flush, Flush+Reload, and Prime+Probe. They use five hardware events, three of which provide information about cache misses in the three cache layers. The authors use a single-layer NN with a softmax activation function (instead of returning a score for "0" or "1", this function returns a score for belonging to n classes i.e., normal, Prime+Probe, Flush+Flush, Flush+Reload), which they also used to detect the subfamily of CacheCSA in the system. The authors tested their mechanism in both noiseless and noisy environments in several server systems. In the noiseless environment, they achieved a maximum detection rate of 100% with a performance overhead of 0.6%, but in the noisy environment, the detection rate dropped to 95.4% and the performance overhead increased to 0.9%.

The proposed solution succeeds in detecting the three CacheCSAs in a noiseless environment with high accuracy and also has a low overhead. However, since the target system is a server, noisy environments are the most likely test case. In this case, the accuracy of the proposed idea decreases and some attacks are missed.

We observed the same in another model proposed by Tong et al. [93], where the authors use the same attack library and target system as Cho et al. [83], but they use a

Support Vector Machine (SVM) that detects attacks with 100% accuracy in a noiseless environment, but when tested in a noisy environment, the accuracy drops to 97% and 0.03% FPR.

Gulmezoglu et al. Fortuneteller [76] Gulmezoglu et al. presented Fortuneteller [76], a local SATHV detector targeting Intel server and laptop systems. The Attack library of this work includes Flush+Flush, Prime+Probe, Flush+Reload, Rowhammer, Spectre, Meltdown, and Zombieload (Intel speculative execution vulnerability only) [94]. The authors used Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) models in an unsupervised manner. They train their models only with normal applications, making their model sensitive to applications that differ too much from the baseline learned normal behavior.

Their technique is based on anomaly detection because they detect malicious applications when their model cannot predict them correctly because they behave differently from the normal applications. They use RNN and LSTM to model the sequential HPC data (time-series) from three hardware events that they extract every 1ms. The model is implemented to use one hundred past samples $(sample_{t-100}, ..., sample_t)$ to predict the next sample $sample'_{t+1}$. To decide whether an anomaly exists, the model calculates the Mean Square Error (MSE) of the actual sample at time $sample_{t+1}$ and the predicted $sample'_{t+1}$. If the prediction error is greater than a threshold τ_A, an anomaly is detected. To reduce the number of false alarms, the authors propose a sliding window technique. During the execution of an application, if the prediction error for the following 50 samples is greater than τ_A, an alarm is raised. This means that the model detects an attack only after 50 ms. They manage to detect malicious behavior with an F-score of 99.7%, an FPR of 0.125%, inducing a performance overhead of 3.5% in the local system.

The main drawback of this proposal, even if it succeeds in efficiently distinguishing malicious and normal applications, is the overhead incurred in the local system. The model has an overhead of 3.5% on an Intel server system, which cannot be compared to the most powerful IoT devices in terms of computation and memory resources. However, the authors show that it is possible to detect SATHV with high accuracy using more complex systems.

From the above SOTA works, we observe that increasing the SATHV attack libraries, SOTA works use more and more complicated ML models to be able to accurately detect the attacks, and still have a low FPR. This might not pose a direct problem to high resource systems, such as desktops and server environments, but it can be restrictive for lower resource devices such as IoT. Since SATHV start to pose a severe threat to IoT

devices, detection solutions should consider their limitations as well. In the next section Section 2.3.3, we present generic malware detection solutions, which specially targeting IoT devices.

2.3.3 State of the Art, Generic HPC Malware Detection for IoT

As we have seen in Section 2.3.2, the proposed solutions are implemented locally on the target system. But despite their good accuracy, they exhibit a high FPR. To reduce the FPR, more complex MLs models are preferred, such as LSTM, ensemble ML, or a specific model for each subfamily of SATHV. The increased model complexity potentially limits their implementation in IoT systems. In this section, we present SOTA works targeting IoT devices for the detection of malware classes included in Table 2.1 and not limited to only SATHV.

Kadiyala et al. presented a local malware solution targeting IoT devices in [79]. To reduce the overhead, they chose *"simple"* ML models such as random forest, decision trees, AdaBoost, and k-Nearest Neighbors. They monitor the system at the system call granularity, i.e., the extract HPC measurements every time a system call occurs. Although their implementations have a detection rate of 98.4%, they have an increased FPR of 3.1%.

Moreover, in [95], the authors showed that simple MLs are unable to identify attacks based on zero-day vulnerabilities and have an FPR of more than 2%. For their experiments, attacks based on zero-day vulnerabilities where malware classes not used during training. Since simple MLs have lower accuracy and higher FPR, more complex ML implementations are preferred. When we refer to complex MLs we refer to ML models with hundred/thousand of parameters more than simpler models. For example, a logistic regression model using six inputs have only 7 parameters, while an LSTM might have more than a thousand. The implementation of a local detection mechanism using complex MLs is presented in [96]. The authors propose the use of Hidden Markov Models (HMM) and LSTM algorithms for local malware detection. They succeed in achieving 95% and 98% accuracy with 0.38% and 0% FPR of the two implementations. Despite the high accuracy and low FPR of the proposed solutions, the drawback is the increased performance and memory overhead incurred for the local system. The authors determined that the LSTM requires 68kB of memory and takes 4 ms to make a decision, which can significantly increase the performance overhead. This restricts their ability to monitor faster than 4ms the system.

To reduce the overhead incurred in the local system, the SOTA works suggest the use

of remote implementations. This approach reduces the performance and memory overhead of the local implementation of the detection mechanism, but significantly increases the memory and communication overhead. In [88], the authors use LSTM and Conditional Restricted Boltzmann Machine (CRBM) in a remote system. They succeed in detecting malware with 99.97% accuracy and have an FPR of 0.5%. In [97], the authors use a one-class SVM and detect malware with 100% accuracy. Both implementations extract data every 1 ms, which significantly increases the communication overhead due to the generated data stored and later sent.

On the other hand, Wang et al. [98] proposed a solution for detecting malware in IoT systems that takes into account the communication overhead of transmitting all extracted raw HPC data. The authors recognized that more complex techniques cannot be implemented locally, while the transmission of raw HPC data can increase the communication bandwidth and thus the load on the network considering the hundreds of IoT installed. To reduce the communication overhead, they propose to compress the raw HPC data locally and send a reduced representation remotely. The authors found that increasing the compression rate reduces accuracy. A compression rate of 20% and 30% showed the best detection rate. This allowed them to detect malware using signature-based techniques that compared the execution traces of different applications against a remote database.

Despite their suggestion, the compression succeeded through complex matrix multiplications, which the authors noted increase the execution overhead exponentially with the size of the data being compressed and linearly with the compression rate.

2.3.4 Summary

In Section 2.3.2 and Section 2.3.3, we presented some SOTA works proposed for SATHV detection and generic malware detection. We show that we can distinguish between two implementations, i.e., local and remote. When the detection mechanism is implemented locally, the device can benefit from fast detection. However, we show that when simpler ML algorithms are used, the overhead incurred in the local system is low, but the FPR increases. If the device sees FPs frequently despite the high attack detection rate, the final overhead due to system *"reset"* actions may reduce the benefits of using simple techniques. On the other hand, using more complex ML models helps reduce FPR, but at the cost of increased overhead in performance and/or memory.

On the other hand, by using remote detection ML we can implement models without resource constraints. However, this comes at the cost of slow detection time, since the

data must be transmitted to the remote system to finally obtain the decision. Second, the bandwidth increases due to data transmission, which can overload the network with hundreds of devices sending several Mb of data per minute.

Detection solutions targeting the IoT should consider the limitations of these devices. If any of their limitations are not taken into account, the security mechanism may not be adopted, even if it has high efficiency.

2.4 Background on Machine Learning and Feature Extraction

In this section, we provide background information regarding the use of ML models for malware detection. We explain how the algorithms works, which will allow us to better demonstrate the applicability of different models later in the chapters.

As previously mentioned in Section 2.3.1, machine learning is a tool that can automatically learn complex rules from the input dataset to be able to differentiate between normal and malicious behaviors. Researchers have extensively used it for dynamic malware detection, proposing many models with different capabilities. In this section, we will provide the necessary information to allow us to better illustrate the benefits and drawbacks of each technique and each machine learning algorithm.

2.4.1 Supervised ML

Supervised ML is a type of ML, where the designer should provide both the inputs and the targets. In this type of ML algorithms, the designer acts like a teacher, providing the input data and teaching the algorithm the correct result. When using supervised ML, we expect the ML algorithm to be able to learn useful patterns during training to be able to distinguish the inputs between the different targets. Since the designer provides both the inputs and the outputs to the model, this type tends to be fast and accurate. However, the model must also be able to generalize, as it may learn to accurately separate training samples, but not be able to determine how to separate unknown to it samples. For example, in malware detection, a model may be able to detect known malware very accurately, but may not be able to detect new malware families or malware that is slightly different. This is a limitation that also applies to static analysis techniques such as antivirus programs.

In the following, we present the supervised ML algorithms used in this work and in SOTA.

Logistic Regression

Logistic regression [99] is one of the simplest classification algorithms. It takes *n* inputs and assigns the observations to a discrete set of classes. If only two classes exist, it is a binary classification. This is the general case in malware detection when there are only two classes, i.e., normal and malicious. If we want to have more output classes, for example, if we want to have a normal class and a class for each malware family (rootkits, SATHV, viruses, etc,), then the problem is a multinomial logistic regression. We consider malicious samples as belonging to the positive class and normal samples as belonging to the negative class. The logistic regression attempts to separate the two classes using a linear hyperplane. When using *n* inputs, the hyperplane is a hyperplane of dimension *n-1*. For example, if we have only two inputs, the hyperplane is simply a line.

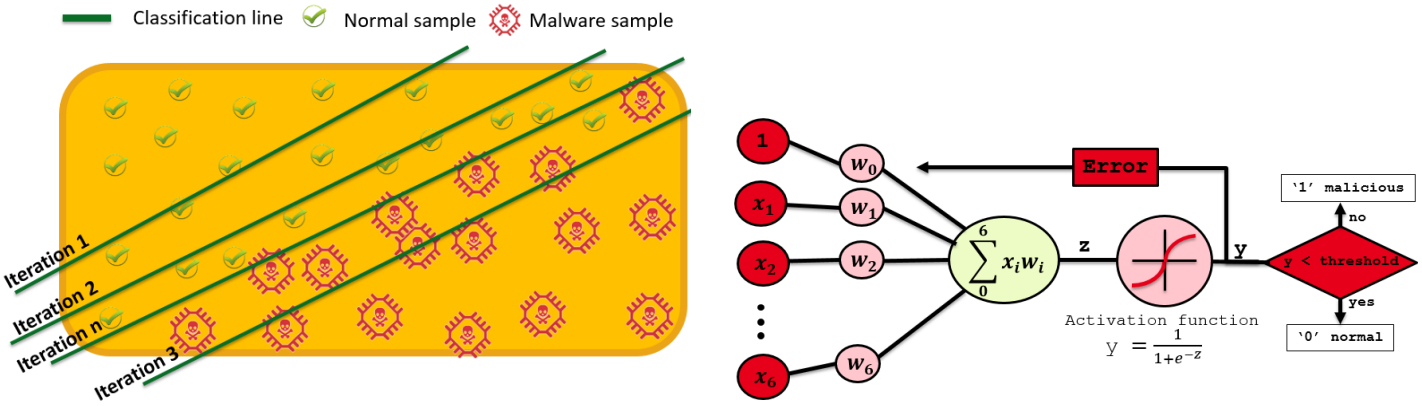
This is a simple algorithm to implement, requires limited resources reducing potentially overheads, but it can solve effectively only linear problems. In the case of malware analysis, where the problem is not linear, misclassifications are most probable.

To find the best hyperplane, the algorithm represents the hyperplane by a linear equation such that the weights are those that minimize the cross entropy loss [100]. The hyperplane can be expressed by the following equation Equation (2.1):

$$z = w_1 * x_1 + w_2 * x_2 + ... + w_n * x_n + b \tag{2.1}$$

In Figure 2.4a we can see how the classification line is changed for each training iteration. For *iteration 1*, the classification accuracy is low because there are multiple FPs. After *iteration 2*, the classification accuracy improves, but there are still multiple FPs. At *iteration 3*, the classification accuracy improves slightly, but now there are multiple FNs. The training is terminated in *iteration n* when the classification accuracy is the best and has the fewest FNs and FPs.

Finally, to make a classification, the classifier takes the output probability $\hat{y}$ and compares it to a predefined threshold. Depending on this threshold, which is usually equal to the 0.5 (50%) probability that the sample is positive, the sample is classified as "1" if the probability is greater than the threshold, or as "0" if the probability is less. You can see the logistic regression model in Figure 2.4b.



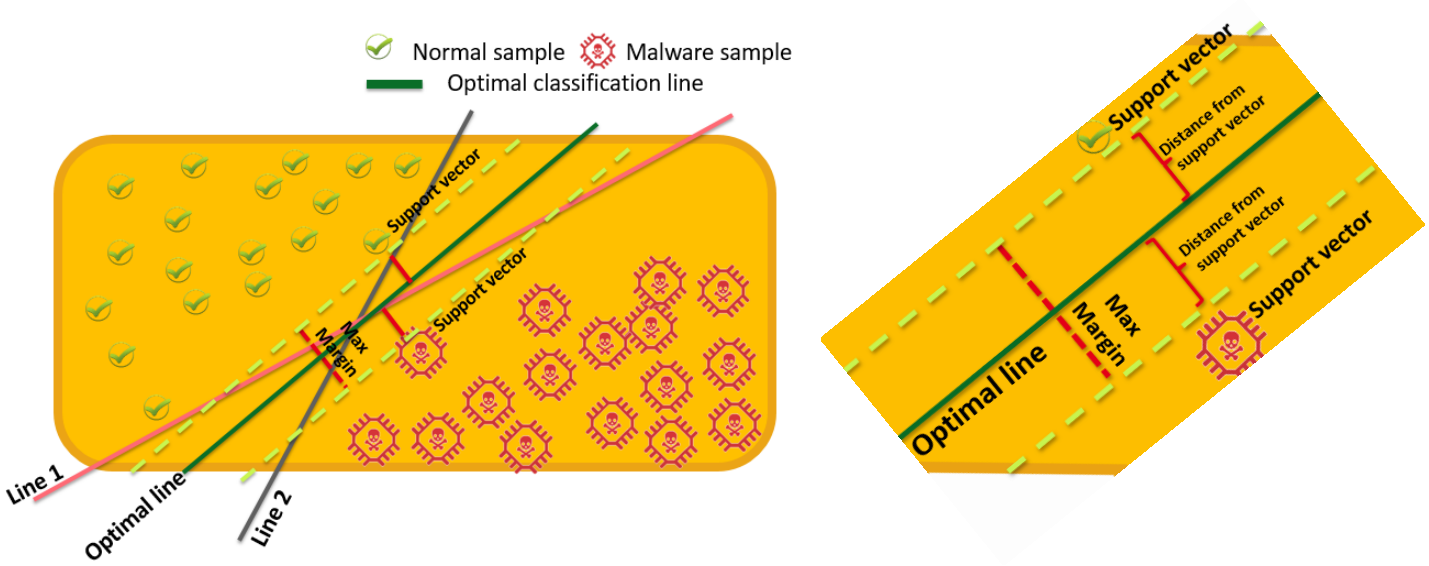
(a) (b)

Figure 2.4: (a) Logistic Regression training iterations, (b) Logistic Regression Visualization with 6 inputs.

Support Vector Machines and Linear Support Vector Classification

Support Vector Machines (SVM) [101] and Linear Support Vector Classification (LinearSVC) [102] are also linear classification algorithms, like logistic regression. Depending on the method used to train the classifier, it can solve linear and non-linear classification problems. It also tries to find a hyperplane that best separates the two classes. However, there are usually one or more hyperplanes that separate the two classes with the same accuracy. Logistic regression stops training when one of these hyperplanes is found for the first time. SVM, on the other hand, tries to find the best hyperplane among the candidates that best generalizes the classification problem. This is accomplished by finding the hyperplane with the highest margin among the candidate hyperplanes. The margin is calculated using the support vectors, i.e., the samples closest to the hyperplanes between the two classes. From Figure 2.5a, we can see all three lines separate the two classes with the same accuracy, since we have no misclassification for all of them. However, we can also see that line 1 and line 3 are very close to the samples of the two classes, which reduces the margin. On the other hand, the optimal line maximizes the margin since it has the largest distance from the support vectors, as can be seen in Figure 2.5b. We can say that SVM tries to find the hyperplane (line) that separates the two classes as much as possible. This increases the generalization, since the classification line is not close to the train samples, but remains as far away as possible and maintain the same accuracy.

Nonlinear problems can be solved by SVM by transforming the input data using nonlinear functions and solving the same problem in higher dimensions, e.g., by creating a new variable equal to $x^2 + y^2$. This comes at the cost of added complexity. Another disadvantage of the nonlinear SVM is the long training time, since it must compare the



(a) (b)

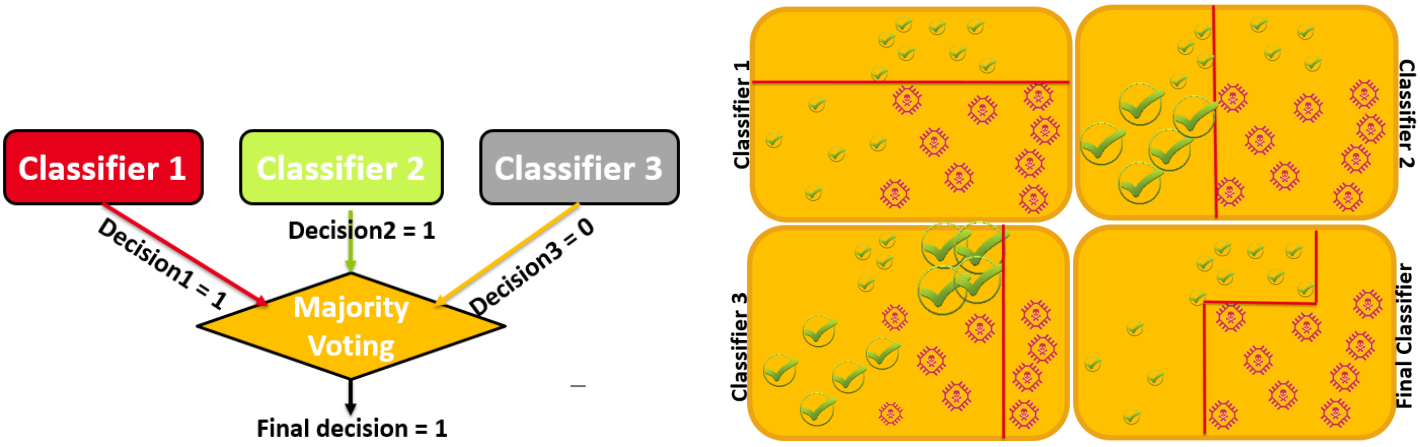
Figure 2.5: (a) SVM classification lines and optimal line according to the support vectors, (b) SVM zoom.

distance between each sample points. According to [103], the complexity of the SVM scales between $\mathcal{O}(n_{inputs} * n^2_{samples})$ and $\mathcal{O}(n_{inputs} * n^3_{samples})$, where ($n_{inputs}$ is the number of information sources (features) we use for classification e.g., six HPCs. When we consider that datasets can include millions of samples, with each sample having many features, the complexity of the problem becomes very high. To reduce the complexity of training and implementation, in this work we only consider the linear approximation, which we refer to as LinearSVC. Compared to the nonlinear SVM, LinearSVC only has a complexity of $\mathcal{O}(n_{inputs} * n_{samples})$.

AdaBoost

So far we have explained classifiers that use a linear hyperplane to separate classes. But usually classification problems are not that simple, and a linear hyperplane cannot separate the two classes efficiently. To increase the accuracy of linear (or non-linear) classifiers, ensemble learning can be used. Ensemble learning combines the results of two or more classifiers and uses majority voting to determine the final classification, as seen in Figure 2.6a.

AdaBoost [104] is an ensemble learning method that attempts to increase the efficiency of binary classifiers through an iterative approach that learns from the mistakes of weak binary classifiers to eventually create more efficient ones. The idea behind Adaboost (Adaptive Boosting) is that a single weak classifier may not be efficient enough to accu-



(a) (b)

Figure 2.6: (a) Ensemble Learning, (b) Adaboost training example using three weak classifiers.

rately predict the class of a sample. However, if we use multiple weak classifiers, each of which gradually learns from the incorrect decisions of the previous one, this can eventually lead to a stronger classifier. An example of such a weak classifier might be a LinearSVC or a Logistic Regression. When we refer to a classifier as weak, we are not referring to a random classifier (i.e., it makes decisions that are correct with a probability of 50%), but to a classifier that has an accuracy of, say, 89%. In Adaboost, a classifier is first trained on the training samples and evaluated. Then, samples that are misclassified are weighted more heavily so that the second classifier knows that it must try to predict these samples correctly. Then the second classifier is evaluated and the weight of the correctly predicted samples decreases while the weight of the misclassified samples increases. The final strong classifier is the combination of the weak classifiers, and a decision is made by a weighted voting. A weighted voting means that if a weak learner makes less errors, it has a higher weight in the final decision.

An example can be seen in Figure 2.6b. The first classifier separates the two classes with the linear line. As we can see, most normal samples and most malware samples are classified correctly, but there are some FPs. Next, we assign more weight to these misclassified samples and less weight to the correctly classified samples, so classifier 2 tries to predict them correctly. Classifier 2 now correctly predicts the misclassified samples, but we have misclassified some other points. Again, we decrease the weight of the correctly classified samples and increase the weight of the misclassified samples. Next, we train classifier 3 with the reweighted samples and obtain a third classification line. Finally, by combining the decisions of the three classifiers, we obtain the nonlinear classification line that correctly predicts most samples and increases the accuracy of the

weak classifiers.

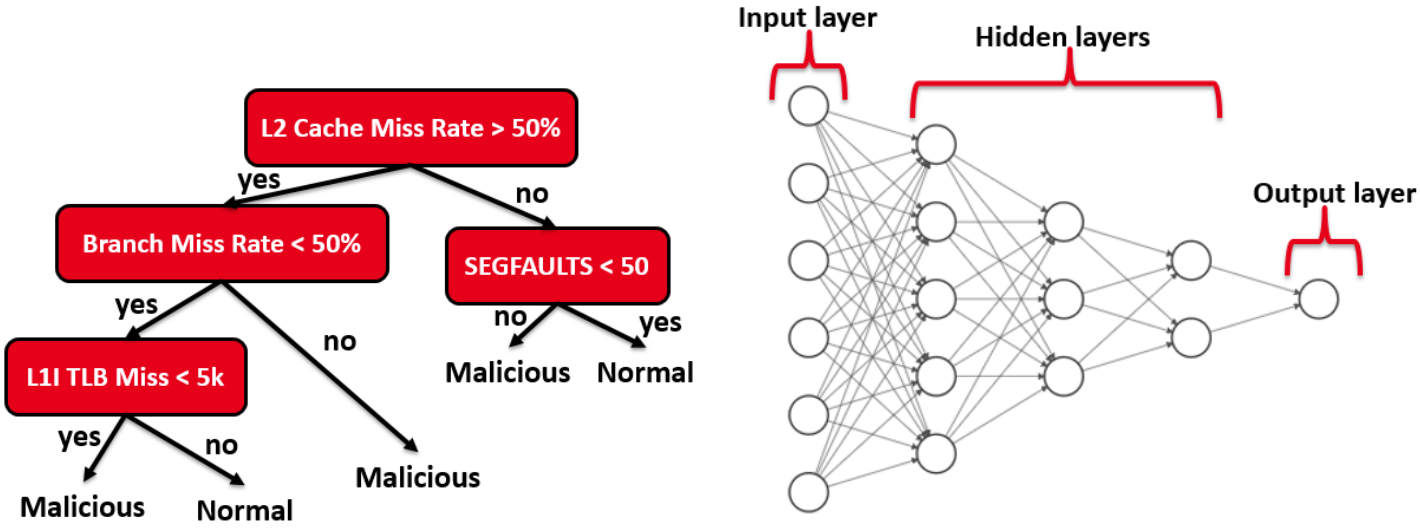
Decision Trees

Decision trees [105] are a non-linear classifier which has the structure of the tree, i.e., it has a root node, intermediate nodes, branches, and leaf nodes. At each node one of the sample's features is evaluated and depending on a condition a different branch is taken. The tree is structured so as an internal node makes a test on a feature of the input samples (i.e., choose one HPC out of six to create a test), then each branch represents an outcome of the test, and each leaf node (or terminal node) holds the predicted class label. During training the decision tree is constructed by recursively evaluating different features and using at each node the feature that best splits the data between each class. An example can be seen in Figure 2.7a. Decision trees are easy to implement as they require no data preprocessing (for example scaling the inputs in the same range of values), and they can be implemented using simple if-else statements, which greatly reduces the implementation cost. A known limitation of decision trees is that they usually overfit, i.e., they learn to accurately separate the training data but fall short on accurately predicting new samples. An exmaple of a decision tree can be seen in Figure 2.7a.

To increase the accuracy of decision trees, multiple decision trees can be combined in an ensemble classifier structure or using a techniques such as Adaboost. Another classifier that is gaining popularity is XGBoost [106]. XGBoost is an implementation of the gradient boosted trees algorithm. It attempts to accurately predict the target class by combining the estimated decisions of a set of weaker trees, similar to Adaboost. In gradient boosting, the model adds a new weak learner with the goal of minimizing the error over the objective function when the weak learners are combined. The idea behind gradient boosting is to set the target outcome for the next model in a way that minimizes the prediction error. As in logistic regression, the new model is modified in the direction of the minimum error using gradient descent algorithm. XGBoost also introduces regularization penalties to ensure that the model does not overfit.

Neural Networks and Deep Neural Networks

Neural Network (NN) [107] are a subset of machine learning and the foundation of deep learning models. A NN has an input layer, a hidden layer, and an output layer. In cases that exist more than one hidden layers we refer to as Deep Neural Network (DNN) [108]. A layer can be seen as a container of multiple neurons. An example of a DNN can be seen in Figure 2.7b. The nodes of the network are interconnected so that it works like a human brain, passing information to the next neurons. Each neuron can be



(a) (b)

Figure 2.7: (a) Decision Tree, (b) DNN with 3 hidden layers.

visualized as the basic logistic regression block presented earlier. The neuron takes the inputs, multiplies them with the weights, calculates the sum of the multiplications and a bias and finally produces an output using either the sigmoid or another activation function. The activation functions are used to introduce non-linearity to the neuron, and decide if the neuron is activated or not. The output of the neuron is then connected to the neurons of the next layer. DNNs are more complex ML algorithms, which allow them to learn more complex patterns. Each layer learns different patterns which then combine to make the proper decision.

Convolutional Neural Networks

The previously described models are simple models that, depending on the complexity of their implementation, can learn complex patterns in the dataset to make correct classifications. A limitation of the previously mentioned models is that the classification task becomes more complicated and many misclassifications may occur if the input features do not provide a strong distinction between the target classes (e.g., the values of different classes are in different ranges). Therefore, it is even more important to select the most relevant features for input. This means that designers should have expertise in this area to suggest relevant features. In this way, these simple models can become more efficient and reduce the complexity of the implementation while maintaining high accuracy.

In our problem, the inputs can be considered as time-series data sets. The HPC waveforms of the different applications can provide us with information about the different

tasks that are performed during execution. This allows us to not only look at the granularity of the samples to decide if malware is present or not, but also look at a time window of samples. This can give us more information to make proper decisions, as we can now look at the sequence of actions rather than just one action. Since malware needs to perform multiple actions to successfully exploit the system, the above intuition seems most appropriate for our classification problem. However, applying this method raises the problem of how we can recognize this set of actions that allow us to distinguish malware from normal applications, as this would require manual analysis of thousands of malware executions.

Convolutional Neural Networks (CNNs) [109] are DNN models that revolutionize the way we use ML. One of the key advantages of CNNs is their ability to automatically extract and generate deep features from the input. CNNs were first used in computer vision and image classification, but slowly they found their way into time-series classification. In short, a CNN uses various convolutions to extract deep features from the input dataset. This creates a new feature map that we can then input into a DNN to determine the class of input samples. A CNN can potentially increase accuracy compared to a simpler ML model, but at the cost of added complexity due to the numerous required complex operations at multiple layers. An example of a CNN for time-series classification can be seen in Figure 2.8. The network takes the input, performs one or more convolutions and creates a new feature layer, which then we input to a DNN to get the output class.

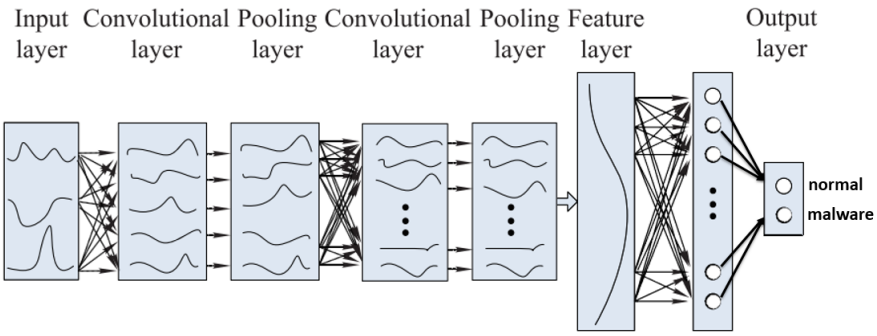


Figure 2.8: CNN for time-series classification [110].

2.4.2 Un-Supervised ML

In unsupervised machine learning, we do not provide the target labels, but instead the network tries to learn itself useful rules to separate the classes. In this work, we do not consider this strict case, but instead we consider the case where we only train the model with non-malicious applications. In such a case, the models try to learn the nor-

mal dataset as best as possible, and detect as anomalies any deviation from the normal behavior. Since the model must learn the normal behavior, be able to generalize, and detect malwares as strong deviations from the normal behavior, these models tend to require a lot of resources. The increasing complexity allows them to learn many complex patterns in the dataset, which increases their learning capacity. But, as with the CNNs, the increasing complexity increases the implementation costs.

Autoencoder

An autoencoder [111] is a special DNN. Autoencoders attempt to learn efficient patterns from training data so that they can be encoded in a compressed version. For example, if the network has n inputs, it will try to compress them into a lower dimension i.e., $m < n$. Then the autoencoders will try to reconstruct the original input from the compressed representation as efficiently as possible. An autoencoder representation can be found in Figure 2.9. In the figure, we see an autoencoder with n=6 HPC inputs, compressing them to a m=2 dimension, and then reconstructing the HPC inputs into a close approximation HPC'.

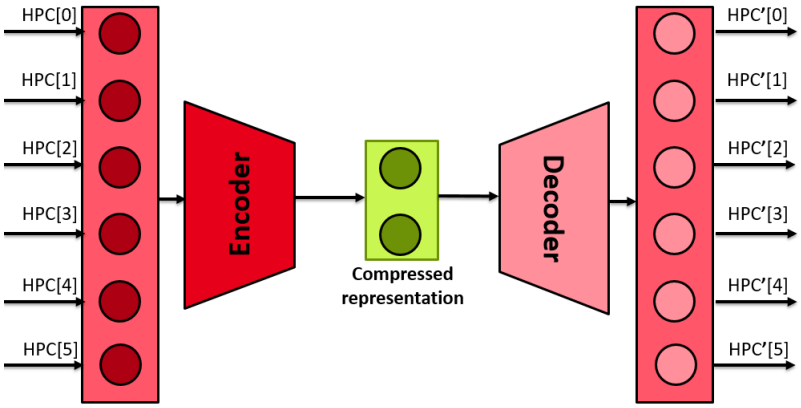


Figure 2.9: Autoencoder representation.

The intuition behind using an autoencoder for malware detection is that it can learn to efficiently reconstruct the normal dataset used in training. Given a normal sample $HPC[0:n]$ as input, it reconstructs it as $HPC'[0:n]$, where:

$$|HPC[0:n] - HPC'[0:n]| < \delta$$

This δ can be used as a threshold to detect anomalies in the input data. If a sample is reconstructed with an error greater than δ, we can conclude that this sample is an anomaly. Malware are applications that the autoencoder did not see during training and that also do not conform to normal behavior. We hypothesize that if a malware sample is inputted to the autoencoder, it is very likely that it will be reconstructed with

a high error greater than δ, so we can raise an alert.

Isolation Forest

Isolation Forest [112] is another unsupervised machine learning algorithm based on the decision tree implementation. It isolates outliers (anomalies) from the dataset based on a contamination parameter. The contamination parameter is defined during training to notify the algorithm the percentage of anomalies in the data set. Basically, the contamination parameter is then used to define a threshold for anomalies that is used to compare the "anomaly score" of the input data to decide if they are anomalies. The algorithm randomly selects features from the training data set and then randomly selects a split value in the *min-max* range of that feature. The intuition behind the algorithm is that randomly partitioning the features results in shorter path distances in the trees for the anomaly samples, i.e., the anomalies are thus closer to the root of the tree. The term path distance is the number of edges taken from the root to reach the final leaf node and defines the anomaly score for this algorithm. In this way, we can distinguish anomalies as points that are closer to the root of the tree.

An isolation forest may consist of one or more trees in some sort of ensemble machine leanring. The final decision is made by averaging the distance to classify a sample of all trees. If the distance is less than a threshold, it is decided that the sample is an anomaly.

The isolation forest, unlike the autoencoder, is not trained only with normal data, but it requires some real anomalies in the dataset. It is unsupervised because we do not provide the target labels, but the algorithm tries to detect the anomalous samples by itself.

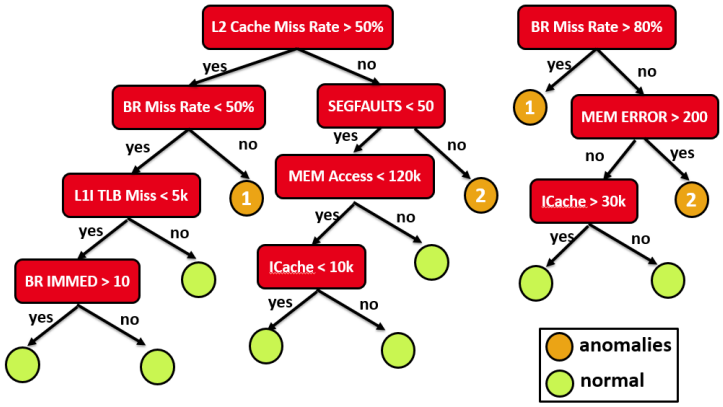


Figure 2.10: Isolation forest with two decision trees.

As we can see in Figure 2.10, the true anomaly samples 1 and 2 are closer to the root for both trees. For example, sample 1 has an average path length of 1.5 and sample 2 has

an average path length of 2. Since both samples have a shorter path length than normal samples setting in this case the anomaly score to 2 can allows us to detect them.

Long Short-Term Memory

LSTM [113] is another unsupervised machine learning algorithm that we use to learn the normal behavior of the system, as in the autoencoders presented in Section 2.4.2. Most of the models presented rely on sampling granularity (i.e., a single sample at a given time t) to decide whether or not a malicious application executes. But as mentioned in Section 2.4.1, the execution of different applications on the system is based on a time-series of events. Instead of looking at a single sample, we can better model the behavior of normal applications by looking at past actions as well. Also, malicious applications usually require more than one action to achieve their malicious goals. For example, in the Spectre attack presented in Section 2.1.3 and [55], the attacker needs to set the cache memory to a known state, remove the value of *array1_size* and all *array2* from the cache, exploit the branch prediction vulnerability, and finally extract the secret using the cache as a side-channel. Since we can model the normal behavior more efficiently, and since we can detect the malicious applications based on a set of patterns with abnormal behavior, the time-series classification can reduce the FPs.

LSTMs can be visualized as a series of cells, with each past cell passing information to the next cell. The next cell then uses the current sample and information from the previous cell to process the data. An example can be seen in Figure 2.11, where this network receives as input n past samples, each sample having six features [0:5], and tries to predict the future m sample points. In this example, the features are six different hardware events.

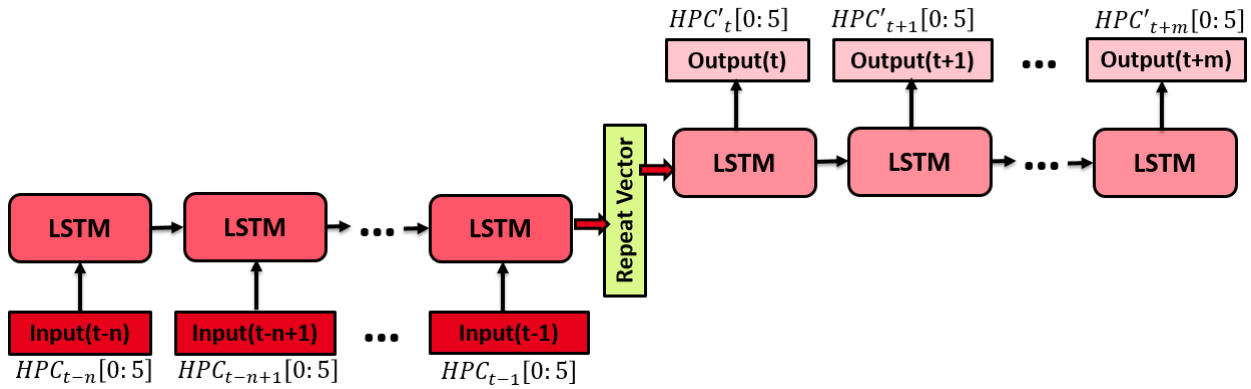


Figure 2.11: LSTM representation which use the n past samples to predict the m future samples. *HPC[0:5] is because we use 6 HPC events.*

LSTMs were introduced by Hochreiter et al. [113] in 1997 and have been improved

over the years. LSTMs are capable of learning long and short term dependencies. This means that there are times when a future task of an application depends on only a few past actions, but there are also times when a future task depends on a number of past actions. In simple words, LSTMs are able to decide how much information from the distant past and how much information from the near past should be incorporated into current sample processing [114].

The basic intuition behind using LSTMs for anomaly detection is that during training, the network can learn to efficiently predict future normal behavior based on past normal behavior. As with autoencoders, the LSTM also learns to predict the future m samples with a small error δ. In the cases where malware is executed, the series of actions that the malware performs does not allow the LSTM to successfully predict its future actions. This means that the LSTM will most likely predict the m future samples based on the unknown malicious behavior with an error greater than δ. If we set the threshold for anomaly detection greater than δ, any prediction error greater than δ can allow us to detect malicious actions on the system.

Long Short-Term Memory Autoencoder An LSTM autoencoder [115] is a special network based on the idea of LSTM, but instead of predicting the m future samples, the network takes as input the n past samples plus the current sample and tries to reconstruct this behavior as efficiently as possible. The network takes the past samples, compresses them, and then reconstructs them, as can be seen in Figure 2.12. As we can see, the network takes as input $HPC_{t-n}[0:5]...HPC_t[0:5]$ and reconstructs it to $HPC'_{t-n}[0:5]...HPC'_t[0:5]$.

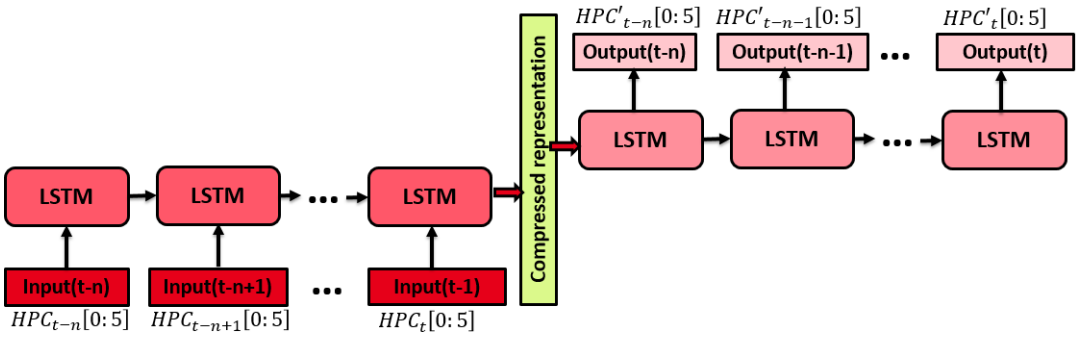


Figure 2.12: LSTM Autoencoder representation which use the ***n*** past samples HPC[0:5] plus the current sample t, compresses them, and later reconstructs them as HPC'[0:5]. *HPC[0:5] is because we use 6 HPC events.*

2.4.3 Supervised vs Unsupervised Machine Learning Models

Each technique used to train machine learning has its own advantages and pitfalls for our classification problem. We summarize them below:

- Using supervised ML can be more efficient and lead to simpler models because we know the malicious and normal behaviors we want to detect. The network can learn useful patterns from the data and knowledge of the target label we specify. In our case, the biggest advantage of the supervised ML is the lower implementation cost as we target resource-limited devices. Also, the cost can be further reduced by using features that best separate the two classes. However, a drawback is that while a supervised ML is capable of detecting known attacks and attacks similar to those already known, it can be not sufficient to detect attacks based on zero-day vulnerabilities. In such cases, it is necessary to retrain the model.
- Unsupervised ML is powerful because it allows us to build a model that efficiently characterizes normal behavior and recognizes any deviation from that normal behavior as "malicious." Unsupervised MLs are limited, however, because the more normal application behavior we want to model, the more complex the model must be. This is because in order to learn more behaviors, we need to increase the learning capacity of the models. For example, if we use an LSTM, we need to increase the number of cells so that the new cells learn the additional normal behaviors. This can severely increase the implementation costs.

Depending on our needs, we may prefer one method over the other. As mentioned earlier, the choice of features used as inputs to the machine learning model plays an important role in the detection accuracy of the model as well as the complexity of the model. In the following section Section 2.4.4, we present some feature extraction methods used to select the most appropriate features for the malware classification problem.

2.4.4 Feature Extraction

When using machine learning, we must be careful about the information we give to the model. The information we input must provide the model with relevant information that will allow us to build an efficient model and also reduce its complexity. To find the most prominent features to be inputted to the system there exist to ways:

- Either to perform an analysis of the SOTA, to see what other works use as inputs for their detection problem.
- Either apply mathematical methods to identify the most relevant source of infor-

mation among a big set of possible inputs. This method can be time-consuming as the number of features to be tested can be very big, increasing the time required to test every one of them. Further, proper preprocessing might be needed so the mathematical methods provides us with appropriate decision.

In this section, we will analyze the mathematical methods used in our work.

To select the most prominent features, designers should run the same set of normal and malware applications for all possible information sources in the system. In the case of HPCs, there may be 60-100 different HPCs depending on the target system, and moreover, they cannot be measured simultaneously. Since there are a limited number of HPC registers per CPU core, designers should configure the system with a limited subset of hardware events per run. This limits the designer's ability to choose feature extraction algorithms that use the entire set of different features at once to eventually select a limited subset. One such example is Recursive Feature Elimination (RFE), which is used in [95]. In RFE, designers extract all HPCs at once and inputs them into the target classifier. Then, they recursively trains and eliminates HPC features that do not provide much information for the final classification. In this way, it is finally left with the desired number of hardware events that provide the best classification results. However, such a method can only be used in cases where all features can be extracted at once, for example a custom CPU target. Since HPCs can only be extracted in a limited number and, moreover, each pass of the normal-malware dataset is not deterministic, this task becomes unrealistic for HPC-based feature extraction.

In other cases, the next step is to calculate the amount of information that an HPC hardware event provides by considering the extracted values and the target label.

Pearson Correlation Pearson's correlation coefficient is a widely used feature extraction method in many SOTA works [78], [98], [116]. Pearson's correlation coefficient is a technique that compares the linear relationship between the target label and the extracted measurements and provides a value in the range [-1, 1]. A value of 1 represents a positive linear correlation and -1 represents a negative linear correlation. A value close to 0 means that the target value and the measurements are uncorrelated. In simple words, a positive linear correlation means that when the target value increases/decreases, the measurements also increase/decrease, and a negative correlation means that when one of the labels or measurements increases, the other decrease.

The logic behind using correlation for feature selection is that the most prominent features are highly correlated with the target label. It also allows us to check whether the features are correlated with each other. If two features are correlated, only one of them

gives us the same information as both of them. In this way, we can remove one of the two correlated features and replace it with another, increasing system coverage.

Pearson Correlation Coefficient is computed as follows:

$$\rho(i) = \frac{\text{cov}(X_i, Y)}{\text{var}(X_i)\text{var}(Y)} \tag{2.2}$$

Where $\rho(i)$ is the pearson's correlation coefficient for hardware event i, X_i is the extracted HPC measurements for hardware event i, and Y is the target label, i.e., "1" for malicious and "0" for normal. $cov(X_i, Y)$ measures the covariance between the HPC measurements and the target label, and $var(X_i)$ and $var(Y)$ measure the variance of both the HPC measurements and the target label, respectively. By ranking the returned pearson's correlation coefficients for each hardware event, we keep those with the highest score. At this point, we also check if two or more hardware events among the selected events with the highest ranking are correlated with each other. If this is the case, we remove the least prominent event and replace it with the next one in the ranking.

Mutual Information Mutual Information (MI) is not as widely used feature extraction technique as pearson's correlation coefficient in SOTA. For malware detection using HPCs in SOTA, we find a handful of papers that use this technique [80]. Using pearson's correlation coefficient, we hypothesize that the two variables (HPC measurements for hardware event i and the target label) have a linear relationship. However, this is not always the case in real-world scenarios, as there may not exist a strong linear relationship between the target label and the HPC measurements. On the other hand, MI is a technique that can detect any type of dependency in the dataset i.e., linear and non-linear [117].

The MI $I(X, Y)$ between two random variables X, Y is defined as the amount of the information we can learn on one of them when observing the other and can be calculated using the following equation:

$$I(X_i, Y) = H(X_i) - H(X_i|Y) = \sum_{x \in \mathcal{X}_\setminus} \sum_{y \in \mathcal{Y}} p_{X_iY}(x, y) \log_2 \frac{p_{X_iY}(x, y)}{p_{X_i}(x) p_Y(y)} \tag{2.3}$$

where p_{X_iY} is the joint probability distribution function of X_i and Y. $H(X_i)$ represents the entropy of the variable X_i which can be understood as the average amount of information one can get by observing a realization x of X_i.

$$H(X_i) = \sum_{x \in \mathcal{X}_\setminus} p_{X_i}(x) log_2(\frac{1}{p_{X_i}(x)}) \; ; \; H(X_i|Y) = \sum_{y \in \mathcal{Y}} p_Y(y) H(X_i|Y = y) \tag{2.4}$$

In the context of side-channel leakage assessment the MI $I(Z, X)$ between the sensitive variable Z and the trace X is computed. In our problem case, we want to compute the amount of information $I(X_i, Y)$ that we gain to predict the target application label Y based on the execution trace X_i for the hardware event i. To compute the MI between the target label and the HPC measurements of k-nearest neighbor, we use the NPEET library [118].

In our classification problem, extracting features with strong linear relationships can benefit the simple linear classifiers, but MI can be more general by using nonlinear models that are able to learn more complex nonlinear relationships from the dataset. In any case, it is very important to select relevant features. Selecting the most prominent HPC events can result in high detection accuracy and a low false positive rate. In cases where more complex models are needed that are capable of learning more complex behaviors, relevant HPC events can lead to reducing the complexity of the model by, for example, reducing the number of hidden layers and neurons. This directly impacts decision latency and implementation costs in terms of performance and memory overhead.

2.5 Summary

In this chapter, we have presented the different malware families that pose a threat to IoT devices. We presented the different software and hardware attacks, and we analyzed a new category of software attacks that target hardware vulnerabilities. Since these attacks directly exploit hardware, software mitigations either incur high overhead and hardware mitigations can only be deployed in future CPU architectures. This increases their severity as they can extract sensitive information exploiting side-channels, escalate privileges, etc. As we target IoT devices and attackers use more and more of these attacks against them, security solutions need to be proposed. In SOTA, we have been more concerned with solutions that attempt to detect the malware, as detection can be applied directly to the vulnerable systems. We analyzed that static analysis can efficiently detect malware, but is vulnerable to new malware and techniques such as obfuscation, where the attacker slightly modifies the malware code to change its signature. Dynamic analysis techniques, on the other hand, can monitor the system at runtime and extract information from various sources, such as API calls or HPCs. But despite the accuracy of API calls in software attacks, they are insufficient in detecting SATHV because they do not use abnormal API calls. For this reason, security developers prefer HPCs that allow them to detect all malware subfamilies because both SATHV and software attacks use hardware components in a different way than normal applications.

In the SOTA, we find several works that use ML and HPCs for malware detection. Two implementations are used, either the local implementation of the detection mechanism on the device or the remote implementation in a remote system. When the detection mechanism is implemented locally, we can quickly detect the malicious actions. This is because when a remote system is used, the local system has to transmit the data and wait for the remote system's decision, which increases the detection time. On the other hand, detection solutions that are implemented locally must not be too complex so that they impose low overhead on the local device. However, we have seen that simpler MLs algorithms can accurately detect malicious activity, but tend to have a high FPR. This can result in a large overhead, as the system must take appropriate actions when an alert is raised. If the FPR is high, these actions may be taken too frequently, increasing the overhead. On the other hand, using more complex solutions such as LSTM, CNN, ensemble ML or a different model per attack, can increase the detection rate and decrease the FPR. The price is higher implementation costs, which can be limiting in the case of IoT devices.

Since we do not have the flexibility to implement complex solutions locally, such solutions can be implemented remotely where resources are not an issue. However, apart from slow detection time, this solution also increases communication bandwidth as hundreds of IoT devices can send Mb of extracted data flooding the network. A solution proposed by Wang et al. [98] recognized that complex solutions are not feasible locally and proposed a method to reduce the data that needs to be sent over the network. The authors proposed a method to compress the data locally and later decompress it remotely. However, the authors found that as compression increased, accuracy decreased and performance overhead increased linearly with the amount of data being compressed. Thus, they were only able to compress the data by 20-30%.

Since the Internet of Things has significant limitations, detection solutions targeting these devices must take into account all of these limitations in order to be adopted. In the next chapters, we analyze how this is feasible.

3

Local SATHV Detection Using HPCs and ML

SATHV pose a major threat to systems. Mechanisms for detecting this class of attacks have been proposed. These mechanisms use information extracted from the system to decide whether malicious applications are present or not. However, the proposed mechanisms often do not study all attack variants, which may lead to false security guarantees, as the attack variants may exhibit different behaviors than those studied. This could allow attackers to bypass the system's protections. Since studying all SATHVs and all variants can be time consuming, security researchers will first identify sources of information already used in other SOTA solutions that they can use to differentiate attack and normal behaviors. In this section, (i) we present and classify sources of information used by other SOTA works for SATHV detection, (ii) compare it with the practical information obtained in this work, and (iii) finally propose a solution for detecting eviction-based SATHV in ARMv7-based systems.

3.1 Motivations of the Work

The main motivations and problematics that have driven the research [119] presented in this chapter are the following:

- Security mechanisms that rely on analyzing system behavior to detect malicious applications require information extracted from the system. Extracting these information sources from the SOTA and classifying them according to certain criteria could improve the efficiency of the solution.
- Then we questioned if the theoretical information listed previously would allow us to accurately detect SATHV in our targeted platform and if practical experimentation would confirm these choices.
- The next problematic was the limited threat model of the SOTA solutions. Considering a limited subset of feasible SATHV in a targeted platform, SOTA detection mechanisms allow the attacker to damage the system or bypass the security mechanism.
- Our final motivation is the ability of attackers to create more malicious applications that can bypass security mechanisms based on evasive techniques. This experimentation will allow us to propose a more robust detection mechanism that takes this possibility into account.

3.2 Theoretical Side-Effects for SATHV Detection

As we mentioned in Section 3.1, security solutions based on malware detection need information they can extract from the system to predict whether the current application is malicious or normal. We refer to this information as side-effects.

Side-effects definition

The term *side-effects*, used throughout our work, includes all signals, configurations, and modifications of the system or a combination of the aforementioned that change their nominal value or range of values due to the execution of attacks alongside legitimate software [120].

That is, if a signal_1 during an attack has a value A greater than the nominal value B, we may be able to detect the attack. However, there will be cases where an increased value of a signal is not sufficient to distinguish between nominal and malicious behaviors. In such cases, a second signal_2 during an attack could have a value C greater or less than the nominal value D, which in combination with signal_1 provides information to successfully detect an attack.

The selection of side-effects is critical to maximize the detection performance of a detection mechanism. For example, if a side-effect does not show a significant difference between normal and malicious applications, it is very likely that it is not a good indicator.

As security researchers who aim to propose a detection mechanism, we first look for solutions in SOTA that already cover part of our problem. Security researchers are likely to use side-effects from SOTA solutions first for three reasons:

- Using side-effects already presented or tested in the SOTA can reduce the necessary development costs because a security engineer will spend less time searching for the most appropriate side-effects since they already know the benefits/detection capabilities associated with those side-effects.
- Second, security engineers may not have the appropriate background knowledge of targeted attack vectors and may not be able to interpret whether the side-effects they observe are good or bad indicators of an attack. On the other hand, SOTA solutions can provide the necessary information and justification.
- From the available pool of HPC events, designers will seek to select the most appropriate ones to minimize the design cost of the resource-constrained system. Increasing the side-effects inputted to the detection mechanism increases the implementation cost and system overhead.

It is in our main interest to explore what side-effects other researchers have used in implementing their mechanisms and how we can use them to design ours. To this end, we conducted a survey listing and classifying the various theoretical side-effects [120]. The goal of the survey was to introduce SATHV and their side-effects on the targeted system, so that researchers can refer to it and identify if any of the side-effects are appropriate for their problem.

In Table 3.1 we list the most relevant side-effects found in SOTA detection mechanisms and attacks [5], [82]–[85], [121]–[130]. We have also tried to list and categorize the side-effects depending on the relevant components (software or hardware). For exam-

ple, a side-effect due to a hardware component refers to the abnormal behavior of the hardware component during the attack and can be extracted using HPCs, such as the number of branch mispredictions. A side effect due to a software component, on the other hand, means that we cannot extract it from the hardware, but must instead use software tools such as the Linux *perf* library, or the side-effect is due to software, such as the Signal Segmentation Violation (SIGSEGV) triggered by the OS. For example, we list the most relevant Cache memory, OS, Memory controller side-effects etc.

In addition to listing all side-effects, we have classified side-effects in [120] using a list of criteria that we have defined. This classification can help researchers referring to our work to narrow down the list of side-effects to use. The defined criteria are as follows:

- *Hardware or Software side-effect:* This criterion informs us of the nature of the side-effect. If it is a software side-effect, it might not be available for a hardware implementation, such as the Page Fault Miss Ratio. On the other hand, a hardware side-effect may be inaccessible by a software mechanism because the software lacks privileges or access to the hardware itself. However, the nature of the side-effect is critical to the implementation of a detection mechanism.

- *Source:* This criterion defines the source of the side-effect, which in our case is a register, interface, or file. The side-effect could be a value stored in a register whose value we need to check to determine if it is nominal or abnormal, or an interface event such as a triggered interrupt that causes a change in the system. The observability of the side-effect in the register or interface is debatable. Some registers or interfaces are easily observed with a debugger, while others are hidden from unauthorized users.

- *Used in an existing detection mechanism:* This criterion as mentioned earlier, could benefit researchers indicate whether the side-effect is already used by an existing SOTA detection mechanism. For example, if a side-effect is used in several detection mechanisms, it could be because it is easy to acquire or strongly correlates with attacks. On the other hand, a side-effect may not have been used because it is difficult to obtain, for example, because a software implementation does not have the necessary mechanisms to extract information from the hardware. Alternatively, the side-effect may correlate weakly with the attacks, making other side-effects more appropriate. It should be noted that the PMU can only be used to count a certain number of HPC events, even though the options of events to count are numerous. This is because the CPU core restricts the number of HPC registers used at the same time. The Instruction Set Architecture (ISA) can specify numerous HPC events, but the cost of implementing special-core registers to

count these events is high. For this reason, vendors of CPUs are limiting their implementation. For example, ARM cores such as ARM Cortex-A9, A72, A53 used in multiple IoT devices implement only six HPC registers per CPU core. In addition, if a researcher uses Artificial NNs, the complexity of the system increases with the number of inputs used. These are some reasons why side effects must be chosen carefully to simplify the complexity of the ML algorithms used by designers.

- *Other:* Availability refers to the moment when the side-effect becomes available. The side-effect can occur during the execution of the attack or after the attack is completed. If we can explore the side-effect during the attack, we may be able to prevent the attack from succeeding with its malicious actions. On the other hand, if the side-effect is not available until after the attack is completed, we may be able to detect the attack, but our system is compromised. The final criterion is the effectiveness of a side-effect in detecting an attack. A side-effect alone may not provide us with sufficient information to detect an attack, and we might need more sources to make an accurate decision.

Table 3.1 summarizes the SOTA side-effects, and we believe it can help a security researcher to identify some side-effects that can serve as a starting point for their experimentation. To illustrate the importance of this, we present the following example related to the development of one of our projects [131]. In the iMRC [131] project, we used a VexRISCV processor to implement an Integrated Monitoring and Recovery Component for embedded systems security that uses information from the HPCs sent to a remote server to assess and detect malicious applications. The RISCV ISA specifies the HPCs, but only a few of the available RISCV cores implement them in hardware. Since VexRISCV does not implement any HPCs events and implementing all necessary HPCs events in hardware can be a time-consuming and costly process, we referred to Table 3.1 to implement HPCs events already used in SOTA for SATHV and malware detection.

Table 3.2 presents a summary of the side-effects already used by SOTA detection mechanisms and mentioned in Table 3.1. The taxonomy table classifies these side-effects based on the criteria defined above. Researchers and designers can thus compare the side-effects, find out where they are employed, what attacks cause them, and finally select those that best fit their detection problem. The side-effects are listed in the second column and the defined criteria in columns 3-6. Using this representation, we can make some interesting observations. The side-effects can be SW or HW. Depending on the nature of side-effect, as mentioned earlier, a detection mechanism may not have access to it. SIGSEGV, for example, is difficult to obtain from a HW implementation. Furthermore, because SIGSEGV is a side-effect that only becomes visible after the attack, it is

			Attacks						
			Memory				Transient execution		Power, Frequency
			Row-hammer	CacheCSA	DRAMA	DMA	Meltdown	Spectre	Clkscrew Voltjockey
		Side-effects							
PMUs	Cache	Cache misses	X	X	X		X	X	X
		Cache hits	X	X	X				X
		Cache accesses	X	X	X			X	X
		LLC Loads					X	X	
		Cache Miss Ratio	X	X	X	X	X	X	X
	CPU stats	Number of executed instructions		X					X
		Number of clk cycles		X					X
	Branch controller	Return Branch taken	X	X				X	
		Speculative Return Branch Taken		X				X	
		Branch Mis Predictions	X					X	
	Bus stats	BUS STATS	X		X	X			
	TLB	DTLB Miss Ratio	X	X					
		Use of Cycle Counter		X	X		X	X	X
	OS	SIGSEGV					X		
		Low page fault miss rate	X	X					
	Virtual to physical mapping translation or reverse engineering	Access proc/PID/pagemap, use of transparent huge pages	X	X	X				X
	Interrupts	Interrupt disable	X	X	X				X
re-order		CPU instruction re-ordering disable	X	X	X				X
	Memory Controller	Memory controller access re-order	X		X				
		Num_Mem_ADDR[1]	X		X				
	Synchronization	Time Stamp Counter (TSC) for synchronization			X				

[1] Number of total Memory Accesses to the same memory cells in a time window chosen by the designer as a threshold for attack detection.

Table 3.1: List of side-effects induced by different attacks

	Side-effects	Criteria				Attacks
		HW/SW	Source	Detection Mechanism	Other	
Cache	Cache misses	HW	Reg	[82] (L3 misses), [83] (L3-L2-L1 misses), [126] (LLC misses)	Availability[1]	Rowhammer, CacheCSA, Meltdown, Spectre, DRAMA
	Cache hits	HW	Reg	[82] (L2 hits)	Availability[1]	Rowhammer, CacheCSA, DRAMA
	Cache accesses	HW	Reg	[82] (L3 accesses), [126] (LLC references)	Availability[1]	Rowhammer, CacheCSA, Spectre, DRAMA
	Cache Loads	HW	Reg	No	Available only in some Intel platforms	Spectre, Meltdown
	Cache Miss Ratio	HW	Reg	[85], [132]	Available as complex event[3]	CacheCSA, Rowhammer, Spectre, DRAMA, Meltdown
TLB	DTLB Miss Ratio	HW	Reg	[132]	Available as complex event[4]	CacheCSA, Rowhammer
Instructions	Number of Executed Instructions	HW	Reg	[82]	Availability[1]	CacheCSA
	Instructions Per Cycle (IPC)	HW	Reg	[83]	Availability[1]	CacheCSA
Branch Controller	Speculative Return Branch Taken	HW	Reg	[83]	Availability[1]	CacheCSA, Spectre
	Return Branch Taken	HW	Reg	[126]	Availability[1]	Spectre, Rowhammer
	Branch Mis Predictions	HW	Reg	[126]	Availability[1]	Spectre, Rowhammer
OS	SIGSEGV	SW	Kernel Interface	[121]	Available after the fault	Meltdown
	Page Fault Miss Ratio	SW	Kernel Interface	[85]	Availability[1]	CacheSCA, Rowhammer
Bus	Num_Mem ADDR	HW	Interface	[123], [125], [129], [5]	Efficient to detect DRAM attacks	Rowhammer, DMA, DRAMA
	BUS_TRANS	HW	Reg	[130]	Efficient to detect DRAM attacks	DMA, CacheSCA, Rowhammer, DRAMA
Units	SPM_PMIC	HW/SW	Reg	No	Availability[2]	Clkscrew, Voltjockey
	Phase-Locked Loop (PLL)	HW	Reg	No	Availability[2]	Clkscrew, Voltjockey
	Operating Performance Points (OPP)	SW	DVFS OS -file	No	Available immediately OS file	Clkscrew, Voltjockey
	Digital Thermal Sensors (DTS)	HW/SW	Reg Interface	No	Availability[2]	Thermal monitor SCA
	Interrupt disable	HW	Reg	No	Availability[2]	Rowhammer, CacheCSA, DRAMA, Spectre, Meltdown, Clkscrew, Voltjockey

[1] Available to export only if selected.
[2] Depends on frequency with which the nominal register value is checked.
[3] Can be computed as Cache misses (div) Cache Accesses.
[4] Can be computed as DTLB miss (div) DTLB Accesses.

Table 3.2: Classification of side effects

not an ideal candidate for monitoring. In contrast, side-effects that can be accessed via registers can be accessed more quickly directly via the HW than via SW. The table lists some detection mechanisms based on side-effects that researchers can refer to in order to investigate how they are already being used.

All side-effects presented in Table 3.1 and Table 3.2 are explained in detail in [120]. It analyzes why each attack causes the side-effects, which can help researchers go deep into the exploited vulnerability and the means to exploit the system.

3.3 Practical Side-Effect Evaluation

In Section 3.2, we have presented a list of side-effects that security researchers can refer to when faced with a SATHV detection problem. We have also presented a taxonomy of these side-effects that may be beneficial for researchers. Since we experimented with the same detection problem as two of the works included in our survey [85], [132], we referred to the list of side-effects used by these two detection mechanisms as a starting point and present our experimental results. Our SATHV libraries include CacheCSA and Rowhammer as [85], [132].

The platform we use for our experiments is a ZYBO Z7, a Zynq-7000 ARM Field Programmable Gate Array (FPGA) System on Chip (SOC) development board. It is equipped with a 667MHz dual-core Cortex-A9 processor with 1GB of DDR3L memory. The system is equipped with a Debian GNU/Linux 10. The instruction set is based on ARMV7. However, in order to perform CacheCSA and Rowhammer on our experimental platform, we need to use eviction-based attack vectors rather than flush-based attack vectors. This choice is necessary because the *flush* instruction is not available in user-space, as it is in ARMv8 and Intel architectures. As CacheCSA we use Evict+Reload [133] and eviction-based Rowhammer [50]. We were also interested in studying eviction-based SATHV attack vectors instead of flush-based, since they are applicable on all architectures. This is because they do not require the existence of the *flush* instruction in the ISA. Further, as some systems restrict the use of the *flush* instruction to user-space as a countermeasure to CacheSCA, eviction-based attack vectors are an alternative that an attacker could use and that researchers should always consider in their threat models. Finally, our normal application libraries consist of the MiBench [134] and PARSEC [135] libraries. We chose the MiBench suite because it is intended to represent the spectrum of embedded applications used in industry. On the other hand, PARSEC focuses on the application areas of finance, computer vision, physical modeling, future media, content-based search, and deduplication. Since we focus on resource-constrained sys-

tems rather than desktop environments, these two suites contain applications that are more representative of our target devices.

The main motivations behind this experimentation [27] are the following:

- How scalable is a detection mechanism proposed for a specific CPU architecture? Could we reuse the same side-effects regardless of architecture?
- If we do not consider all possible attack variants in the threat model, does this pose a major security threat to a system in which a detection mechanism is in place?
- Do eviction-based attack vectors cause different side-effects than flush-based attack vectors and why?

3.3.1 Effectiveness of Proposed Solutions on our Platform Using an Extended Set of SATHV

Eviction techniques attempt to remove the targeted address from the cache without the aid of cache maintenance instructions such as the *flush* instruction. When an attack vector uses the *flush* instruction to remove a target address, this task is simplified because the attacker only needs to specify the target address and the *flush* instruction will directly remove it from all cache levels. On the other hand, eviction-based attack vectors need to find a specific set of addresses that will map in the same cache line as the target address. If two addresses are mapped to the same cache line, then there is a probability that both will be placed in the same cache cell, resulting in the CPU removing one address to place the other. The attacker's goal is to create a eviction set that removes the target address with the highest probability. The way addresses mapped to the same cache line are stored is defined by the cache replacement policy. ARM uses a random-replacement policy, meaning that it randomly selects an item from the cache and replaces it with the new element. Due to the random replacement policy in ARM caches, more addresses than the size of the cache line are needed to construct an eviction set because of the probability that two addresses in our eviction set will be placed in the same cache cell. Therefore, eviction takes more time than simply "flushing" an address. For example, in Raspberry Pi3 Model B+ flushing an address from the cache takes around 4.28ns, and in contrast the quickest eviction strategy using the libflush [136] library requires 350ns to remove the targeted address. In the following, we will show that this difference greatly modifies the expected side-effects.

Attackers can be more versatile in using different eviction strategies. Figure 3.1 illus-

trates this. We can observe that the L2 Miss Rate and the number of L2 Misses do not increase in attacks compared to normal applications, as suggested in [85], [132]. In HexPADS [85], the attacks studied were the Flush+Reload [47], Prime+Probe [38], and Rowhammer [50]. The side-effects monitored by this mechanism are the number of instructions executed, LLC accesses and LLC misses. In addition, HexPADS exports from the kernel the status information of each process, e.g., the number of minor page faults, the number of major page faults, and the execution time. HexPADS uses the following detection logic to detect CacheCSA and Rowhammer: If the Cache Miss Ratio is greater than 70%, the cache misses are greater than 100 thousand, and the page fault miss ratio is less than 0.01%, an attack is detected. In our experiments, we did not focus on the thresholds used, but more on the detection logic and side-effect selection. We use boxplots to illustrate our results. Boxplots allow us to visualize the median of the extracted values (horizontal line inside the boxes in Figure 3.1) and with the boxes we visualize the Interquartile Range (IQR), which represents the range in which 50% of the extracted values lie.

As we can see from Figure 3.1a, the median of the extracted values for CacheCSA and normal applications are at the same level. Rowhammer, on the other hand, has a higher Miss Ratio, on average. Moreover, in our experiments with Cache Miss Ratio, we observed that most normal applications present high values during context switches, which can increase FPs. This is because during the context switch, we either need to load the data into the cache for the first time or reload the data removed from the previous processes resulting in an increased Cache Miss Ratio. The Cache misses (labelled as Coherent linefill misses in ARM Cortex A9) decreased for both attacks in Figure 3.1b. To explain, when attackers use eviction techniques, they will observe many Cache hits. Since the attackers perform the attack in a loop to reduce noise and check the different key hypotheses, they have already used the eviction set in the previous attack loop to remove the target address. Since they reload the eviction set in the current loop, many of the addresses are already present in the cache due to the previous loop, resulting in multiple Cache hits. As there are multiple hits due to the use of eviction-based variants, the Cache Miss Ratio decreases and so does the number of Cache misses.

On the other hand, Peng et al. [132] observed that normal applications exhibit also a high Cache Miss Ratio. To reduce the False Positives (FPs), they observed that CacheCSA and Rowhammer cause a low Data Translation Look-aside Buffer (DTLB) Miss Rate. Figure 3.1c presents the DTLB misses and Figure 3.1d presents the DTLB Miss Ratio. The DTLB Miss Ratio is calculated as DTLB Misses divided by the total DTLB Accesses. From Figure 3.1d and Figure 3.1c, we can observe that the median values of the extracted HPCs for the two attacks are indeed lower than the median values of the normal

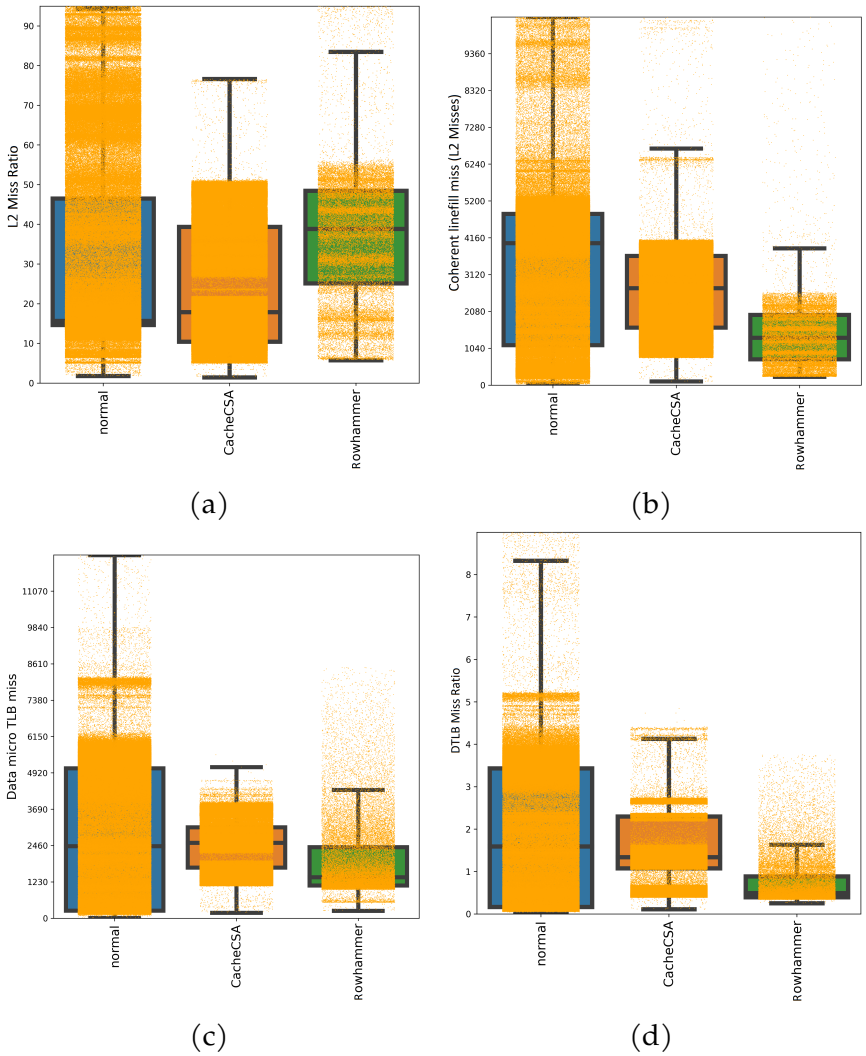


Figure 3.1: (a) L2 Miss Ratio (100% * L2 Misses / L2 Accesses), (b) L2 misses labelled as Coherent linefill miss, (c) Data Micro TLB misses, (d) DTLB Miss Ratio.

applications. However, we can also notice that normal applications have a wide distribution, which could lead to FPs. If we also consider Figure 3.2, we can see that during Rowhammer the DTLB Miss Ratio increases in some points. The main reason for this increase is that during eviction-based attacks, when the attackers try to find congruent addresses to evict the target, they request unused data addresses, which leads to an increased DTLB Miss Ratio since the translation from virtual-to-physical addresses is not cached in the TLB. In our experimental platform, we have observed many MiBench applications exhibiting low DTLB Miss Ratio and a high Cache Miss Ratio. We showcase an FP in Figure 3.3 from MiBench's CRC32 application using the methodology proposed in [132]. As we can see from Figure 3.3, for this application both conditions (higher than a Cache Miss Ratio threshold and lower than the DTLB Miss Ratio threshold) are met for this application, so this application is falsely detected as malicious.

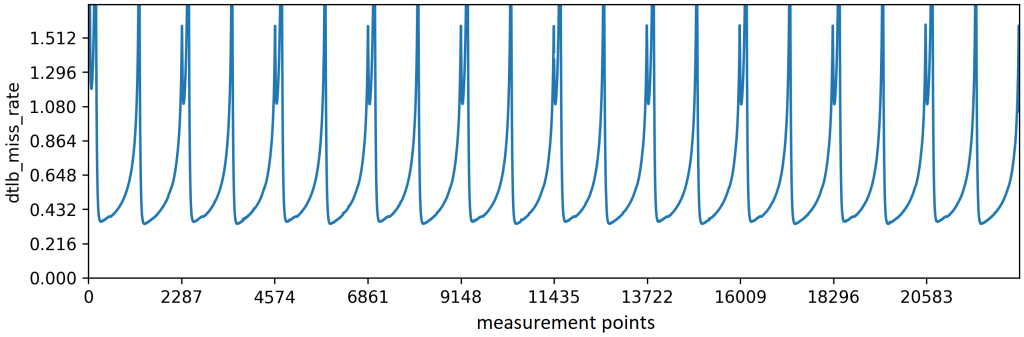


Figure 3.2: 10 round Rowhammer DTLB Miss ratio.

As security researchers facing a SATHV detection problem, we first look at the SOTA work to see if we can use the proposed methodologies. We experimented with two proposed solutions [85], [132], using the same threat model but different experimental platforms. We observed that eviction-based attack vectors can exhibit different behaviors than flush-based attack vectors, for example the cache miss ratio does not significantly increases compared to normal applications as proposed in [85]. Before starting our experiments, we set some questions that motivated our work, and in this section we will try to answer them based on our results.

The first question we set was, *"How scalable is a detection mechanism proposed for a specific CPU architecture? Could we reuse the same side-effects regardless of architecture?"* This question cannot be fully answered because it depends on whether the previous works included all the "sub-attack" variants in the threat model. For example, if we include CacheCSA SATHV in our threat model, but only use a subset of the variants to build and test our model, then the side-effects may not be trustworthy. As we explained earlier, eviction-based attack variants exhibit different behavior than flush-based attack vari-

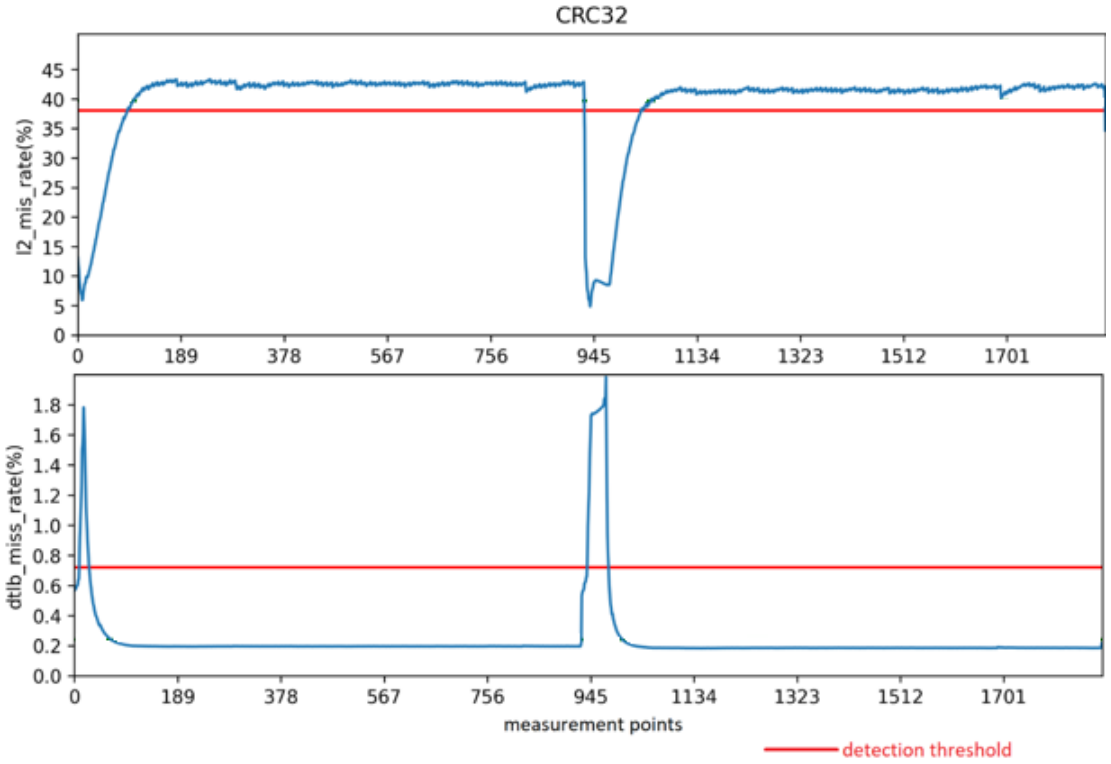


Figure 3.3: CRC32 False Positive using the methodology proposed by Peng et al. .

ants. Since eviction-based variants can be used against all architectures (for example, RISCV and ARMv7 do not implement a *flush* instruction in user-space), not considering these variants when building the detection mechanism may lead to false security guarantees.

The second question we set was *If we do not consider all possible attack variants in the threat model, does this pose a major security threat to a system in which a detection mechanism is in place?* As mentioned earlier, a security mechanism configured for only some attack variants may provide false security guarantees. For example, an attacker may use eviction-based attack variants to try to bypass a detection mechanism that has been designed and tested only for flush-based attacks. This can be a potential security vulnerability.

The third and last question we set was the following: *Do eviction-based attack vectors cause different side-effects than flush-based attack vectors and why?* The answer is yes. Since flush-based attack vectors use the built-in *flush* instruction to perform the attack, the attacker only needs to request the removal of the target addresses and the hardware will automatically remove the requested address. This will cause the target address to be removed in the next few clock cycles, allow the victim to execute the sensitive code, and reload the target address to observe if the victim has used it. If the victim used the address, we will observe a hit, and if not, we have a miss. Since we are testing the same

input for different key possibilities, the misses will be more than the hits, resulting in an increased Cache Miss Ratio. For eviction-based CacheSCA, if the attackers load the addresses with the intention of removing the target address, they will observe multiple hits, resulting in a reduced Cache Miss Ratio, which for our platform is on average less than the normal applications Figure 3.1a.

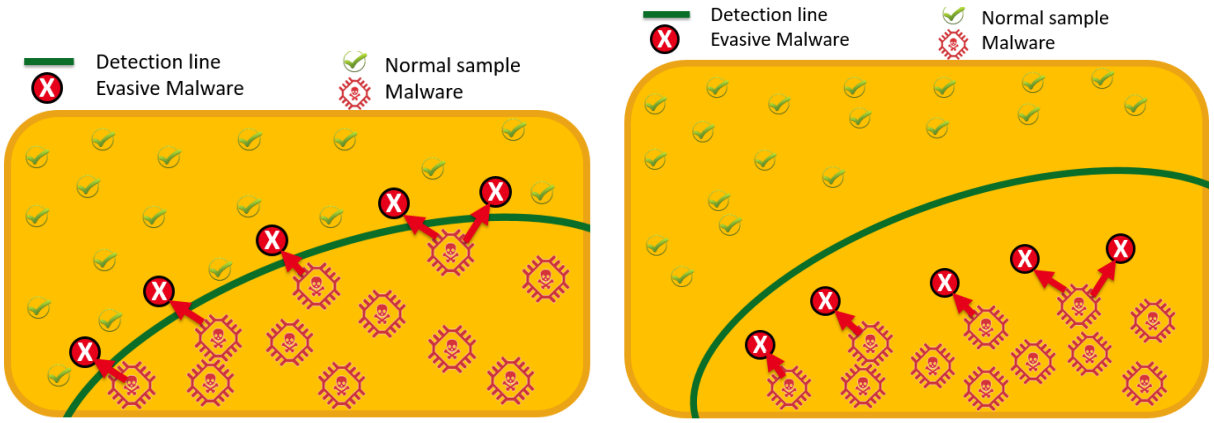
To summarize Section 3.3, when using SOTA solutions as a starting point for their work, security researchers need to be careful about the information they extract, as this could lead them to not detect some attacks. From the results presented, it appears that the selection of side-effects plays an important role in the accuracy of a detection mechanism. In order to find a wider application on different platforms, we need to select side-effects with stable behavior on different platforms and between different attack variants. In the experiments conducted, we found that eviction-based attacks exhibit different behaviors than expected from the proposed detection mechanisms, so they could potentially go undetected on our platform. In order to detect them, we need to take a closer look at their malicious behavior. We propose to search for solutions that cover all different possibilities of an attack vector, which would allow them to gain valuable development time and still reduce development costs, as we mentioned at the beginning of Section 3.2. It is also necessary to extend the attack vectors as much as possible to recognize all possible SATHV behaviors, as we will present in the following Section 3.4.

3.4 Evasive Malware and Monitoring Interval

Even when security solutions are in place, attackers seek ways to circumvent them. One way to do this is to use code obfuscation techniques [19], [137], which lead the malicious applications to be harder to analyze or detect. There are several techniques that a malware author can use to obfuscate the code of a malicious application, such as re-ordering the code, replacing the code, reassigning registers, etc.

The attacker's goal is to change the malware's behavior to be closer to normal behavior. Figure 3.4 presents two attempts by the attacker to create evasive malware. In Figure 3.4a, the detection line separating the two classes has a limited distance from the samples, which means that the attacker has to put less effort to create evasive malware using less obfuscation techniques. On the other hand, in Figure 3.4b the detection line farthest away from the two classes, which means that the attacker must expend more effort to successfully evade detection, otherwise the malware can still be detected. As we will show later, attackers can only use a limited number of obfuscation techniques, as adding additional code can lead to unsuccessful attacks or reduce the success rate.

We will also demonstrate the defenders' ability to create such clear separations, to make the attackers' job more difficult.



(a) Successful Evasive Malware. (b) Unsuccessful Evasive Malware.

Figure 3.4: Normal, Malware, and Evasive Malware and detection line.

In [138], the authors propose obfuscation strategies regarding Spectre. They identify the tasks necessary for the attack to exploit the targeted vulnerability. They also found that disrupting these necessary tasks reduces the success rate of the attack. To this end, they propose the following strategies to create evasive SATHV:

- Inserting instructions between tasks or after all tasks in an attack loop have completed.
- Put the application to sleep between tasks or after completing all tasks in an attack loop.

During Spectre, Meltdown, or CacheCSA, we need to run the attack in multiple loops to successfully extract the sensitive information due to the noise of the system and other applications running in parallel in the system. For example, during Spectre, when we try to read a byte from the restricted memory, we have to read the same byte several times and choose the extracted secret with the highest frequency at the end. The attacker can identify these loops in the original SATHV code and insert the proposed techniques.

We adopted their methodology and adjusted it to all SATHV libraries in our testbench. As shown in [138], inserting obfuscation strategies reduces the success rate of the attack because they insert noise into the execution of the attack. This places a limit on the amount of obfuscation techniques an attacker can add before the attack start to fail. Our evasive SATHV was created by adding NOP instructions, normal applications (piece of codes from the normal application libraries), and *sleep*(). We push to the limit the inserted obfuscation code until our libraries begin to fail. The goal of the evasive SATHV

is to bypass the detection mechanism by modifying the values extracted from the HPCs to be in a range closer to the values extracted during normal application execution, or in an intermediate range between the malicious and normal values.

To give an example of how we can create evasive SATHV, we will describe Rowhammer and Spectre. Rowhammer continuously hammers a set of DRAM rows to succeed a fault before the DRAM refresh interval, which is 64ms. If security implementation extracts HPC values every 100ms, an attacker can perform the attack in a loop for 64ms and then put the application to *sleep* for 36ms. Spectre on the other hand takes 650μs to read a byte from memory on our platform. The attacker can be evasive by performing ten attack loops of $650\mu s * 10 = 65ms$ and sleep also for 35ms reducing the induced side-effects. This leads to a reduction in observed side-effects that could influence the decisions of a ML classifier. Using Figure 3.5, we can observe a scatter-plot of the Data Cache Misses versus Instruction TLB allocations for normal applications, Spectre, and evasive Spectre. As normal applications we use the suites MiBench [134] and PARSEC [135]. As we can see in Figure 3.5, the evasive Spectre values (red squares) are shifted compared to Spectre values (blue triangles). The evasive Spectre values are shifted in a region where only normal values exist, which can confuse a ML classifier.

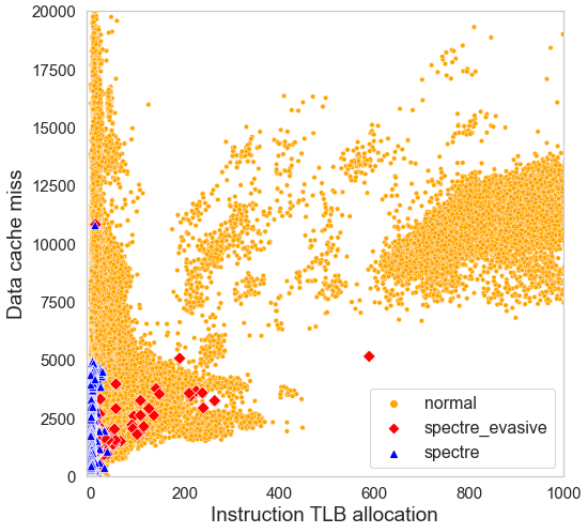
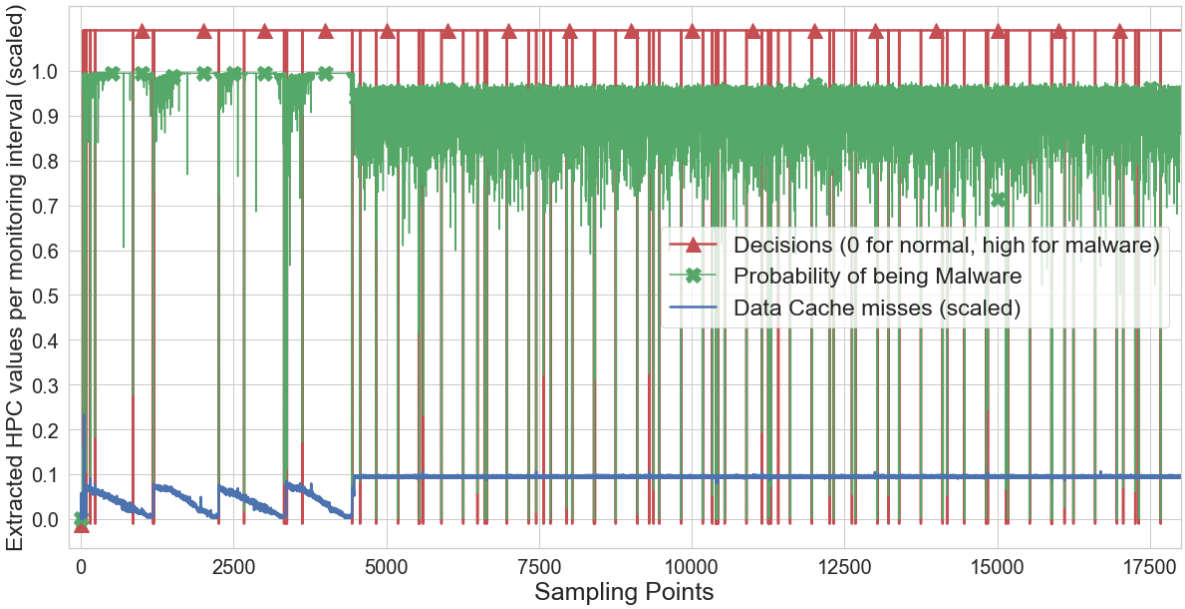


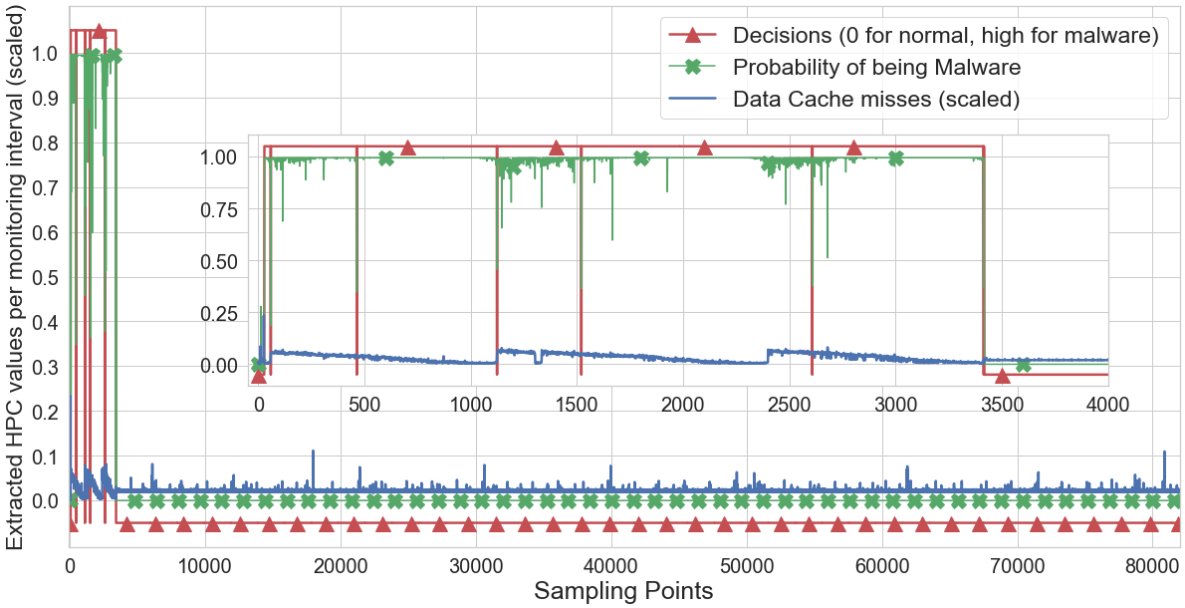
Figure 3.5: Data cache miss versus Instruction TLB allocation for Evasive Spectre.

Monitoring Interval and Evasive Attacks

As we mentioned earlier, the attacker's ability to insert obfuscated techniques depends first on the time required to complete a sensitive task or an attack loop, and second on the time available after the end of the sensitive task and until we extract the HPC values i.e., the monitoring interval. Though, the monitoring interval plays an important



(a) Spectre.



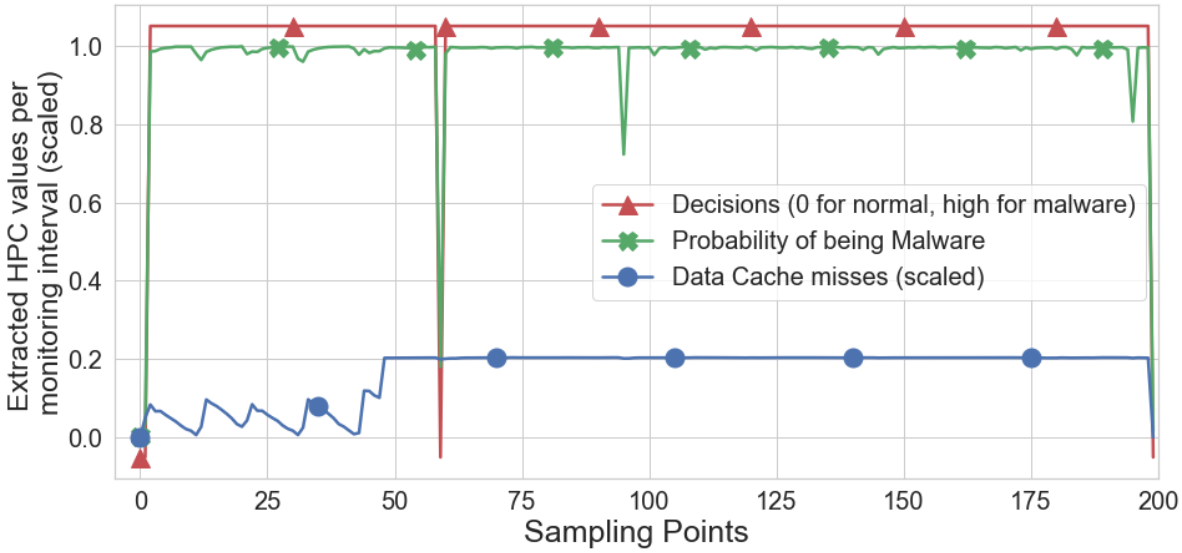
(b) Evasive Spectre.

Figure 3.6: Waveforms and decisions of an MLP classifier using a monitor interval of 1ms.

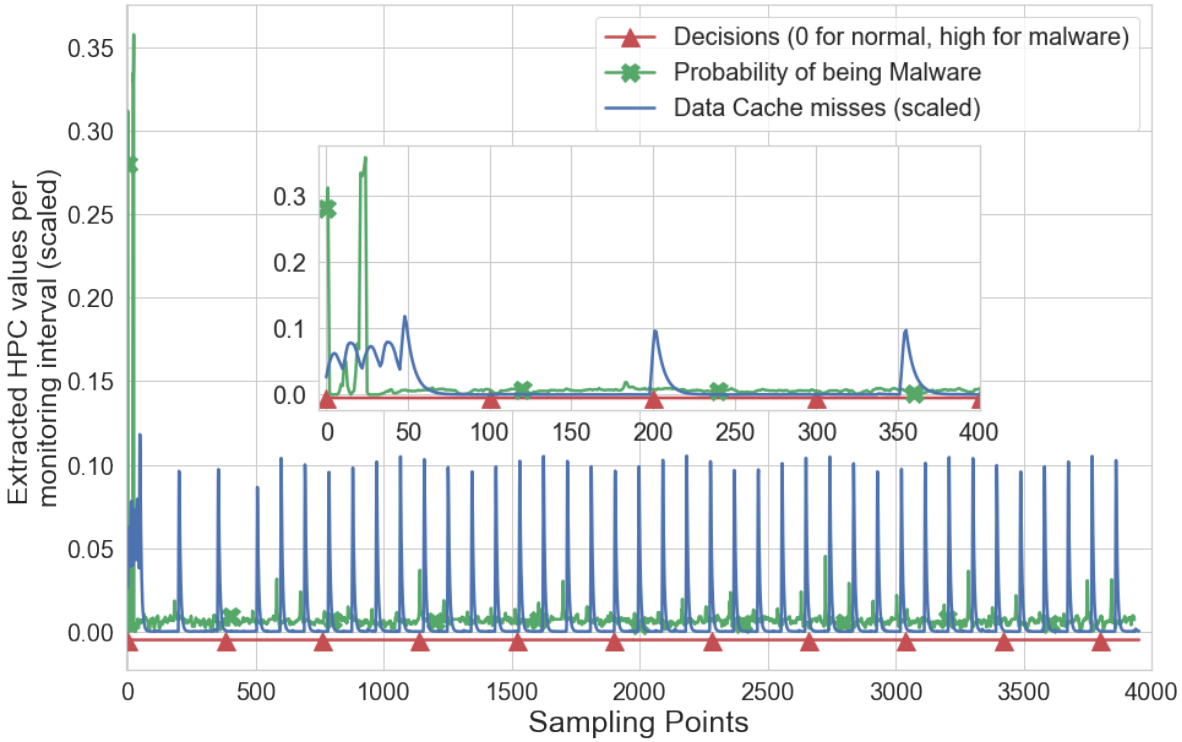
role in the time an attacker has to add obfuscation techniques. Higher frequency monitoring gives the attackers less time to obfuscate the code and reduces their ability to successfully evade detection.

To better illustrate the idea, we present some examples. The following examples we consider as SATHV attack vectors Spectre and Rowhammer and for normal applications the suites MiBench and PARSEC. We present the two examples with a monitoring interval of 1ms and 100ms. The monitoring interval of 100ms was chosen because several SOTA detection mechanisms [84], [86], [87] that use information from HPCs specify it in their configurations. The monitoring interval of 100ms is also the minimum interval at which the utility *Linux perf* allows us to extract the values of HPCs. In Figure 3.6 and Figure 3.7, we show the waveform of Spectre for the HPC event Data Cache Misses (blue line), the MLP classifier decisions (red triangle line), and the probabilities that the samples are malware (green x marked line). In Figure 3.6b and Figure 3.7b we also add a zoom at the start of the execution of the attack.

Figure 3.6 shows the figures of the attacks when the monitoring interval is 1ms. The figure shows in red the decisions of the classifier for the current sample, in green the probability that the sample is malware, and in blue the scaled HPC value. As we can see in Figure 3.6a, the MLP classifier correctly predicts the entire execution of Spectre as malicious. On the other hand, we can see in Figure 3.6b that the MLP classifier fails to predict the entire execution of the malicious application as an attack when evaluating the evasive Spectre. From Figure 3.6b, we observe that after the sample point 3500 the classifier outputs negative i.e., *'0'* decisions (red waveform), as the probability (green waveform) of the samples being malware is close to zero i.e., less than the detection threshold *0.5*. Nevertheless, the classifier correctly predicts the beginning of the evasive Spectre as malicious (samples 0 to 3500 have positive decisions as the probabilities of being malware are greater than the detection threshold 0.5), so we can detect the attack. It is worth noting that the beginning of the Spectre waveform is dedicated to the construction of the eviction sets (execution from sample point 0 to sample point 3500 in Figure 3.6a). Since this part of the attack has several sensitive tasks, it is more difficult to insert enough obfuscation techniques between the 1ms monitoring interval to alter the execution and trick the classifier. On the other hand, this is not the case when we use a 100ms monitoring interval. In Figure 3.7a we see that the classifier correctly predicts most of the execution trace of Spectre as malicious, but completely fails to predict evasive Spectre as malicious in Figure 3.7b. This is because Spectre takes 650μs to read a byte from memory on our platform. This leaves us with about 340μs to input obfuscation techniques if the monitoring interval is 1ms (we read the HPCs every 1ms, but the monitor takes about 10μs to read the values from the HPCs and make a decision



(a) Spectre.



(b) Evasive Spectre.

Figure 3.7: Waveforms and decisions of an MLP classifier using a monitor interval of 100ms.

based on the extracted values). On the other hand, if we use a monitoring interval of 100ms, the available time for entering obfuscation techniques increases, which gives the attacker the opportunity to modify the behavior of the attack. We also note from Figure 3.6b and Figure 3.7b that the execution time of an evasive SATHV increases significantly compared to Figure 3.6a and Figure 3.7a .

Monitoring interval and system overhead Monitoring the system at high frequency allows security designers to strengthen the system against the evasion techniques presented earlier. This comes with the cost of an extra performance and memory overhead, as we need to interrupt normal execution more frequently to read and store HPC values locally when we need to send them to a detection mechanism implemented in a remote system [88], [97]. Since the monitoring interval plays an important role in creating an accurate and effective detection mechanism, we experimented with different monitoring intervals to determine the performance overhead for the system. We extract measurements by running the normal application with and without the monitor running in parallel (Section 3.5.1), and calculate the overhead as an increase in total execution time.

From Table 3.3, we can observe that monitoring intervals of less than 1ms cause an overhead of more than 5% in the local system. We consider that monitoring intervals less or equal than 1ms introduce an acceptable overhead in the local system. But the performance overhead is not the only factor we consider important when choosing the interval. If we monitor the system at a low frequency, we could allow the attacker to damage the system even before we detect him. For example, if we monitor the system at a frequency of 100ms, an attacker running Rowhammer could inject a fault in DRAM and potentially act undetected. Also, the lower the frequency we monitor the system, the more time an attacker has to use evasion techniques. We also identified a parameter that allows an attacker to hide any malicious activity in multi-processing systems. In multi-processing systems, the OS schedules each process to run on the system and share resources with other applications. When an application is scheduled to run, the other applications wait in a queue. Each application that is scheduled runs for a minimum time defined by an OS parameter called *sysctl_sched_min_granularity* in Linux, which defaults to 4ms in our platform. The OS scheduler schedules applications

monitoring interval	overhead
100us	21.08%
300us	10.2%
500us	5.48%
1ms	1.27%
4ms	1%
8ms	0.5%
10ms	0.24%
100ms	0.13%

Table 3.3: Monitoring interval system overheads

for this minimum time only if there are multiple applications in the queue. This is usually the case when more than 5 applications are using the resources of CPU. An attacker can use this to mix the execution of the attack with normal applications by stressing the OS scheduler. If we monitor the system with a monitoring interval of more than 4ms, we could extract measurements that contain side-effects of the attack and normal applications i.e., shifting the extracted values closer to nominal. This could be a serious problem for solutions that use RNNs [76] (which use previous execution traces to predict future ones), as there is a possibility that the malicious applications hiding between the execution of other normal applications will not be detected. Also, we observed that using monitoring intervals of less than 1ms increases the inconsistency between successive measurements, causing noise. This is because we use the *nanosleep* command to create the monitoring interval by sleeping and letting the application execute before receiving an interrupt to read the HPCs after the specified period. Since our monitor relies on interrupts served by the OS, the time it takes OS to serve the interrupt and schedule the execution of our monitor application introduces noise.

> Since 1ms introduces an acceptable performance overhead and allows us to closely monitor applications running at the minimum granularity of *sysctl_sched_min_granularity*, we configured our monitor with the monitoring rate of 1ms.

3.5 MaDMAN: Detection of Software Attacks Targeting Hardware Vulnerabilities

There are many solutions in the literature that use HPCs and ML to detect SATHV. Since the number of side-effects a designer can use to build its detection mechanism is limited to the number of available HPCs in the target system, the proposed solutions only protect the microprocessor from a limited number of SATHV. For example, [85], [132] target only Rowhammer and CacheCSA, [84] is limited to Spectre, [82] targets only CacheCSA, [121] is limited to Meltdown, [122] targets only Meltdown and Spectre, e.g., . In contrast, [76] aims to detect CacheCSA, Spectre, Rowhammer, and Meltdown using LSTM and RNN networks, and manages to detect these SATHV with an F-score of 99.70%. However, since [76] aims to detect SATHV in Server and Desktop environments, it can afford to use complex ML networks such as LSTM and RNN. As LSTM and RNN require multiple processing resources, these solutions may not be the best

solution considering the induced overheads for micro-processors such as ARM cores used in IoT devices.

The motivations for this work [28] are the following:

- Proposed SOTA detection mechanisms are specific to some SATHV and cannot detect all possible SATHV applicable in the system. As the number of available attack vectors increases, attackers can circumvent the detection mechanisms by using attack variants that are not considered by the detection model. Such examples include SATHV that use eviction strategies, as we presented in Section 3.3.

- In SOTA works, evasive SATHV are not considered. We include evasive SATHV presented in Section 3.4 in our threat model. It is important to consider attackers that try to hide their malicious activities.

- Since we are targeting IoT and IIoT systems where resources are limited and we want to respond to attacks as quickly as possible, implementing a local detection mechanism is necessary. Since we cannot use complex ML as in [76] in these resource-limited devices, we need to rely on simpler ML networks. As simpler ML can learn less information compared to more complex MLs, such as an LSTM or RNN, which means side-effect selection and filtering of FPs are crucial.

- SOTA implementations usually consider noiseless environments. We refer to a noiseless environment as running one application at a time. This allows them to use aggressive FPs filtering, such as in [77], which decides whether a FP is present based on the last 16 decision points. On the other hand, if we consider noisy systems, such as multi-processing systems, applications share resources, as explained in Section 3.4. This leads to noise in our measurements due to context switching. In addition, we cannot use aggressive FP filtering because we are not sure if a malware was executed between normal applications, which could lead to filtering the attacks.

For these reasons, we propose MaDMAN (Malware Detector - Monitor, Act, Notify), a SATHV malware detector that collects information from HPCs to detect a large set of SATHV on an ARMv7 platform. MaDMAN uses a Logistic Regression classifier. In our threat model, CacheCSA, Rowhammer, and Spectre SATHVs are included. Our detection mechanism detects these attacks with an accuracy of 98.96%, an F-score of 96.3%, and an FPR of 0%. In addition, MaDMAN works in noisy environments and can successfully detect evasive malware.

3.5.1 Methodology

HPC event selection

From previous works, it is possible to detect SATHV using HPCs. We use HPCs to measure the stress that malicious applications induce on microprocessor hardware components. Since we can only afford to implement a simple ML locally to reduce the performance overhead, we need to carefully select the side-effects as mentioned in our motivations. The selection of side-effects is crucial for a simple ML algorithm, as we may not be able to make accurate decisions due to ML's inability to learn useful information. Moreover, the number of HPCs we can monitor simultaneously is limited, which increases the need to choose them carefully. For example, in ARM architectures, only six HPCs can be configured and monitored simultaneously plus the cycle counter that it is a dedicated register.

To select the most appropriate side-effects for configuring and building our ML, and since we can not rely on the side-effects listed in Section 3.2, we use the feature extraction methods proposed in the SOTA [78]–[80]. In these works, the following feature extraction methods are used to find the optimal set of side-effects: Pearson's Correlation Coefficient, Analysis Of Variance (ANOVA), and Mutual Information. These methods use the extracted HPC values and the targeted label i.e., normal or attack sample and return us a score. The higher the score, the more "information" the side-effect can provide us. The downside of this method is that we need to run the experiment for all the available HPCs, which increases the development time significantly. Using the above methodology, we made the following observations and adjust it for our needs:

- The aforementioned feature extraction algorithms look at the extracted HPC values of a side-effect and the targeted label and return us a score. Since there are side-effects that could give us the same information, we need to remove the unnecessary ones. For example, on ARM Cortex-A72 there are the following HPC events: L2D_CACHE _LD, L2D_CACHE _ST, and L2D_CACHE. In our experiments, we observed that the feature extraction algorithms for these HPC events will return a similar score. This is because they provide us with information from the same HW component. If this is the case, we keep the HPC event with the highest score and remove the others since they do not provide us with any additional information.

- Since there exist multiple available HPC events, and each HW component may have multiple HPC events that provide information for each operation, these feature extraction methods may return HPC events that target the same HW compo-

nent but do not belong to the bullet point above. For example, a feature extraction algorithm may return the following five HPC events: L2D_CACHE, L2D_CACHE _REFILL, L2D_CACHE _WB _VICTIM, L2D_CACHE _WB _CLEAN, L2D_CACHE _INVAL. In this case, all five previous HPC events do not fall under the case mentioned in the first bullet point, but all of them provide us with information for the L2 Cache. In this case, if we train our ML classifier with this set of HPC events, it could accurately detect a current set of attacks, but it might not be able to detect SATHV based on zero-day vulnerabilities that do not target the L2 Cache. It could also be the case that attackers try to modify the behavior of the attack on this component to bypass detection. In this case, their task might be more plausible than in the case where the attacker should craft an evasive code to modify more than one HW components. In such a case, we also remove a set of these HPC events as we want more diversity in the choice of SATHV.

- It is in our interest to monitor the operation of several HW components. This gives us a more comprehensive view of the functionality of the system. However, it may be the case that a component does not provide us with much information to distinguish normal execution from that of an attack. In this case, we take the ranking of HPC events scores after feature extraction and select the best HPC event from the different components, and only if the HPC event provides more information than the threshold we set i.e., higher than a score.

After selecting the most appropriate HPC events, we can configure our monitoring system and train our ML.

Monitoring algorithm

MaDMAN is implemented in SW, runs in parallel with applications, and extracts HPC values using assembly code. MaDMAN monitors the system with a monitoring interval of 1ms, as explained in Section 3.4. To reduce noise induced by our monitoring application, we disable HPC counters when our monitor reads HPC values and then make a decision using ML. Once the decision is made, we re-enable the HPC counters and let the applications continue their execution. The monitoring and detection algorithm used is listed in Algorithm 1.

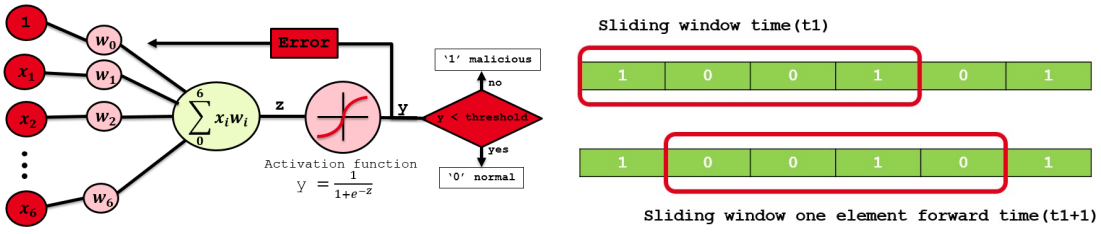
Machine Learning algorithm

There exist multiple normal applications, and each of them triggers different behavior in the system. Ideally, we would prefer that normal applications have a similar HPC side-effect distribution with low variance, and that attacks have distributions that we

Algorithm 1: Monitoring algorithm plus detection using ML

```
Configure HPCs with selected events;
while True do
    reset HPCs;
    nanosleep(monitoring interval);
    // Stop HPC counters from measuring component activity due to our monitoring
       module.
    disable HPCs;
    read HPCs and take decision using ML;
    if malware detected then
        Act ( Reset OR Restore to safe State) AND Notify system;
    end
    re-enable HPCs;
end
```

can easily separate from the normal ones. Since this is not the case, we use ML algorithms to learn how to separate the two classes. In our case, we have two classes or labels, i.e., malicious or normal. In our case, we use supervised ML, as we provide the inputs and the target values. We use a 70%-30% split of training and testing applications. That is, we use 70% of normal and 70% of malicious applications to train our ML and keep the rest for testing. Since the applications vary in execution length and we want to have a balanced length between each application, we use data augmentation techniques [139] to increase the length and undersampling techniques to decrease the length of an application. For our detection mechanism, we chose to use a Logistic Regression classifier because of its simplicity. We can think of Logistic Regression as a single-layer neural network Figure 3.8a. The model receives as input the HPC extracted values (x_i) and outputs a probability that the current sample is malicious (y). If the probability is greater than a threshold we set (0.5), then we output a positive decision *'1'*, otherwise, we output a negative decision *'0'*.



(a) (b)

Figure 3.8: (a) Logistic Regression and (b) Sliding Window Visualizations.

False Positive Filtering Using Exponentially Weighted Moving Average

Logistic Regression is a linear classifier, that is, it attempts to separate the two classes based on a linear hyperplane. Since it may not be possible to separate our classes exactly linearly, we will have FPs and FNs. Ideally, we would like our classifier to have as few FPs and FNs as possible. Since our goal is to detect the attacks, a small percentage of FN samples is acceptable if we can still detect all attacks. The attack code does not perform only malicious actions throughout its duration, but also performs normal operations such as array initialization. Since we classify the entire execution of an attack as malicious, it would be normal to have some FNs considering such normal operations inside the malicious code. On the other hand, it is not acceptable to identify a normal application as malicious and raise a False alarm. We give the following reasons why we try to avoid this:

- When we identify a sample as malicious, we notify the system to take appropriate actions. This could be resetting the system or restoring it to a previous safe state. These actions might be too costly if a FP occurs with high to moderate frequency, such as every 30 seconds to 1 minute.

- The normal application is critical, for example, an IoT insulin pump. In this case, setting the device out of service for a few seconds might be life-critical for the patient.

To filter the FPs, we use a similar technique as in [77], [84] called Exponentially Weighted Moving Average (EWMA). EWMA is an algorithm that allows us to compute a "type" of average in a sliding window Figure 3.8b. From Figure 3.8b, we observe the visualization of the sliding window. The red windows is the time frame where we average the decisions. For example, the new decision at the top window will be 0.5 and to the bottom window 0.25. The advantage of EWMA is that we can compute the average during runtime when new decisions arrive, which means low computational and memory overhead. Further, the weight of a new decision added to the average decreases exponentially over time, biasing the average towards more recent data. This means that the most recent decisions in the window are more important than the older ones. To compute the average in a sliding window of size N as in Figure 3.8b, EWMA requires a decay factor α. The larger the decay factor α, the more the average is skewed towards more recent decisions. To calculate EWMA, we use the Equation (3.1).

$$S_t = \begin{cases} Y_1, & \text{t = 1} \\ \alpha Y_t + (1-\alpha) \cdot S_{t-1}, & \text{t >1} \end{cases} \tag{3.1}$$

where Y_t is the decision at time *t* and S_{t-1} is the value of EWMA in time *t-1*. The final decision is positive (*'1'*) if S_t is greater than a threshold, otherwise it is negative (*'0'*). The size of the sliding window is crucial for the detection accuracy of our local detection mechanism. A small window could lead to high FPs and a large window could lead to high FNs, as it is more likely to miss the malicious behavior in a larger window. To further reduce the noise of our measurements, we also apply EWMA to the HPCs measurements to reduce the high-frequency noise. We will refer to the size of the window for filtering the decisions of the ML as *decision window* and the size of the window for filtering the HPC measurements to reduce noise as *measurement window*.

Evaluation of the Detection Performance

When evaluating our ML detection mechanism we use the following metrics:

- Accuracy measures the number of correct predictions divided by the total number of predictions.
- Precision measures the proportion of positive observations that truly are positive.
- Sensitivity or True Positive Rate (TPR) measures how well we identify TPs.
- F-score measures the global trade-off between precision and sensitivity.
- FPR, refers to the percentage of FPs compared to all positive predictions.
- Receiver Operating Characteristic (ROC) curve is a visualization of the classifier's performance. The ROC curve visualizes the TPR and FPR for different classification thresholds. ROC shows the tradeoff between the TPR and FPR. The Area Under Curve (AUC) is a measure of the accuracy of the model. A model with a ROC closer to the diagonal line (Figure 3.13b the No Skill blue line) is less accurate and a model with 100% accuracy has an AUC of 1 [140]. AUC 1 is a ROC curve with a point $(0, 1)$ indicating 0% FPR and 100% TPR. Another interesting feature we can extract from the ROC curve is the optimal threshold. To find the optimal threshold, we look for the threshold that gives the ROC point with the highest G_{mean}. G_{mean} is a metric that searches the balance between the TPR and the FPR.

Accuracy, Precision, Sensitivity, F-score, FPR, and G_{mean} are calculated using the following equations:

$$Accuracy = \frac{T_P + T_N}{T_P + T_N + F_N + F_P} \quad (3.2) \qquad FPR = \frac{F_P}{F_P + T_N} \quad (3.3)$$

$$precision = \frac{T_P}{T_P + F_P} \quad (3.4) \qquad sensitivity = \frac{T_P}{T_P + F_N} \quad (3.5)$$

$$F_{score} = \frac{2 * (precision * sensitivity)}{precision + sensitivity} \quad (3.6) \qquad G_{mean} = \sqrt{specificity * sensitivity} \quad (3.7)$$

True Positive (TP) is the observation corresponding to the actual attack predicted as attack. True Negative (TN) is the observation corresponding to normal application predicted as normal. False Positive (FP) or False Alarm is the observation corresponding to a normal application predicted to be an attack. False Negative (FN) is the observation corresponding to an actual attack predicted to be a normal application. Figure 3.9 presents the ROC and AUC. To create the ROC, we extract the TPR and FPR for different classification thresholds, starting with a threshold of 0 and continuing to a threshold of 1. A threshold of 0 results in a point at (1,1), which means 100% FPR and 100% TPR, since we detect all normal applications as malicious and all malicious applications are also correctly classified. On the other hand, a threshold of 1 yields a point at (0,0), which means 0% FPR and 0% TPR, i.e., we have no FP, since we correctly predict all normal applications as normal, but also classify all malicious samples as normal. As mentioned earlier, the perfect classifier yields a point at (0,1). Also shown in the figure is the red dashed line indicating a random classifier i.e., we have 50% probability of correctly classifying a sample. Classifiers with a ROC curve below the random classifier are worse and above it are better. Of the three ROC curves presented (blue, green, and brown curves), the blue ROC curve is the best among them. The optimal threshold for the classifier with this ROC curve is the point with the maximum G_{mean}, which we can also see in the figure. Finally, the subfigure at the bottom right in Figure 3.9 shows the AUC.

3.5.2 Results

In the following section, we present our experimental platform, HPC event selection, and MaDMAN. Finally, we evaluate the detection performance of MaDMAN.

Experimental Platform

Our experimental platform is based on the Zybo Z7-20 evaluation board, which contains a dual-core ARM Cortex-A9 processor running at 667MHz. It is equipped with 1GB of DDR3L memory and a Debian GNU /Linux 10. The ARM Cortex-A9 processor is a 32-bit processor core that implements the ARMv7-A architecture. Since in ARMv7 cache maintenance operations can only be executed in privileged modes, we use eviction techniques when needed. Our attack libraries include CacheCSA, Spectre, and

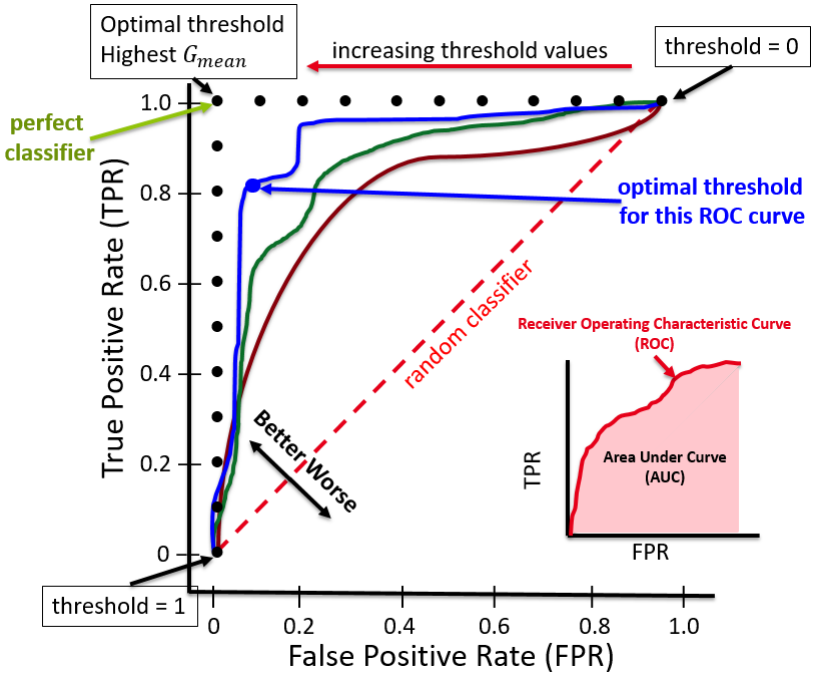


Figure 3.9: ROC Curve and AUC explained.

Rowhammer attacks. CacheCSA includes Evict+Time, Prime+Probe, and Evict+Reload, which target the implementation of AES T-Table. Our attack library includes both one-sided and two-sided Rowhammer attacks. Our libraries for normal applications includes the MiBench and PARSEC test suites.

HPC Event Selection

The ARM Cortex-A9 core implements 55 events, but only allows simultaneous counting of 6 events. As we have already seen, we cannot rely on theoretical side-effects proposed in SOTA for detecting eviction-based attack vectors. On the other hand, we have gained valuable information about the HW components that are most stressed during SATHV execution.

To find the optimal set of hardware events to monitor, we use the feature extraction techniques mentioned in Section 3.5.1. After analysis, the 6 hardware events we select for monitoring are *L2 misses, Instruction cache misses, Data cache misses, ITLB miss, ITLB allocation, and DTLB allocation*. We also calculate two percentages i.e., *the percentage of TLB allocations due to a DTLB request divided by the number of L2 misses* and *the percentage of TLB allocations due to an ITLB request divided by the ITLB misses*. To further illustrate and validate the results of the selected HPC events, we present their boxplots. The boxplots can provide us a first indication of the feasibility of detection based on the targeted HPC events using a linear hyperplane.

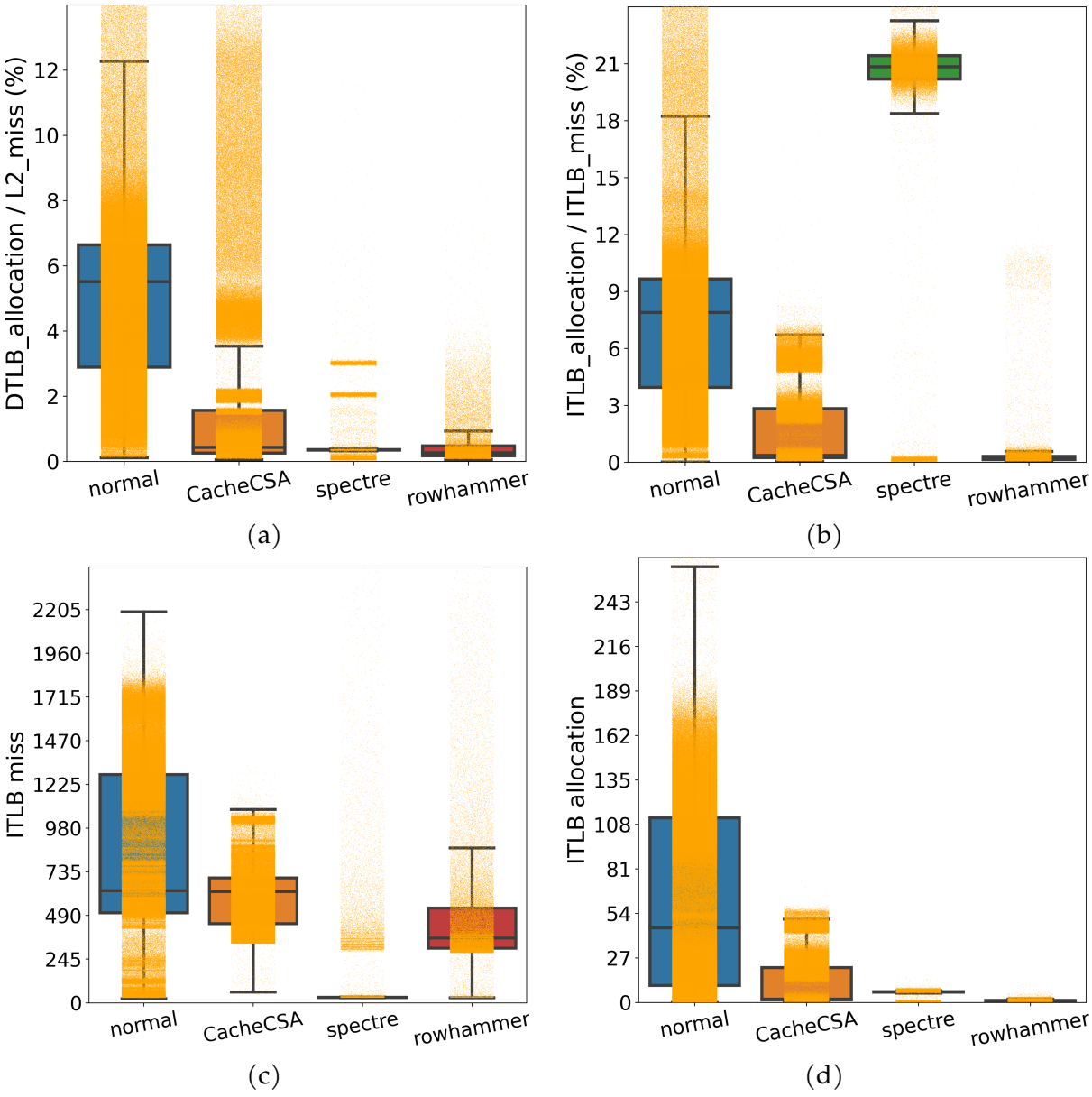


Figure 3.10: (a) percentage of TLB allocations due to a DTLB request by L2 misses, (b) percentage of TLB allocations due to an ITLB request by ITLB misses, (c) ITLB misses, (d) TLB allocations due to an ITLB request.

In Figure 3.10a, we can observe that the median of the percentage of TLB allocation due to a DTLB request by L2 misses is very low for all attack vectors compared to normal applications. Moreover, the boxplots of normal and malicious applications do not overlap. This is explained by the fact that the attacks target specific address ranges and they do not need to bring a new translation to the TLB frequently. When the attack vectors get a miss in all levels of the cache, they observe the miss at data addresses for which the translations exist in the TLBs and are frequently used. Normal applications, on the other hand, are more versatile, and if a miss occurs in all levels of the cache, it is more likely to be a data address that has never been used. The TLB must then allocate the missing translation information. This event can replace the L2 Miss Ratio used in previous works [82], [84], [132] as it shows that malicious code is missing from all levels of the cache and accessing frequently used data addresses.

We also observe in Figure 3.10b that for Rowhammer and CacheCSA attacks, the median of the percentage of ITLB allocations by the ITLB misses decreases. In contrast, it increases for Spectre attacks. For Rowhammer and CacheCSA attacks, this is due to the attacks using a small attack code in a loop. In Spectre, it increases despite the small code because in Spectre we evict all data from the cache, which also removes instructions. If the translation is evicted from the TLB due to the replacement algorithm, when we need to use the instructions of the loop, we need to bring the translation information into the TLB again. This side-effect can inform us how small the instruction code is when we replace the Branch Miss Ratio used in [84].

For the rest of the selected HPC events seen in Figure 3.10 and Figure 3.11, we note that the distributions differ slightly from the distribution of normal applications. The boxplots can only give us a first insight into whether our classification task is feasible with a linear classifier, but they could also lead to a misinterpretation of the feasibility of the task. This is because we need to look at the combination of side-effects and not each one individually.

MaDMAN Implementation

Since we want to keep the implementation simple so as not to impose a large performance overhead on an already limited system like IoT, we chose Logistic Regression as our classifier. We implemented our solution in SW with a monitoring interval of 1 ms, as explained in Section 3.4, to account for evasive malware. We test our implementation in noiseless and noisy environments. In the noisy environment, we stress the system with five applications that simultaneously consume the CPU resources. In this case, OS schedules the execution of each application with a minimum granularity of 4 ms.

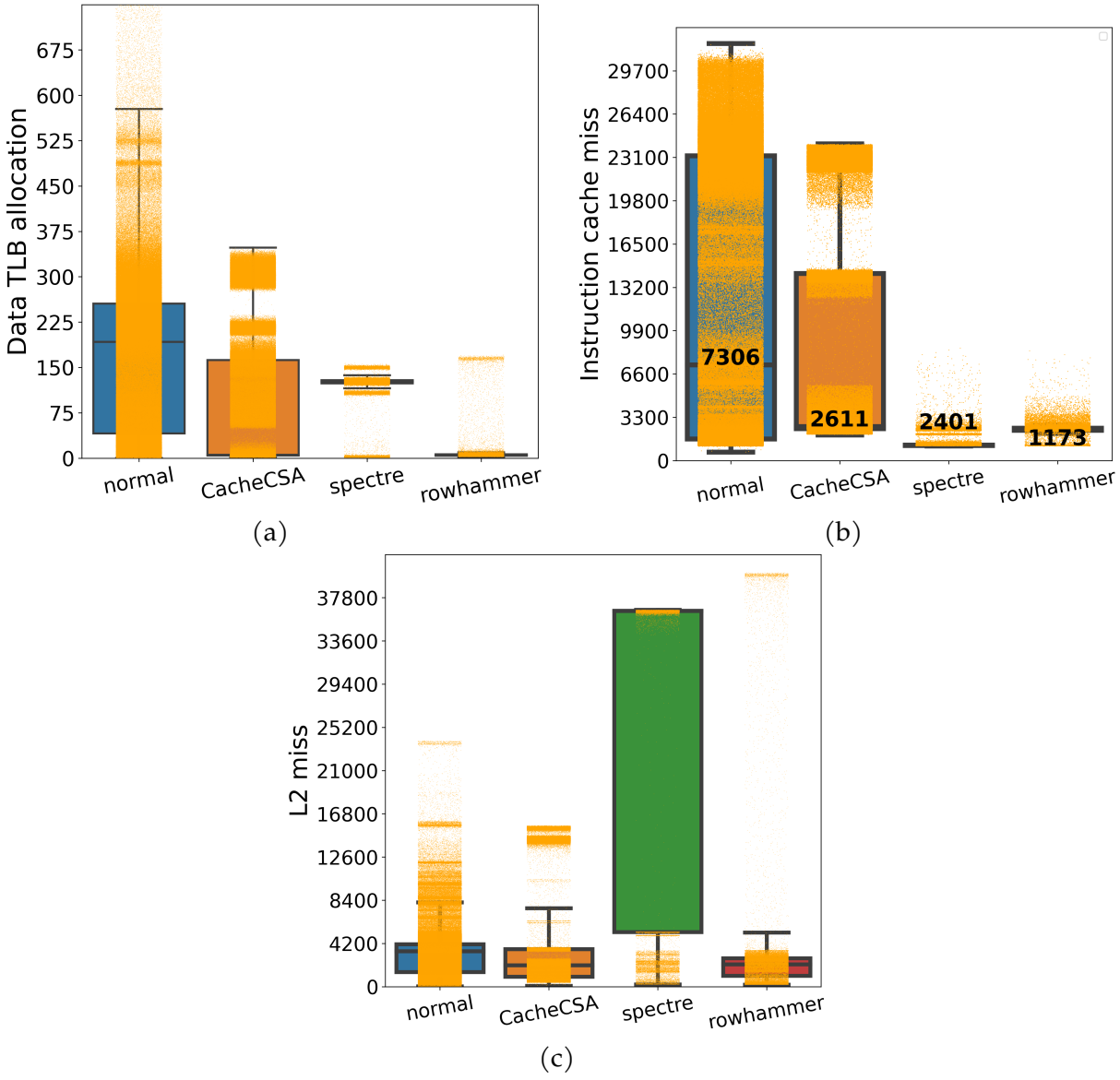


Figure 3.11: (a) Data TLB allocations, (b) Instruction Cache Misses, (c) L2 misses.

Due to the simplicity of the chosen classifier, we use EWMA to filter the FPs. As mentioned in Section 3.5.1, EWMA averages the decisions in a sliding window that gives more weight to the most recent decisions. To choose the decision window, we experimented with different sizes and made the following observations:

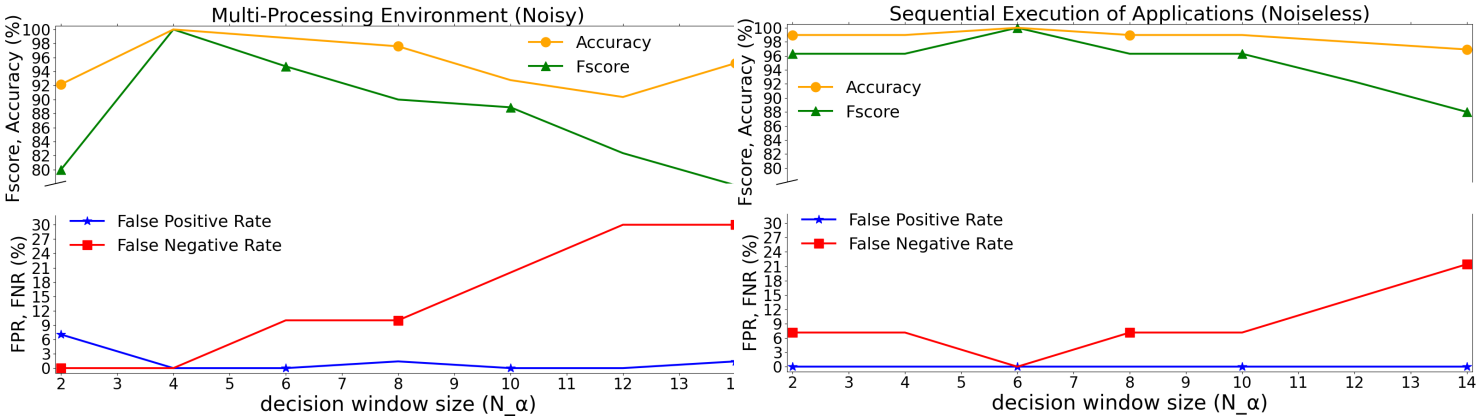
- When using small decision windows, we observe a high FPR when testing in a noisy environment. We can observe this in Figure 3.12a, the FPR (blue line) is 7% for a window size of 2. For small window sizes, we assign more weight to current decisions, which means that current False decisions have more significance in the final average. When testing in a noiseless environment, we can see in Figure 3.12b that we have an FPR of 0% because we have less noise due to context-switches, which translates to less mis-classifications due to the added noise in the measurements.

- On the contrary, increasing the decision window, the FPR decreases to less than 1%. Increasing the size of the window means that the current measurements have less weight, and to calculate the final decision we use more the past decisions. However, increasing the size of the decision window also affects the False Negative Rate (FNR), as we can see from the red line in Figure 3.12a and Figure 3.12b. This is because it is more likely to miss malicious behavior if we consider multiple previous decisions to compute the current decision. The sliding window average except malicious decisions will include normal decisions, shifting the average closer to a negative than a positive decision. We also note that in the noiseless environment seen in Figure 3.12b, the window size at which the FNR starts to increase is 7, while in the noisy environment it is 5. This is because in the noisy environment, when the window size is 5, decisions are averaged with an execution time of 5 ms, while our monitoring interval is 1 ms. Essentially, the decisions of the current application and at least one decision of the previous application are averaged. EWMA is successful in removing FPs because we observed that FPs are adjacent to normal decisions. On the other hand, TPs are adjacent with other TPs i.e., an attack performs continuously malicious operations in contrast to a FP which happens between normal operations.

- Finally, we can see from both Figure 3.12a and Figure 3.12b that metrics change more easily in the multiprocessing system than in a noiseless environment. For example, we can see that the classifier in the noisy environment has an F-score of 80% for a decision window of 2, while the same classifier in a noiseless environment has an F-score of 96%. For both classifiers, the F-score increases as the size of the decision window increases, as we successfully filter FPs. However, at

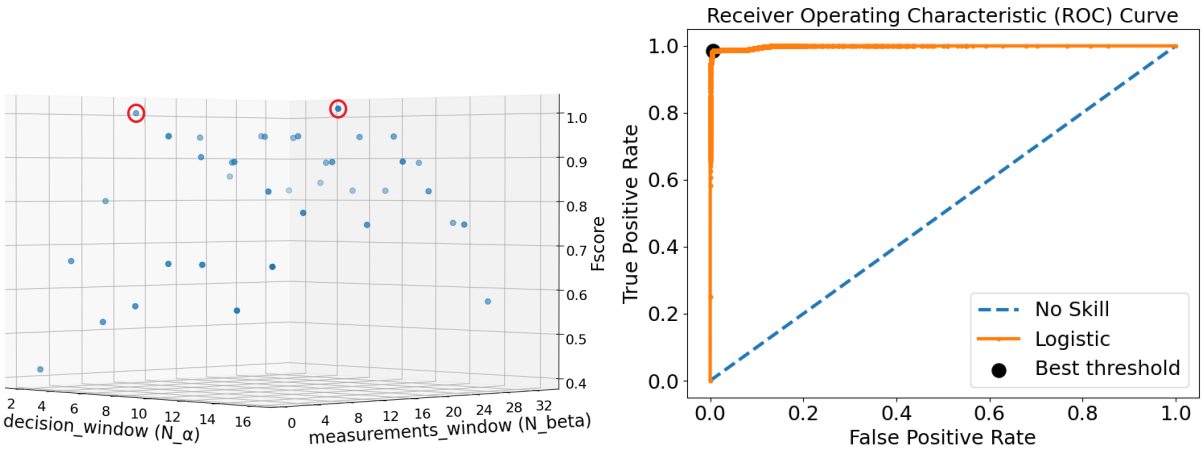
a certain point, the F-score of both classifiers starts to decrease as we increase the window size, since we miss the malicious behavior when we increase the window size used for filtering FPs. However, the classifier in the noisy environment experiences a steeper change in the F-score than in the noiseless environment. This is because increasing the window size in a noisy environment increases the likelihood that normal decisions due to a previously executed normal application are included, as opposed to sequential execution where it is highly likely that previous decisions belong to the currently executed application. This observation illustrates why it is important to consider multiprocessing systems when designing a detection mechanism, since failing to account for noise due to context switching and performing aggressive filtering can lead to false security guarantees in real execution environments.

We chose the window size that balances the FPR and FNR in the noisy environment, which in our case is a sliding window of size 4. Figure 3.13a shows the F-score for different sizes of decision windows and measurement windows. We mentioned earlier that we also apply EMWA to the extracted HPC values to reduce the high-frequency noise. Smoothing too much the extracted values could lead to a loss of information. In the figure, we observe the two points highlighted in red that give us the maximum F-score. We choose a decision window of 4 and a measurement window of 9 to configure our detection mechanism. From Figure 3.12 and Figure 3.13a we observe that the best F-score lies in the boundary of the minimum granularity, which in this case is 4ms.



(a) Multi-processing (Noisy environment). (b) Sequential execution (Noiseless environment).

Figure 3.12: Classification Metrics for different window sizes used in EWMA to remove False Positives in the Multiprocessing and Sequential systems.



(a) (b)

Figure 3.13: (a) F-score depending on the measurement window and decision window. The two red circles highlight the window sizes (decision_window= 4, measurement_window = 9) and (decision_window = 16, measurement_window = 9), (b) The receiver operating characteristic curve obtained for our classifier.

Evaluation of the detection performance of MaDMAN

We evaluate the performance of our local detection mechanism in both noiseless and noisy environments using the configurations described in Section 3.5.2. We also evaluated the local detection mechanism using evasive SATHV according to the approaches mentioned in Section 3.4. It is worth mentioning that we train the classifier with the fastest possible eviction strategy on our platform. Next, we created some attack variants with slower eviction strategies and some based on inserting NOP instructions and random *sleep*() functions during code execution which we only use for evaluation. The insertion of NOP instructions and *sleep*() functions was carefully placed during atomic tasks, so the attacks were still successful. In Figure 3.13b we observe the ROC curve of our classifier in a noiseless environment. The mechanism has an AUC of 0.983 and an F-score of 98.72%. The ROC curve was generated with different classification thresholds and without applying EWMA for FP filtering. With the classification threshold of 0.5 and the application of EWMA, we obtain the following results:

- For the noiseless environment, the classifier has an accuracy of 98.96% and an F-score of 96.3% with an FPR of 0%.
- For the noisy environment, the classifier has an accuracy and F-score of 100%. The classifier performs better in a noisy environment because the side-effects are sensitive to context-switching and we choose a sliding window size to maximize performance in noisy environments. For example, from Figure 3.12, the classifier achieves a maximum F-score of 100% in a noisy environment with a sliding win-

dow size of 6.

Regarding evasive SATHV our classifier was capable to detect all evasive SATHV without any FNs. In contrast, the detection mechanism of Li et al. [126], [138], detects with 0.23% FNs for Rowhammer and 3.83% FN for Spectre, also using Logistic Regression. When they tested their detection mechanism with an evasive Spectre attack, their detection accuracy dropped to less than 90%. The difference between our mechanism and that in [126], apart from the selection of side-effects, is the monitoring interval. Li et al. used perf tools to extract the hardware features, which only allow them to monitor at a minimum granularity of 100ms. Because we monitor 100 times faster, we are more likely to detect malicious activity in the presence of evasive techniques because we can monitor application behavior more closely and there is less time for an attacker to create evasive yet successful applications.

Last but not least, we measure the total performance overhead caused by the detection mechanism in the local system. This is a major concern as the detection mechanism runs in parallel with the other applications, which affects their execution. To measure the performance overhead, we ran the same set of applications with and without the monitor using the *time* command in Linux. The calculated performance overhead was 1.3%, which we consider acceptable.

Detection mechanism	**Machine Learning**	**Attacks**	**Evaluation metric**	**Evasive**	**Evasive accuracy**	**Performance overhead**	**Monitoring Interval**	**Real Time**
Payer et al. [85]	No, conditions	flush-based CacheSCA, Rowhammer	100% Accuracy	not considered	N/A	0.91% Intel Core i7-3770	1s	yes
Li et al. [126]	polynomial SVM	flush-based Spectre, Rowhammer	100% AUC	yes	less than 95% Accuracy	N/A Intel Core i3-3217U	100ms	yes
Chiappetta et al. [82]	Neural Network	flush-based CacheSCA	86% (averaged)	not considered	N/A	2.30% Intel Xeon W3670	1μs	yes
Gulmezoglu et al. [76]	LSTM	flush-based CacheSCA, Rowhammer, Meltdown, Spectre	99.70% F-score	not considered	N/A	3.5% Intel Xeon E5-2640v3	1ms	yes
Ours	Logistic Regression	eviction-based CacheSCA, Rowhammer, Spectre	100% F-score	yes	100% F-score	1.3% ARM Cortex A9	1ms	yes

Table 3.4: SATHV local detection mechanism comparison

3.6 Summary

In this section, we presented our experiments on attack detection using information from HPCs. SATHV is becoming increasingly popular and researchers are proposing

mechanisms to either secure the system or detect the attacks and make appropriate decisions. We mainly focus on detection mechanisms since we can use them directly, as opposed to solutions that try to secure the system and could only be applied in future architectures. In this work, we collected the most interesting side-effects used in proposed detection mechanisms, and created a taxonomy. Our goal is to help security researchers interested in implementing detection solutions against SATHV. The list of side-effects could aid them choose which side effects to use in order to reduce development time and cost. We also investigated whether we could reuse the theoretical side-effects when we have the same threat model but target different architectures. We showed that attack variants of the same targeted vulnerability can potentially bypass detection mechanisms, leading to false security guarantees. For example, most SOTA works use flush-based attack vectors due to their simplicity, low latency, and high success ratio. However, eviction-based attack vectors can be applied from the user-space in most architectures, leading to various side effects in the system that could allow them to evade detection as they exhibit different behavior than flush-based attacks. Finally, we proposed a local detection mechanism that aims to detect eviction-based SATHV in an ARMv7 system. To reduce the performance overhead induced by our solution in the local system, we proposed the use of a classifier with Logistic Regression. Since Logistic Regression is a linear classifier, proper selection of side-effects and filtering of FP have a major impact on the final evaluation. We evaluated our solution in noiseless and noisy execution environments to account for multi-processing systems. Our solution successfully detects all SATHV and evasive SATHV with an FPR of 0%. Finally, Table 3.4 presents a taxonomy of our mechanism and other mechanisms proposed in the SOTA.

However, our solution is limited to ARMv7 systems, since cache maintenance instructions such as the *flush* instruction are not available in the user-space. As we mentioned earlier, eviction-based attack vectors cause different side-effects than flush-based attack vectors. On a platform where we can test both approaches, as well as the complexity and different behavior of modern normal applications, there is a possibility that a linear classifier will not be able to correctly distinguish normal from malicious applications due to the limited amount of information it can learn. More complex solutions could be used, but at the cost of higher performance and memory requirements. As we aim to protect IoT devices, the already limited computational resources may pose difficulties in implementation.

4

A Local-Remote Security Mechanism for the Detection of Attacks in IoT

With the increasing number of resource-constrained systems such as IoT and IIoT devices, we are observing an increase in the cyber-security attacks on these devices. Malware detection solutions exist in the literature, but despite their accuracy, they are generally not suitable for resource-constrained systems. In such systems, performance, memory, energy consumption, and communication bandwidth, as well as decision-making latency, play an important role in adopting a security solution for attack detection. In this chapter, we propose an edge computing attack detection security solution that uses a hybrid Local-Remote Machine Learning implementation to strike a balance between accurate and fast malware detection while addressing the constraints of resource-constrained systems in terms of memory, performance, and communication bandwidth. We also evaluate different implementations in terms of their detection accuracy and overheads on the target system using Software Attacks Targeting Hardware Vulnerabilities.

4.1 Motivations of the Work

We first present the motivations and problematics of this chapter. More details can be found in [141].

- Modern applications are becoming increasingly complex, resulting in a wide range of different behaviors on the local system. Moreover, attack vectors exhibit different behaviors even among variants of the same attack family, as we saw with the eviction-based and flush-based attacks in Chapter 3. In the previous Chapter 3, we showed that only flush-based detection methods may not be sufficient when eviction-based attack vectors are used. Porting eviction-based and flush-based attack vectors into the evaluation platform, such as an ARMv8 system, may reveal the need for additional security implementations. Simple ML mechanisms may not be able to learn to distinguish between normal and malicious applications, resulting in many false positives. Complex ML solutions, on the other hand, are able to learn the various features to accurately distinguish between normal and malicious applications.

- IoT pose many limitations. IoT and IIoT devices are generally resource constrained. Performance, memory, energy consumption, and communication bandwidth are some of the key parameters that need to be considered when designing a mechanism for these devices. Proposed solutions that do not adhere to these parameters may not be adopted by the market. Our goal is to evaluate different trade-offs to help product designers optimize their detection mechanism depending on the context of their applications.

- Remote detection alone is not sufficient. Since simple ML mechanisms may not be sufficient, and since complex ML mechanisms can be, but we may not be able to implement them locally, remote implementations are instead used. This establishes a connection between the devices and a remote control center, which may leave the devices unprotected in the event of a connection failure. One such case is a network outage. In such a case, the device should be also autonomous, which means we need to implement a data processing locally. A detection implementation based on the idea of edge-computing could be more efficient.

4.2 Introduction to the Local-Remote Detection Mechanism

We looked at Chapter 2 to basic concepts of implementing a detection mechanism. Security designers suggest either an implementation in the local system or a remote implementation on a Server. The main reasons for these two options are the following:

- A local implementation processes the extracted samples on the fly, makes decisions and takes appropriate actions once malicious behavior is detected. This allows the device to be autonomous.
- In a local implementation, it is not necessary to store the extracted samples to send them later to a remote security mechanism.
- A remote system allows the implementation of a complex ML mechanism that might be locally constrained.

The above reasons lead security system designers to decide which implementation they prefer, taking into account their system and security parameters. A remote security mechanism obeys the concept of Cloud Computing. Cloud computing is drastically changing the way we collect and process data, accelerating the development and use of IoT and IIoT devices. IoT devices collect data on-site and send it to a remote server for processing. But as the number of IoT devices deployed increases, the Internet is flooded with data, much of which may be of limited value. This brings us to the post-Cloud era, where IoT devices produce data that is stored, processed and analyzed, and perform actions closer to the edge of the network. This will take pressure off cloud centers, as the amount of data generated by users can exceed the global IP traffic of data centers [142]. We call this post-Cloud era Edge-Computing.

In SOTA, there are three interesting approaches that use edge-computing to preprocess and minimize the data extracted and transmitted to the Cloud. The first is adaptive sampling, proposed by [143]. The authors propose to dynamically adjust the sampling based on the risk level of a vital sign according to the changing health status of the patient to reduce the extracted data. This approach works by examining the level of variance between the data collected over a period of time and dynamically adjusting the frequency of sampling. Adaptive sampling approaches work well in applications where the time series being collected are stationary. For rapidly changing data, such as the data extracted by HPCs, these approaches perform poorly. Since this technique observes past data and reduces the sampling rate when the risk is below a certain level, an attacker in our case could use evasive techniques to reduce the probability of in-

creasing the risk level and still keep the sampling rate low. Low frequency monitoring, as described in Section 3.4, could then allow the attacker to more easily bypass the system's protections. The next approach, described in [144], uses a DNN to compress the data before sending it to the Cloud for further processing. Since they use a DNN, they do not implement it in the device, but in the edge node, where the DNN receives data from the device every minute, compresses it, and sends it to the Cloud for further analysis. In our case, since we perform attack detection, we would like the detection mechanism to be implemented in the device in case of connectivity failures. As well, if the edge node is attacked, the data processed in the edge node cannot be trusted. Finally, since we monitor the system with high frequency, the generated data needs to be stored locally before sending it to the edge node, which increases the memory overhead. The third approach was presented in [145]. The motivation of the authors was to clean the "big data" extracted from the edge-devices, with the goal of increasing the accuracy on the analysis of the data in the edge-node or in the Cloud. The authors use anomaly detection algorithms locally to identify anomalous data, which they then filter out. They demonstrated that this technique can improve the judgment and feedback of the Cloud analysis on the extracted data.

On the other hand, we use the idea proposed in [145] but to perform the inverse. In our case study, we try to detect anomalous data due to attack execution, and filter normal data. To this end, we also propose a Local (edge-device) - Remote (Cloud, remote Server) solution for the detection of malicious applications. The proposed Local-Remote solution uses the concept of edge-computing to enhance the security and detection performance of the proposed SOTA solutions with respect to the constraints of an IoT system. Our implementation takes into account the following IoT constraints which we summarize:

- Security implementations in the local system must not cause significant performance and memory overheads. In contrast, in a remote system, there is no limit to the resources available.
- The local system must be able to identify malware in order to be independent of the remote system to some degree. This is useful when the network is down. Though, a local security implementation is required.
- There is a need to accurately detect malware. We cannot allow malware to continue to perform its malicious activities. Also, we cannot afford to misclassify normal samples as malicious, as this can result in a large system overhead due to the actions we need to take when identifying malware. For example, the recovery of the system in case of a false alarm will impact severely the system with the resets

overhead. Since there are multiple malware families and each family has different behaviors, it is more challenging to correctly classify all malware samples as malicious using simple techniques. Moreover, the different behaviors of normal applications make our task even more difficult. Nevertheless, further analysis of a ML implementation based on complex ML algorithms is necessary. As mentioned in related work, complex ML implementations can detect malware with higher accuracy than simple ML implementations. ML Implementations that can learn complex relationships between captured data and the target variable, i.e., normal or malicious behavior, will be able to better distinguish malicious and normal behavior.

- Since the simple ML cannot accurately distinguish between malware and normal applications, we use a similar technique as in [146] to increase the accuracy. The local ML only triggers an alert for samples with high probability. If a sample has intermediate probability but it is not a high probability sample, the local ML classifies it as suspicious and forwards it to the remote system for further evaluation. Using the technique described above, the edge device can preprocess and filter the extracted data HPC data. This reduces the storage space required to store HPC data and further minimizes the communication interface, energy consumption, and network overheads. In addition, successful filtering of most normal sensitive HPC data reduces the risk of data interception.

We base our detection solution on the idea of edge computing. It allows us to move parts of the storage and processing resources away from the remote center and closer to the local device. This idea overcomes many of the limitations of traditional cloud computing such as latency, service delays, network outages, and reliability.

The main novelty of this chapter can be summarized in the following:

- We use edge-computing solutions, targeting to minimize the communication bandwidth, but in the context of IoT security.
- We propose the use of a low-cost ML algorithm locally, to successfully filter normal data, and quickly detect malicious behaviors.
- We propose the use of a more complex solutions in a remote system, to further analyze accurately the received data.
- We focus on minimizing the FPR, but still keeping our ability to recognize malicious behaviors high.
- We evaluate the filtering succeeded, and we propose the analysis of certain pa-

rameters that will further allow us to quickly detect malicious behavior, even if we use a remote ML.

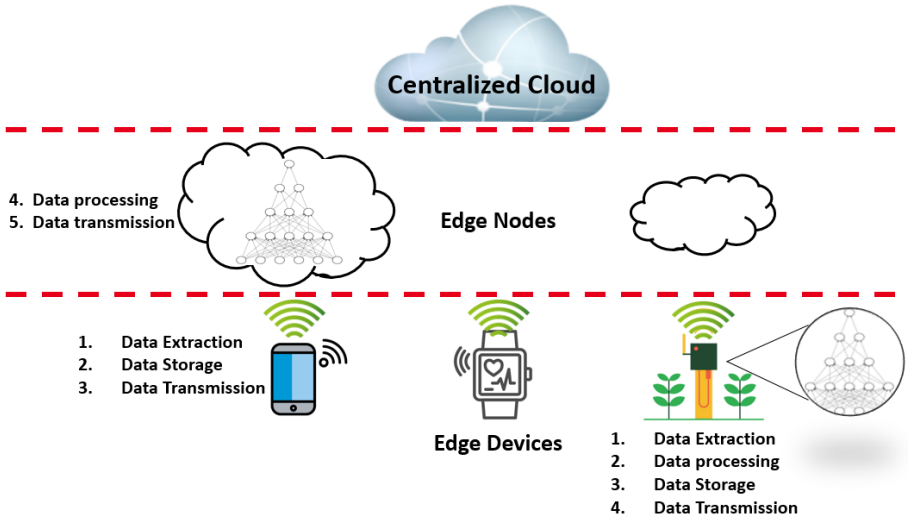


Figure 4.1: Local-Remote Edge-Computing approaches and the different network layers.

Finally, Figure 4.1 shows the different network layers and where our approach and the approaches of [143]–[145] are implemented. At the lowest layer are the edge-devices, above that are the edge-nodes, and then the Cloud layer. There may be other layers between the cloud and the edge-nodes, but they are not necessary for the purpose of this work. On the left side of the figure, we see the edge device and the different steps to process the extracted data using the approaches presented in [143], [144]. The edge-device extracts the data from the sensors, stores them locally, and then transmits them to the edge-node after a predefined period of time. The edge node then processes them and sends the processed data to the Cloud for further analysis. On the other hand, our approach and that of [145] can be seen in the right side of the figure. In this case, we extract the data from the edge-device, pre-process it before storing it locally, and finally transmit it to the remote system (Cloud) after a predefined period of time.

4.2.1 Local ML Implementation

The local ML can be a supervised, semi-supervised, or unsupervised ML algorithm. The local ML reads the samples from the HPCs and outputs a probability between 0% and 100% for each sample to be malware. The above implementation, as shown in Figure 4.2, allows us to trust samples with a low probability of being malware, send samples with an intermediate probability to a remote ML for further evaluation, and raise an alert if there are samples with a high probability. This is achieved by setting two thresholds:

- The first threshold (*alert threshold*) serves as a trigger for detecting samples that

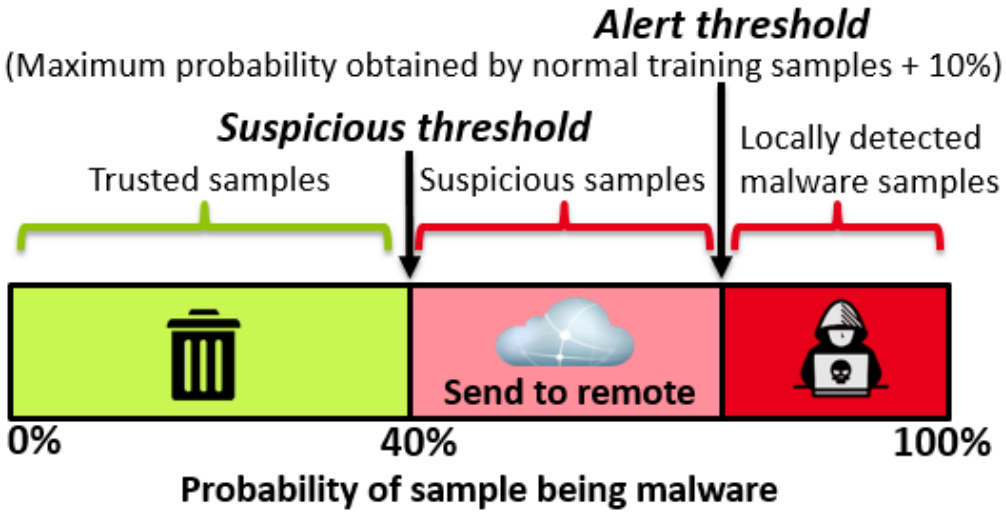


Figure 4.2: Two-level detection threshold implementation.

have a high probability of being malware. We choose the alert threshold based on the maximum probability obtained using the local ML for only normal training samples, plus an additional offset. This offset serves as an extra uncertainty level and was set to 10% for the purposes of this experiment. By using only the probabilities of the normal training data to calculate the *alert threshold*, we minimize the possibilities of having FPs. This is because during the evaluation, it is less likely that there will be a normal sample whose probability of being malware is greater than the maximum probability of the normal training data. Adding also an offset further decreases the probability of observing FPs, which we would like to avoid, as mentioned earlier.

- The second threshold (*suspicious threshold*) is set to store the samples that our local ML cannot identify with high confidence and send them to the remote ML for further processing. Since MLs from related work use the probability that a sample is malware at 50% to classify samples as malicious or normal, the suspicious threshold is set to 40%. By subtracting an offset of 10%, the local ML can flag suspicious, malicious samples with behavior closer to normal and future attacks based on zero-day vulnerabilities.

The 10% offset used by the two levels, and as it can be seen in Figure 4.2, was chosen only for the purpose of this experiment and to demonstrate our idea. More advanced algorithms can be used to define them properly. The algorithms that we use are presented in Algorithm 2 and Algorithm 3.

Suspicious threshold With Algorithm 2 we define the *suspicious threshold*. As mentioned earlier, we only use the training dataset to define the two thresholds. After filter-

Algorithm 2: Algorithm to choose suspicious threshold

input : Probabilities of a **training samples** being malware
input : EWMA window size
input : *offset* ; `// to be used as uncertainty`
output : *Suspicious threshold*

filter probabilities using EMWA;
threshold := threshold where G_{mean} is maximum ; `// As defined in Equation (3.7), balance between the TPR and the FPR.`

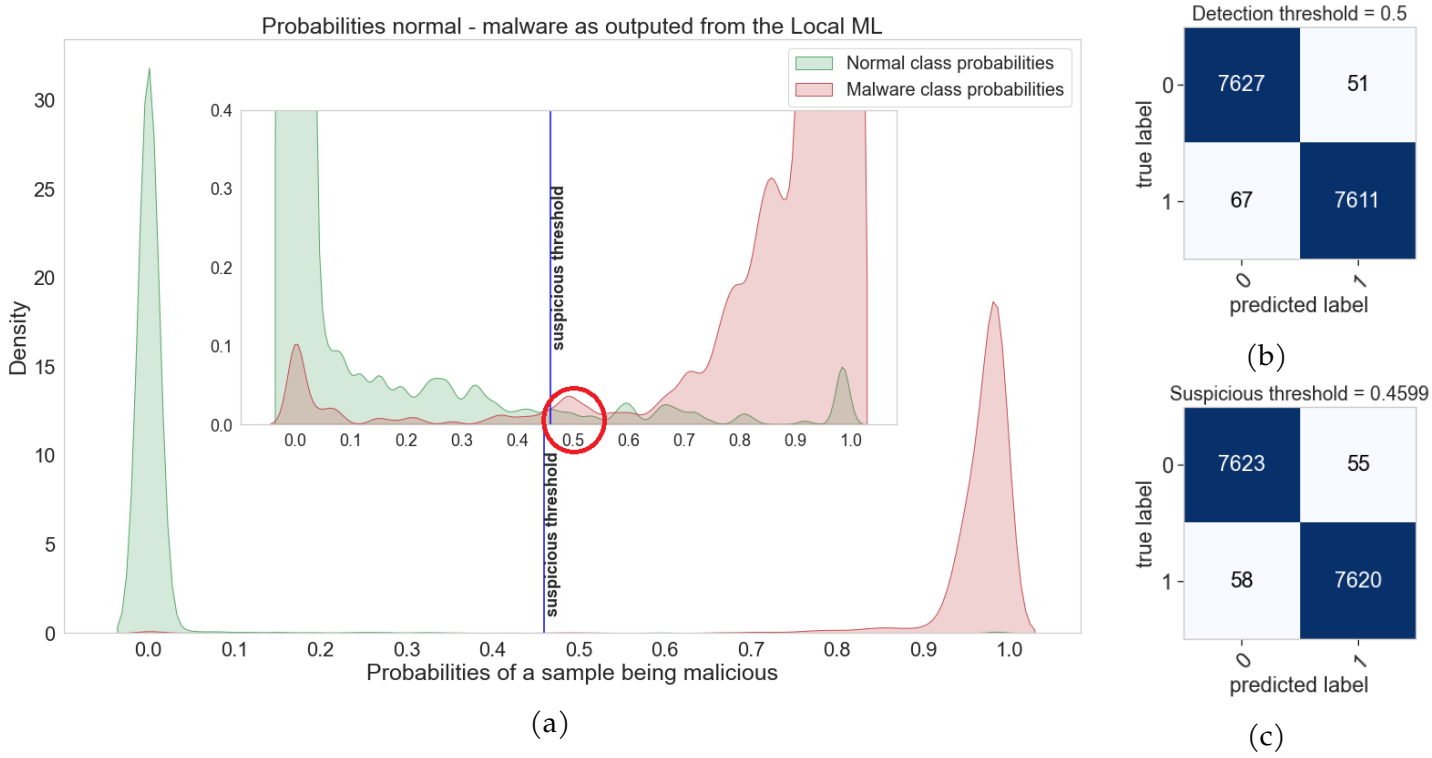
if *threshold* > 0.5 **then**
    Suspicious threshold := 0.5; `// If the threshold is greater than 0.5, we reduce our ability to detect malicious samples with behavior close to normal samples, with such a case being attacks based on zero-day vulnerabilities.`
else
    Suspicious threshold := threshold;
end

return *Suspicious threshold - offset*;

ing the probabilities of the training samples being malware, we choose the threshold that yields the highest G_{mean} Equation (3.7), i.e., we seek for a balance of FPR and TPR. In other words, we look for a threshold that balances the maximum TPs to be stored locally among all positively labeled samples for further evaluation in the remote, and the maximum TNs to be filtered from all samples labelled as "0". If the threshold is greater than 0.5, we set it to 0.5. We chose this value because a *suspicious threshold* greater than 0.5 could filter out malicious samples with behavior close to the normal samples, with such as case being attacks based on zero-day vulnerabilities.

To further explain the reasoning behind certain parameters, two examples are presented in Figure 4.3 and Figure 4.4 where the threshold returned after computing G_{mean} is less than or greater than 0.5. In the general case, Algorithm 2 reduces the classification threshold to less than 0.5 only if the current classifier cannot effectively distinguish between malicious and normal samples, i.e., there exist many malicious samples with less than 50% probability of being malware. An example of this case is shown in Figure 4.3. As can be seen from the confusion matrix using 0.5 as the detection threshold (Figure 4.3b), there are a greater number of FNs than FPs for the same number of samples from the two classes. Since the number of FNs is greater than the number of FPs, G_{mean}, which seeks a balance between TPR and FPR, must decrease the classification threshold to less than 0.5 until there is a "balance" between the two factors. For this example, using G_{mean}, we obtained the threshold of 0.4599, which as we can see from the confusion matrix in Figure 4.3c, almost balances the FPs and FNs i.e., now FPs=55, FNs=58 when before we obtained FPs=51, FNs=67. From the distribution of probabilities that a sample is malware in Figure 4.3a, we can see that there is a small spike in malware

probabilities at 50%. By moving the detection threshold down, we effectively reduce the FNs and on the other hand increase the FPs, though balancing FNR and TPR. In such a case, we accept the reduction of the detection threshold because we want to evaluate as many malicious samples as possible in the remote system.



(a) (b) (c)

Figure 4.3: Distribution of the probability a sample is malware for a classifier that G_{mean} returns a threshold less than 0.5 (a), the confusion matrices with a detection threshold 0f 0.5 (b) and the modified suspicious threshold (c).

On the other hand, if G_{mean} yields a threshold greater than 0.5, it means that there exist many normal samples with probability greater than 50% i.e., FPs. We can see this in the confusion matrix in Figure 4.4b, the number of FPs is greater than the number of FNs i.e., FPs=59, FNs=44. Though G_{mean} returns a threshold greater than 0.5 to balance the two, which in this example is 0.6545. From the confusion matrix using the new threshold in Figure 4.4c, we can now see that there is a balance between FPs and FNs as now we have FPs=49, FNs=50 compared to previously having FPs=59, FNs=44. Also, in Figure 4.4a, which shows the distribution of probabilities, we can observe a small spike in the probabilities of normal samples around 60%. By increasing the classification threshold from 0.5 to 0.6545, we correctly classify these normal samples. This decreases the FPs, but we increase the FNs, so we successfully balance TPR and FPR. Since we can afford not to filter some normal samples, and since there may be new or evasive attacks with behaviors closer to normal, a threshold higher than 0.5 only decreases our

ability to detect these malicious behaviors by potentially filtering the incoming samples. Therefore, we set the threshold back to 0.5 if Algorithm 2 returns a threshold value greater than 0.5. Finally, an offset could be used to add an extra uncertainty level so that more samples could pass the filtering and be evaluated in the remote considering that our network maps most of the normal samples to probabilities around the 0%.

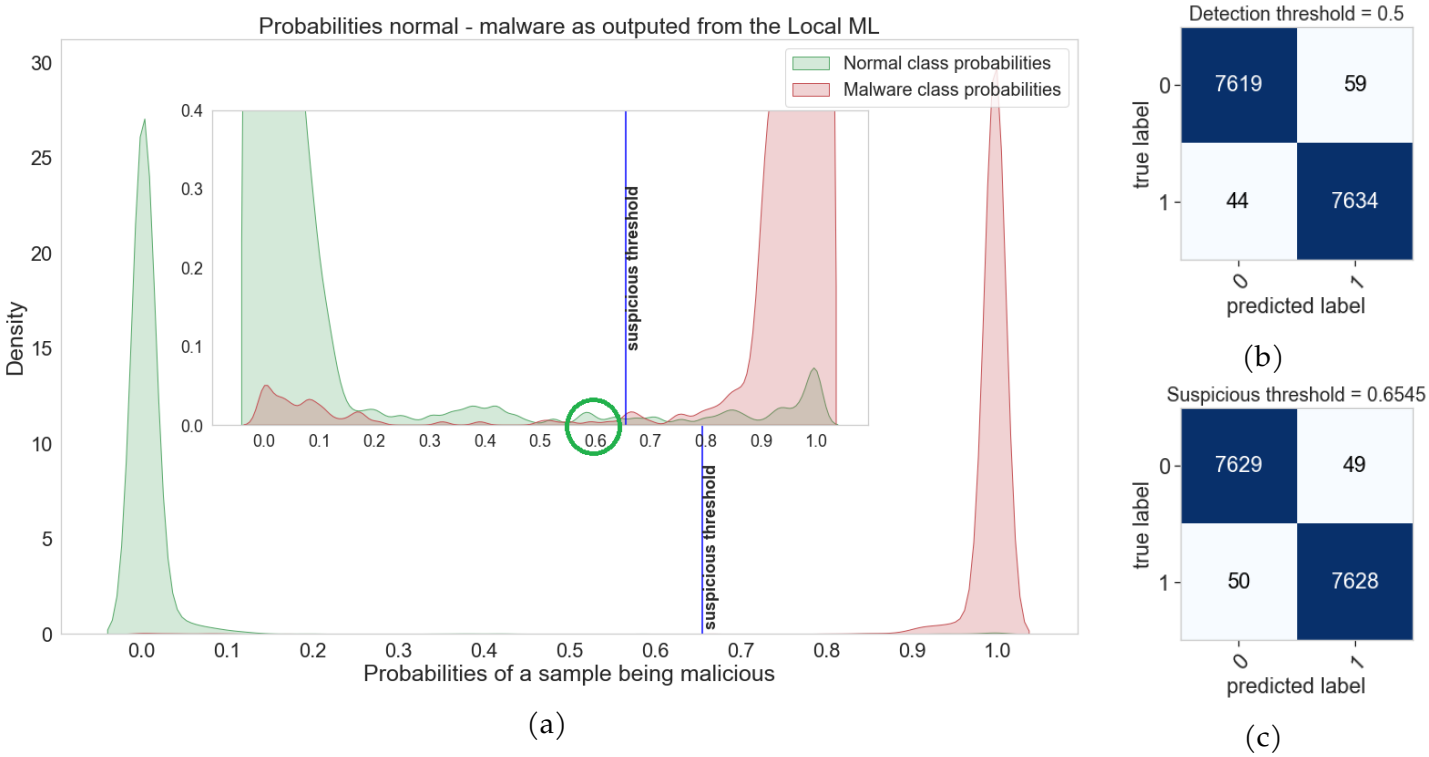


Figure 4.4: Distribution of the probability a sample is malware for a classifier that G_{mean} returns a threshold greater than 0.5 (a), the confusion matrices with a detection threshold 0f 0.5 (b) and the modified suspicious threshold (c).

Alert threshold To select the *alert threshold* we use the Algorithm 3. Here we also filter the probabilities of the training samples being malware to reduce noise. We initialize the threshold with the maximum probability of being malicious among **only** the normal samples. Then we decide if the sample is malicious (**"1" or *positive***) or normal (**"0" or *negative***) based on whether the probability of the sample is above or below the threshold. Next, we apply EWMA to the decisions to filter FPs as described in Section 3.5.2. We then obtain the number of FPs from the confusion matrix. If there are FPs, we increase the threshold by one step and add an offset, as in *suspicious threshold*. If there are no FPs, we decrease the threshold by one step and repeat the process. We do this because there might be some normal samples that have a high probability of being malware, but are successfully removed by EWMA filtering. If we do not check this, the threshold might be too high to successfully alert the system in case of malicious sam-

ples. Again, the offset is used as uncertainty because novel normal applications could exhibit behavior similar to malicious programs, and we want to avoid detecting them as attacks and triggering a False alarm.

Algorithm 3: Algorithm to choose alert threshold

```
!htbp
input  : Probabilities of a training samples being malware
input  : EWMA window size
input  : offset ;                                   // to be used as uncertainty
output : Alert threshold

filter probabilities using EMWA;
threshold := maximum probability of normal samples;

while True do
    decisions := 1 if probability > threshold else 0;
    filter decisions using EWMA window size;
    extract confusion matrix;
    get number of FPs;

    if FPs then
        threshold := threshold + 0.001 ;                // Use previous threshold
        break;
    end

    // We decrease the threshold because there might be some probabilities of
       normal samples that are above the threshold, but EWMA can filter these out.
    threshold := threshold - 0.001;
end

Alert threshold := threshold

return Alert threshold + offset;
```

4.2.2 Remote ML Implementation

Suspicious samples are stored locally and then sent to a remote ML implementation for further analysis. The local system sends suspicious samples to the remote system each Δs which specifies the period of time that the local ML stores suspicious samples in a local storage and the frequency with which these samples are sent to the remote system via a communication interface. The remote system is based on a complex ML, which is capable of learning complicated behaviors. It can be either a supervised, semi-supervised or unsupervised ML implementation. An unsupervised ML implementation, e.g., an LSTM, learns only normal behavior and detects as malware any behavior that deviates significantly from the patterns our ML mechanism is trained on. The remote system checks suspicious samples received from the local system for deviations from normal behavior, and if there are deviations, they are flagged as abnormal. The purpose of the remote system is to flag as abnormal any malware application that the simple local system cannot identify, and also to recognize zero-day attacks. Examples of

such complex MLs are Autoencoders, one-class SVM, LSTM and CNN networks. After receiving the suspicious samples and processing the data, it notifies the local ML about the presence of a malware sample according to a third detection threshold.

In Figure 4.5 we present a global view of the proposed idea. As we can see from the figure, each device is equipped with a local detection mechanism. The local detection mechanism extracts information from the HPCs, which then inputs to the local ML. After the local ML outputs the probability that the current sample is malware, the system performs a probability check. Depending on the probability, the sample is classified as either trusted, suspicious, or malicious. If the sample is classified as malicious, an alert is triggered. If the sample is classified as suspicious, it is stored in local storage to be later evaluated in the remote ML. At each Δs, the local system sends the samples stored in local storage to the remote system, which receives and processes the remote ML. If a sample in the remote ML is determined to be malicious, the remote system notifies the local system so that it can take appropriate action.

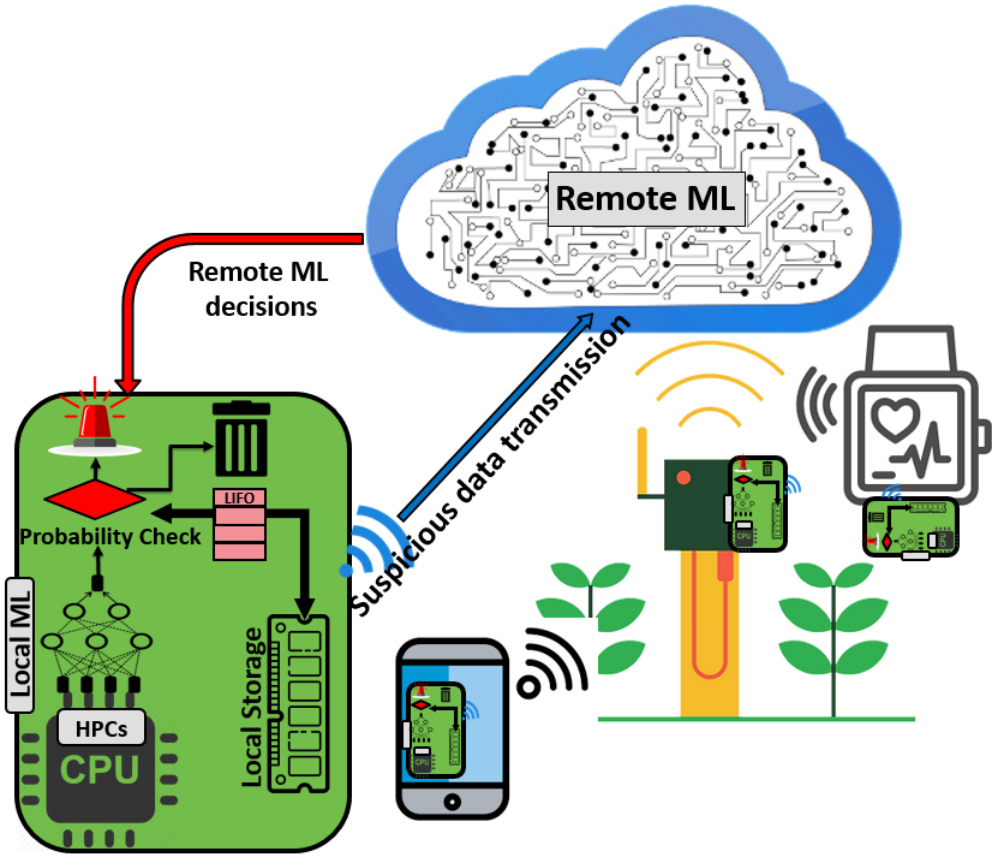


Figure 4.5: Global view of the Local-Remote ML implementation.

Saving the Necessary HPC Samples Depending on the Remote ML Configuration

Depending on the remote ML, the local system is configured differently to save suspicious samples. If the remote ML uses only one sample as input to predict whether it is malicious or normal, the local ML stores only the suspicious samples in local storage. This is the case, for example, when we use an Autoencoder or a one-class SVM. An Autoencoder is a network that takes an input, compresses it, and then tries to reconstruct it as explained in Section 2.4.2. Since the Autoencoder is trained to reconstruct only normal samples, if the input is a malicious sample, it will have difficulty reconstructing it, resulting in a large reconstruction error. Since the Autoencoder does not perfectly reconstruct normal samples, we choose the quantile (99.85%) of the distribution of the mean square of the reconstruction errors of the normal training dataset as the detection threshold during training. If the reconstruction error is larger than the detection threshold, the Autoencoder alerts the local system. In this case, the Autoencoder takes the suspicious sample as input and decides whether it is normal or malicious depending on the reconstruction error.

But since an Autoencoder only looks at one sample to decide for the presence or not of an attack, this might decrease its ability to efficiently distinguish between normal and malicious samples. Looking also at the past behavior might allow us to take more appropriate decisions by looking at a sequence of actions. As we will show later in Table 4.1, the remote models that use also the past information have better capacity to identify malicious from normal applications.

When such networks are deployed remotely, some modifications must be considered. An LSTM is one such network. The reason is that the LSTM needs information from the past to predict the present or the future samples. In Figure 2.11 in Section 2.4.2, we saw a representation of an LSTM network that has as input the past n samples and tries to predict the current plus m future samples. In such a case, to predict the label of the suspicious sample, the remote ML also needs the past samples. However, in the previously proposed local filtering approach, the past samples could be filtered by the local ML and there is no guarantee that suspiciously labeled samples are continuous in time. To account for this, each time the local ML flags a sample as suspicious, we also store the necessary past and future samples in the local storage. Thus, when an LSTM remote ML is used, the additional information about the past and future HPC samples increases the data stored and sent to the remote.

A control-flow diagram illustrating the steps we need to take to properly store all the required information can be found in Figure 4.6. For this example, we chose to implement a remote ML that requires $n = 3$ past samples to predict $n = 0$ future samples,

i.e., we only predict the samples at time t. The component that helps us accomplish the efficient storage of the past 3 samples in the local storage is a First In First Out (FIFO) with a maximum size of the number of past samples plus the number of samples to be predicted, which in this case is 4. The FIFO has the main advantages:

- If a sample is neither suspicious, nor a high probability sample, then it is filtered through the local ML using the previous methodology when the remote ML is not an LSTM or another complex ML requiring information from the past samples. In the current case that the remote ML is an LSTM, a sample with probability of being malicious less than the *suspicious threshold* is pushed to the FIFO. If the local ML continuously filters samples, which means that we continuously push the filtered samples into the FIFO, then the FIFO stores only the necessary m past samples and discards the unnecessary ones. This is because, if the FIFO is full and a new sample is pushed, the FIFO will pop the sample added first to add the one added last i.e., now.

- In case of an LSTM, to predict the label of a suspicious sample in the remote ML, we need the n past samples. However, if there are multiple continuous suspicious samples, then the required past samples are already stored in the FIFO or local storage.The FIFO helps us avoid storing unnecessary information. If we pop the FIFO samples to store them in the local storage, then the FIFO will be empty. If the next sample is also suspicious, it will be the only one saved in the local storage. However, we are sure that the previous samples are already saved and avoid saving the same samples again.

- The FIFO help us maintain consistency over time, as the samples saved earlier at FIFO are also saved first in the local storage.

The algorithm in Figure 4.6 is explained as follows: when we start monitoring, we push the first n=3 samples to the FIFO. Then we check if the local storage is full. If it is, we immediately send the samples saved in the local storage to the remote ML. This could help us identify malicious samples faster than Δs, as we will see in the next sections. Then we check if the acquisition time Δs is over. If it is, we also send the saved data to the remote ML. If not, we continue extracting samples and decision making. If the sample has a probability greater than *alert threshold*, we raise an alarm, otherwise we check if the sample has a probability greater than *suspicious sample*. If so, we add an extra bit '1' to flag it as suspicious and push it to the FIFO. This allows us to identify in the remote ML which samples we need to predict their label, since we only want to decide if a suspiciously flagged sample locally is really malware using the more complex remote ML. Then we pop all the elements from the FIFO and saved them in the local storage.

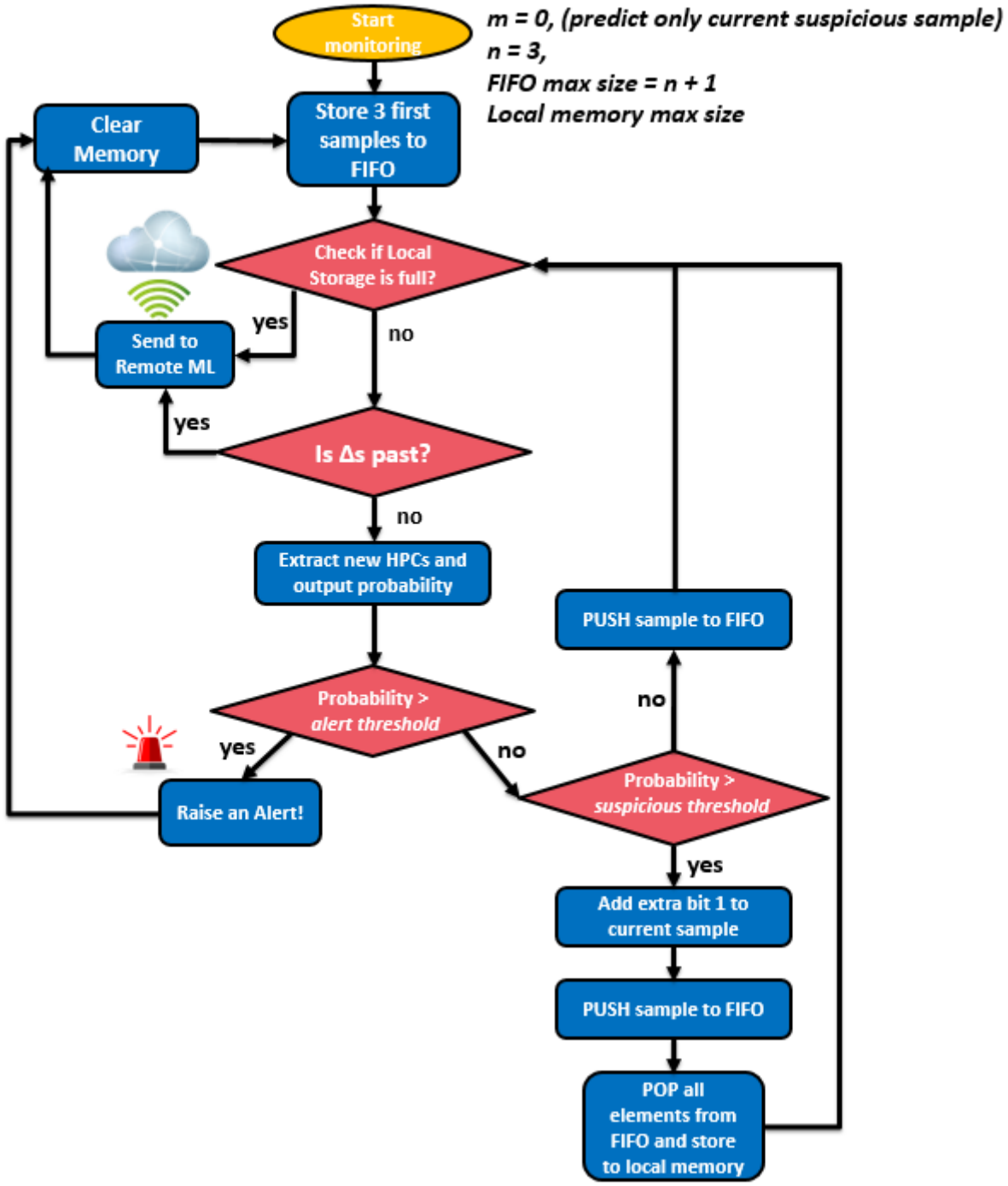


Figure 4.6: Control-flow graph of the steps necessary to store previous samples when the remote ML requires it. In this example, we require $n = 3$ past samples and we predict the current plus $m = 0$ future samples.

If the sample has a lower probability than *suspicious sample*, we mark it as '0' and push it to the FIFO. Then we continue the loop.

To better explain the concept, we present an example in Figure 4.7 and Figure 4.8. In Figure 4.7a we see the first step of the control-flow graph pushing the first 3 samples in the FIFO. In Figure 4.7b the newly extracted sample is labelled as "0" and is also pushed to the FIFO. As we can see, the FIFO contains the current sample and the necessary past samples. As in Figure 4.7c, the new sample is also labelled as "0", and when pushed into the FIFO, this causes the FIFO to discard the latest added sample, which effectively filters it. In Figure 4.7d, the new sample is flagged as suspicious and pushed into the FIFO, which now stores the suspicious sample plus the $n=3$ last samples needed for prediction in the remote ML. Then all the samples stored in the FIFO are popped and saved in the local storage. The timestamp is only for the purpose of this example and is not actually saved. The remote ML looks only for the samples with the positive label and uses the necessary previous samples to make a prediction. In Figure 4.8c we have the case where the previous sample is suspicious and the current sample is also suspicious. In this case, the current sample is pushed to the FIFO and popped to be saved in the local storage. The samples needed to predict the current sample are already in local storage because of the previous suspicious sample. Finally, in Figure 4.8d we see that the samples labelled as "0" are filtered by the local system at times $t+7$, $t+8$, $t+9$.

The local filtering allows us to reduce the memory required to store the extracted samples locally and though minimizing the communication and network overhead of transmitting them to the remote system each Δs. In the following sections, we evaluate different Local-Remote implementations and present their overheads and detection capabilities.

4.3 Local-Remote Parameter Configuration and Evaluation

In the following subsections, we present our experimental platform in Section 4.3.1, HPC event selection in Section 4.3.2, and detection metrics for various simple and complex MLs in Section 4.3.3. Next, in Section 4.3.4, we evaluate the Local-Remote system. We present the detection metrics and also evaluate the communication overhead per Δs and the percentage of filtering succeeded by the proposed approach. Finally, in Section 4.3.8, we evaluate the performance-memory overheads of the local detection mechanisms and the latency to make a decision for the different local MLs.

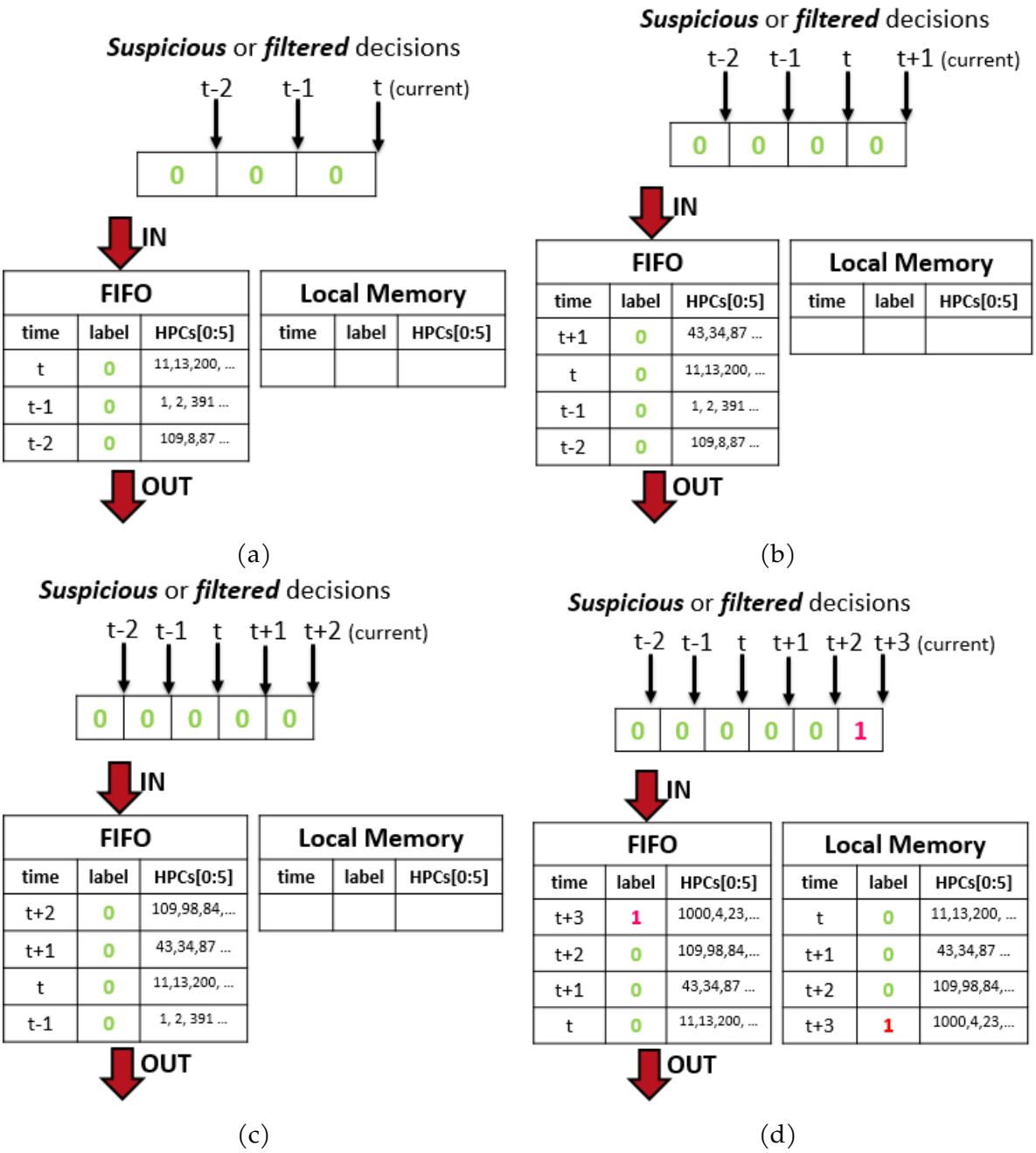


Figure 4.7: FIFO and Local Memory snapshots for sample storing from t-2 until t+3.

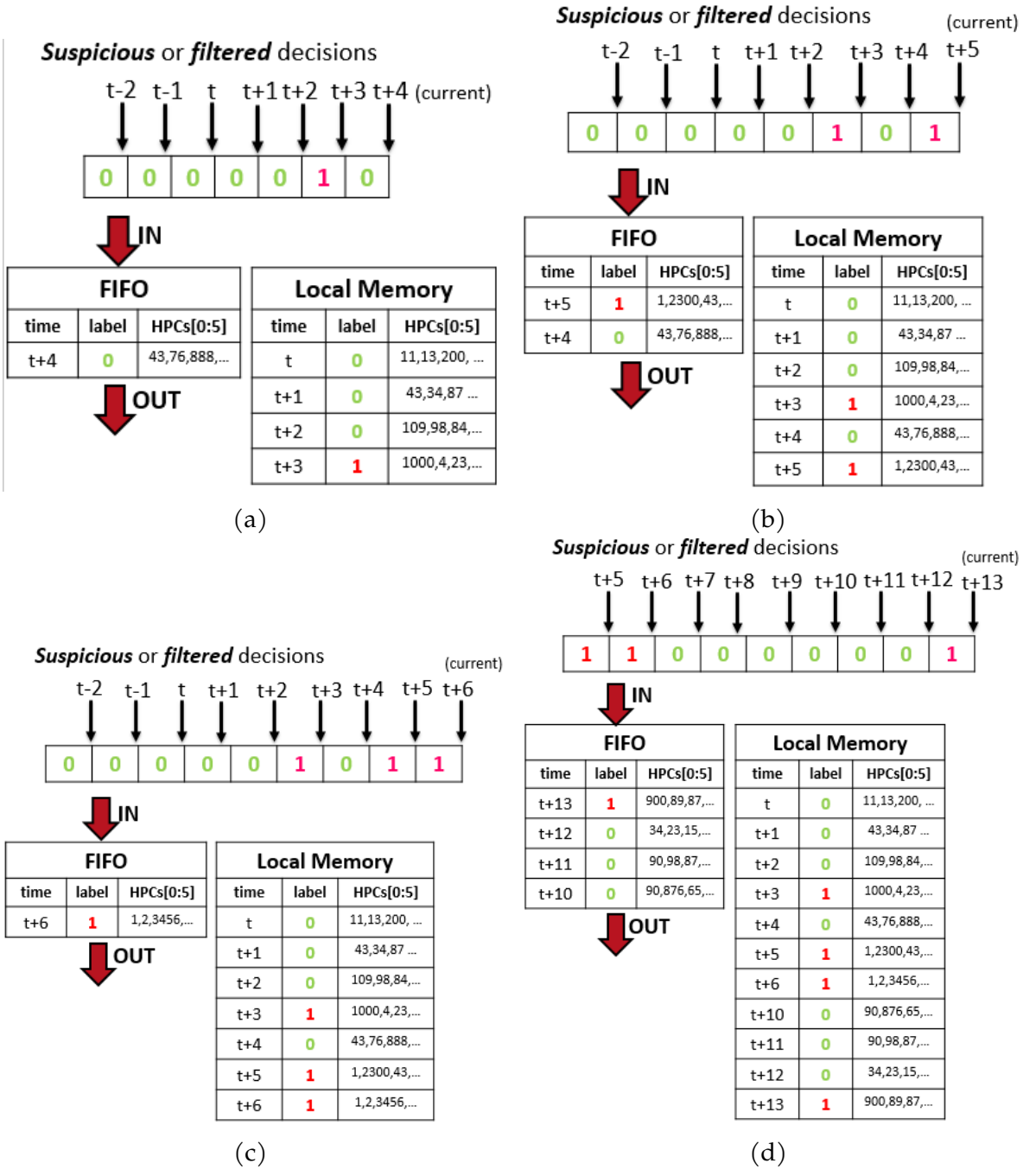


Figure 4.8: FIFO and Local Memory snapshots for sample storing from t+4 until t+13.

4.3.1 Experimental Platform

Our experimental platform is based on a Raspberry Pi4 Model B. It is a widely used device for IoT devices with high resource requirements. It integrates a high-performance 64-bit quad-core processor based on the ARM Cortex-A72 processor (ARMv8) running at 1.5GHz. It is equipped with 4 GB LPDDR4 RAM memory and an Ubuntu 20.04.2 LTS system. This platform enables the extraction of HPC measurements from six registers per core simultaneously.

To demonstrate our idea, we use several Software Attacks Targeting Hardware Vulnerabilities (SATHV) and evasive SATHV as test vectors. These include Cache Side-Channel Attacks (CSCA) variants, Spectre variants, Meltdown, and Rowhammer. More specifically we use CacheCSA attacks on the AES, RSA using the Flush+Flush, Evict+Reload, Flush+Reload, and Prime+Probe. For Spectre we use Spectre v1, v2, v4 using flush-based and eviction-based techniques. To test Meltdown we use Meltdown (v3) and finally for Rowhammer we use single and double-sided hammering using also flush-based and eviction-based techniques. Finally, evasive SATHV were created using the techniques presented in Section 3.4. The total attack library dataset has in total 31 attacks executables. This platform also allows us to test eviction-based and flush-based attack vectors as in ARMv8 the *flush* instruction is available from userspace in contrast to the ZyboZ7 (ARMv7) board used in the Chapter 3.

For our normal applications library, we include MiBench [134] and PARSEC [135]. These are industry-standard benchmarks for embedded systems and include a wide variety of applications as explained previously. The total number of normal executables from these two libraries is 77.

Finally, we extract measurements from the HPC registers each 1ms as explained in Section 3.4, and for the purposes of this experiment, we send the suspicious samples to the remote every $\Delta s = 1\ minute$.

4.3.2 HPC Event Selection

In a previous Chapter 3 we have presented a first version of our methodology and HPC event selection. The six best HPCs we considered for detecting malware in the context of IoT are the following: ISB SPEC, L1D TLB REFILL, BR IMMED SPEC, L2D CACHE REFILL, BR MIS PRED, and MEM ACCESS LD. This was based on a complete evaluation of all available HPCs from the ARM ISA [147] according to the experimental setup described in Section 4.3.1. The HPC events were chosen based on the Mutual Information between the HPC values and the target label.

As we can see, the selection of HPC events that give us the most information to distinguish between normal and malicious applications is not the same as the one in Section 3.5.2. In the previous section we selected the *L2 misses, Instruction cache misses, Data cache misses, ITLB miss, ITLB allocation, and DTLB allocation*. This is due to the fact that both eviction and flush-based attack vectors are included in this dataset in contrast to the previous chapter.

4.3.3 Performance Metrics of Different ML Algorithms

The first step in our experiments is to identify how each ML algorithm performs on our classification problem. Our motivation for evaluating the performance of each ML classifier is to demonstrate that simple local MLs are capable of identifying malicious samples but have a high FPR. This allows us to demonstrate the need for using complex MLs with a low FPR and finally the advantages of a hybrid Local-Remote ML.

We selected six local ML classifiers to test: Logistic Regression, Linear SVC, XGBoost, Neural Network, and AdaBoost. We also selected three remote ML classifiers: Autoencoder, LSTM, and LSTM Autoencoder. To keep the implementation simple, we chose a Neural Network implementation with 4-2 hidden layers (refer to as NN_4_2) and an AdaBoost implementation with 5 Logistic Regression estimators (refer to as AdaBoost[LogReg]). In addition we use two configurations for the LSTM, the first using the seven past samples to predict the current one (LSTM7), and the second using eleven past samples (LSTM11). We use the same configurations to test the LSTM Autoencoder. The goal is to investigate whether the accuracy of the remote ML requires an increase in communication overhead by sending more samples. That is, if we use more past samples to predict the current sample, does the accuracy of the model increase?

To evaluate how the following MLs perform **individually** on this classification problem, we use the metrics presented in Section 3.5.1. Further, as these simple MLs will be used as local MLs in the following, we will mention their benefits and pitfalls when used as local detection mechanisms. We pay close attention to FPR, as low FPR is as important as high recall for resource-constrained and critical devices. In Table 4.1 we display the results according to the different classification metrics for each of the classifiers. These results for the simple MLs are obtained by using as the classification threshold the probability 50% of a sample being malware. For the complex MLs, we use as the classification threshold the quantile (99.85%) of the distribution of the mean square of the reconstruction errors of the normal training dataset as in [148]. The quantile (99.85%) is equivalent to the point of the mean plus three sigmas in a normal distribution. We evaluate all MLs using all samples.

	Classifier	F-score	Recall	Precision	FPR	Accuracy
Simple MLs	*LinearSVC*	96.06%	99.75%	92.64%	1.63%	99.39%
	Logistic Regression	96.45%	99.74%	93.73%	1.46%	98.75%
	XGBoost	91.09%	85.78%	97.09%	0.53%	97.14%
	AdaBoost [LogReg]	98.22%	99.70%	96.79%	0.68%	99.39%
	NN (4,2)	90.37%	84.76%	96.77%	0.58%	96.92%
Complex MLs	*Autoencoder*	2.63%	1.33%	100%	0%	50.67%
	LSTM7	84.32%	72.99%	99.82%	0.03%	95.37%
	LSTM11	84.20%	72.87%	99.70%	0.04%	95.33%
	LSTM Autoencoder7	94.98%	92.41%	97.69%	0.45%	98.33%
	LSTM Autoencoder11	90.20%	84.11%	97.24%	0.49%	96.88%

Table 4.1: Classification metrics for the different ML algorithms.

Simple MLs used to Implement the Local Detection Mechanism

As we can see from the Table 4.1, the simple MLs have an F-score greater than 90.37%. AdaBoost[LogReg] delivers the highest F-score among the other simple ML implementations, LinearSVC has the highest recall, and XGBoost has the highest precision. Since the local MLs are responsible for filtering normal behavior and allowing malicious samples to be stored for further evaluation, recall and precision are the most interesting metrics. Recall indicates how well we allow malicious samples to pass our filtering and send them for further evaluation. Precision gives us an estimate of the FPs in the data to be sent for further evaluation and how well we manage to filter out normal behavior. Since the F-score measures the global tradeoff between precision and recall, the classifier with the higher F-score is best suited for our problem. Finally, as we do not want to trigger false alarms, the FPR informs us of the number of normal samples the local ML detected as malicious among all normal samples. The smaller the FPR, the better. In the Table 4.1 we observe that AdaBoost[LogReg] has the highest F-score and a very low FPR.

Complex MLs used to Implement the Remote Detection Mechanism

Moreover, in Table 4.1 we can observe the classification metrics of the complex ML classifiers. Since we use these complex MLs in the remote system, we will also mention their benefits and pitfalls, when used as remote ML detection mechanisms. We can observe that the remote MLs have lower F-score than some of the local MLs, but all of them have higher precision and lower FPR. This is because the remote MLs are trained only

with normal data, which helps the remote ML model to recognize most of the normal samples, i.e., to have high precision and low FPR.

Local ML most important metrics

For the local ML, the most important metrics are F-Score and FPR. F-Score because it measures the tradeoff between labelling suspicious malicious samples and filtering normal samples. And also the FPR because we cannot afford to trigger false alarms in the local system, as the overhead to reset the system may be undesirable.

As before, the detection threshold for the remote MLs is the quantile (99.85%) of the distribution of the mean square of the reconstruction errors of the normal training dataset. Since the detection threshold for the remote ML is close to the maximum reconstruction error of the normal training data, the remote ML succeeds in detecting the normal behavior. However, since malware does not perform malicious actions throughout its execution, but also performs operations similar to normal behavior, the remote ML will flag this part of the malware execution as normal, which subsequently decreases the F-score of the remote ML. As we observe in Table 4.1, the LSTM7 has an F-score of 84.32%, while the AdaBoost[LogReg] has 98.22%. On the other hand, LSTM7 has a precision of 99.82% and an FPR of 0.03%, while AdaBoost[LogReg] has a precision of 96.79% and an FPR of 0.68%. This decrease in the F-score is acceptable for our problem because we only require a few samples to flag an application as malicious, not the entire malware execution. For the remote ML, it is more important to have a precision very close to 100% and an FPR close to 0%.

Remote ML most important metrics

For the remote ML, the most important metrics are Precision and FPR. Precision and FPR because we need to accurately label the normal behavior as normal to avoid false alarms. Recall is also important as we need to identify as many malicious samples as possible. However, since we need a few samples to flag an application as malicious, Recall is not as important as it is for local MLs.

Another interesting observation from Table 4.1 is that the Autoencoder has poorer classification metrics compared to the LSTM7. This is also expected since the LSTM uses information from the past samples to make a prediction for the current samples, while

the Autoencoder only uses information from the current sample. The increased F-score of the LSTM7 compared to the Autoencoder comes at the cost of an increase in the communication overhead, since more samples must be sent. From Table 4.3, we note that the additional communication overhead is 0.22% for the LSTM7 and 0.35% for the LSTM11 when the local ML is based on an AdaBoost[LogReg]. The low communication overhead, despite the need to send the past samples in case of an LSTM, is due to the observation that the samples labeled suspicious by the local ML are close to each other. This allows us to filter storing past samples that are already saved in the local storage due to a previous suspicious sample and only save the necessary samples as we have seen in Section 4.2.2 and Figure 4.8c. In any case, despite its high precision, the Autoencoder, it is not suitable as a remote ML for the current classification problem because it has a lower ability to recognize abnormal behavior. An Autoencoder might be better suited for simpler problems, such as when the IoT is running a limited number of applications.

Meanwhile, using Table 4.1, we can see that both LSTM11 and LSTM Autoencoder11 have lower precision than using the last seven samples. If we increase the information from the past samples, i.e., we use 11 instead of 7 past samples, the network can use more information to correctly reconstruct the normal samples. However, increasing the capacity of the network also leads to over-fitting, i.e., the network becomes too good at reconstructing samples that are very similar to the training samples, which means it cannot generalize. This results in malicious samples that have close behavior to normal samples being predicted with good accuracy and noisy normal samples being predicted poorly, increasing FPR and decreasing recall. As we can see from the Table 4.1, the FPR of both LSTM11 and LSTM Autoencoder11 is higher than that of LSTM7 and LSTM Autoencoder7. In addition, the recall is lower for both LSTM11 and LSTM Autoencoder11 compared to LSTM7 and LSTM Autoencoder7. Since there is no performance improvement when more information is used and since the communication overhead increases, the implementations with the seven past samples are preferred.

4.3.4 Local-Remote Implementation Detection Metrics

After experimenting with the various ML algorithms, we evaluate the Local-Remote ML implementation. As mentioned in Section 4.2, the local ML should not trigger false alarms and should only raise an alarm for samples with high probability. This is achieved with the two-level thresholding scheme. Samples flagged as malicious in the local ML are labeled as "1". Samples filtered as normal are labeled as "0". Next, the remote ML evaluates the suspicious samples and informs the local ML of their labels. The final

metrics for the Local-Remote ML are the joined results presented in Table 4.2.

Local+Remote	F-score	Recall	Precision	FPR	Accuracy
Logistic Regression + LSTM7	84.40%	73.09%	99.87%	0.02%	95.39%
XGBoost + LSTM7	84.41%	73.07%	99.91%	0.01%	95.39%
AdaBoost [LogReg] + LSTM7	84.41%	73.09%	99.88%	0.02%	95.39%
NN_4_2 + LSTM7	84.40%	73.04%	99.94%	0.01%	95.39%
LinearSVC + LSTM7	84.40%	73.09%	99.89%	0.02%	95.39%
LinearSVC + LSTM11	84.32%	72.95%	99.88%	0.02%	95.37%
LinearSVC + LSTM Autoencoder7	95.86%	92.66%	99.29%	0.14%	98.64%
LinearSVC + LSTM Autoencoder11	91.25%	84.36%	99.36%	0.11%	97.24%

Table 4.2: Classification metrics for the different Local-Remote ML configurations.

As we observe from Table 4.2, the Local-Remote implementation increases the precision of the local ML to that of the remote ML. We also note that the FPR drops to less than 0.02%. We can see that the combination of Local-Remote significantly reduces the FPR of the local ML. For example, in Table 4.1 we see that the Logistic Regression has an FPR of 1.46%, but combining it with an LSTM7 reduces the FPR to 0.02%. Even though combining the two does not reduce the FPR to 0%, the improvement is noticeable and we can consider FPs to be rare events at this point.

Using Table 4.1 and Table 4.2, we can observe that the LSTM7 is the remote ML that improves precision the most. We find that combining LinearSVC as the local ML with the remote MLs gives a precision of 99.89% when LSTM7 is used as the remote ML. Further, in Table 4.2 we can see that the metrics for Local-Remote combinations using the same remote do not differ. This is because the remote ML is the one that characterizes the final solution. This is explained by the fact that these metrics are relevant to the ability of the combination to recognize abnormal samples. Since the remote ML characterizes the final solution and detects the sent samples as malware, these metrics are relevant to the ability of the remote MLs to detect the malware sample as malicious.

If the local ML has a high recall, i.e., , is able to correctly identify malicious behavior, it is after the ability of the remote ML to identify it as malicious as well. We observed that the samples that the local ML alerts on are also the samples that the remote ML alone flags them malicious. However, now that the local ML does not trigger an alert on intermediate probability samples, but rather sends them to the remote, it is in the remote ML ability to detect them as malicious. Since the remote ML, as seen in Table 4.1, has lower accuracy, the accuracy of the combination also decreases.

4.3.5 Local-Remote Filtering and Communication Bandwidth

To continue our experiments, we evaluate in Table 4.3 the filtering performed by the local system. As mentioned earlier, we monitor the system every 1ms. We monitor 4 cores and extract 4 bytes of information for each of the 6 available HPCs per core. We also send data from the local ML to the remote ML every 1 minute. The total size of data extracted per 1 minute is 5760kb. We divide the metrics into two categories. The first category is when only normal applications are executing in the system. The second category is when only malware is executing in the system. The *data send per minute* column is the average size of extracted HPC data labelled as suspicious per minute of execution by the local ML. For the results shown in Table 4.3, the local alert is disabled and the local system filters only extracted HPC data. This choice was made because in a real scenario, when an attack is detected due to an high probability sample, the local system automatically stops the execution of the current application and takes appropriate action. Activating the local alert does not allow us to demonstrate our idea. The goal of Table 4.3 is to show the ability of the local ML, not to filter malicious samples, in contrast with its ability to filter normal incoming samples, and to better compare the percentage of filtering. In a following Table 4.5 a more realistic example will be presented.

From the Table 4.3, we can observe that most local MLs manage to filter most of the extracted normal HPC data and label most of the extracted malicious HPC data as suspicious. For example, AdaBoost[LogReg] filters 99.32% of normal HPC data in normal operation and only filters 0.9% of the malicious data under attack. This means that on average a local ML configured with AdaBoost[LogReg] will only send 39kb out of the 5760kb extracted HPC data under normal operation. We consider sending 39kb of data per minute in a remote system is an acceptable size of data, considering that SOTA solutions [88], [97] must send all the extracted data even under normal operation. This shows that the local ML is able to successfully reduce the bandwidth in cases when the system is not under attack and only requires high communication bandwidth during an attack.

Local ML (filtering only)	**only normal apps execute**		**only attack apps execute**		**LSTM extra data overhead (percentage of data increase)**	
	Data send per minute	*filtering percentage*	*Data send per minute*	*filtering percentage*	***LSTM7***	***LSTM11***
AdaBoost [***LogReg***]	39kb	99.32%	5707kb	0.90%	0.22%	0.35%
LinearSVC	95kb	98.34%	5710kb	0.85%	0.72%	1.05%
Logistic Regression	86kb	98.50%	5709kb	0.87%	0.75%	1.11%
XGBoost	31kb	99.46%	4909kb	14.77%	0.32%	0.49%
NN_4_2	34kb	99.41%	4852kb	15.76%	0.24%	0.37%

Table 4.3: Data send per minute for each category of applications running in the CPU and percentage of filtering when 5760kb of HPC data are extracted per minute. The LSTM extra data overhead refers to the increase in the amount of data to be transmitted to the remote due to the need to sent past HPC samples.

Figure 4.9, Figure 4.10, and Figure 4.11 also demonstrate the ability of the local ML to filter most normal samples and send most malicious ones to the remote. The dotted line indicates the total size of data stored in the local memory at a given time. In the figures, the dotted line is updated every 1 second. Every Δs 1 minute, the local system transmits the saved data to the remote system for further analysis, and the local storage is emptied. The total size of data sent to the remote system per Δs is represented by the triangles. Finally, the asterisk line indicates the time period during which the malware was executed. When the asterisk line is high, the malware is executed on the local system. Figure 4.9 shows a fifteen-minute execution of normal and malicious applications on the local system. As we can observe from the figure, the amount of suspicious data saved in local storage increases when the system is under attack. The rate of increase in the amount of data saved locally is higher when the system is under attack than when the system is running under normal conditions.

In addition, Figure 4.11 presents the data saved in local storage and the data transferred in the remote system when the system is under normal operation. From the figure we can observe that when the system is under normal operation, the total size of data to be transferred per Δs is less than 50kb when we have to consider the total size of extracted data is 5760kb. This effectively means that the local ML filters at the worst case 99.13% percent of the extracted normal samples. This clearly demonstrates the ability of the local system to successfully filter the normal extracted HPC samples and minimize the required communication bandwidth. Comparing with the solution of Wang et al. [98], which succeeded reducing the communication overhead by 20-30% while keeping a high accuracy, our work succeeds in reducing the communication overhead by more that 90%. Further, in contrast to [98], we are also able to detect high probability samples

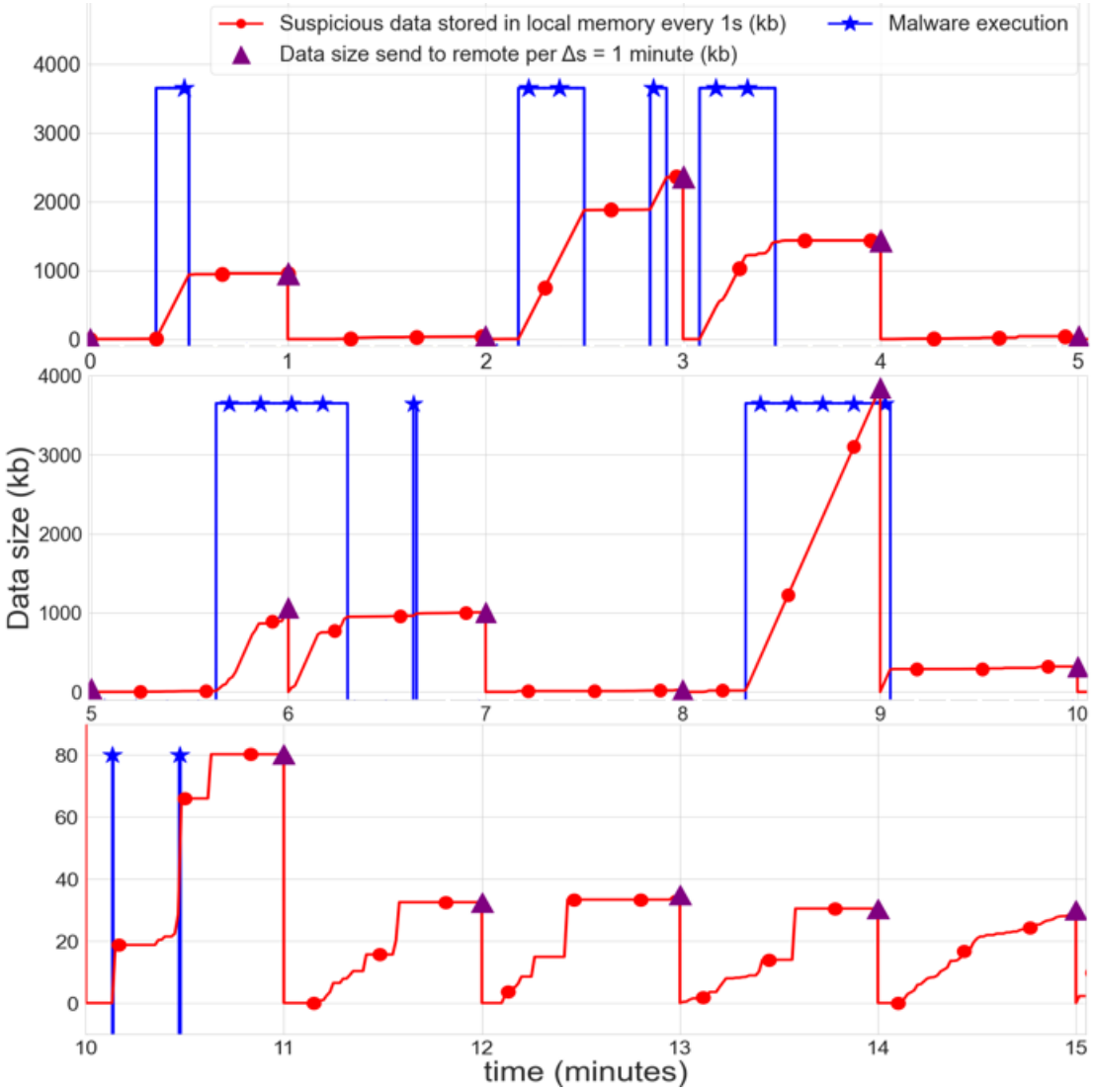


Figure 4.9: Data stored to the local memory per second (red dotted line) and sent to the remote ML for further evaluation each Δs 1 minute (purple triangles). The blue asterisk line is high when a malware executes. Local ML $\rightarrow$ NN_4_2, monitoring interval $\rightarrow$ 1ms

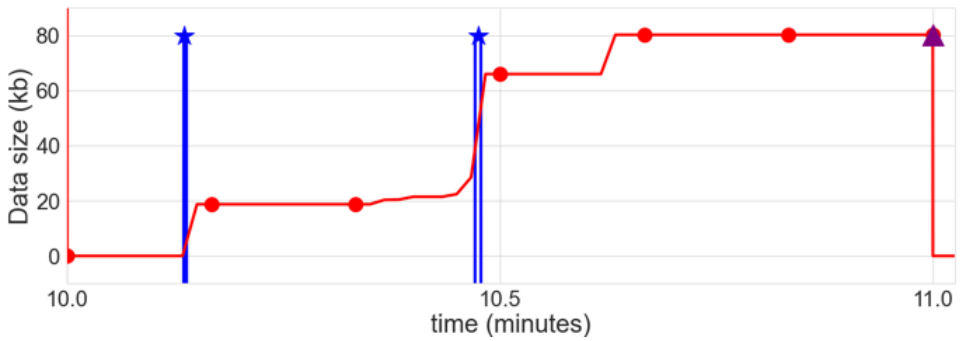


Figure 4.10: Zoom of the execution presented in Figure 4.9 for the minutes 10 to 11.

locally, reducing the detection time, and keeping the local overheads low.

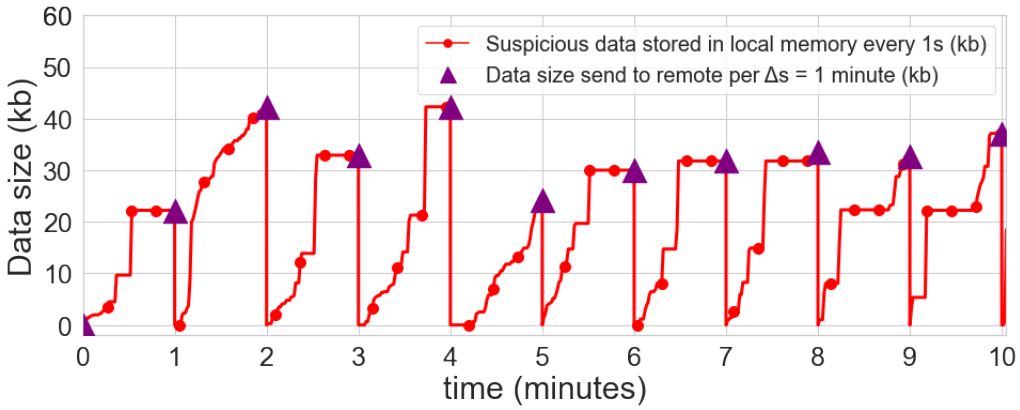


Figure 4.11: Data stored to the local memory per second (red dotted line) and sent to the remote ML for further evaluation each Δs 1 minute (purple triangles) when only normal applications execute on the system. Local ML $\rightarrow$ NN_4_2, monitoring interval $\rightarrow$ 1ms.

As can be observed from Figure 4.11, the local system stores less than 50kb of data per Δs 1 minute during normal operation. Using our normal training dataset, we can calculate the maximum size of data that the system must save in local storage per Δs in the worst case when the system is running in normal mode. If we set the size of our local storage to the calculated maximum, then the local storage will overflow before Δs when malware programs are executed. This is true in most cases, except in cases where the malware has a very short execution time (malware in Figure 4.10) or is difficult to classify as suspicious. This overflow before Δs time could be an additional factor for malware detection as we will need to transmit the stored suspicious data remotely before the expected time. In addition, we can put the system in caution mode when memory overflow occurs. In the previous example, the time span of Δs 1 minute was set only for the purpose of the example.

In conclusion, we have seen that the local ML has the ability to successfully filter the normal samples and reduce the communication bandwidth during normal operation. On the other hand, it successfully saves suspicious samples for further preprocessing in the remote system without a significant loss of information. Finally, we observed that during normal operation, the samples saved in local storage do not exceed a certain amount. This has the advantage of using a limited amount of memory, effectively reducing the area overhead, and allows us to have an additional indicator of an attack, as malware tends to overfill local storage with data before the specified Δs.

4.3.6 False Positives Reduction using an Isolation Forest

As we explained in Table 4.2, the Local-Remote implementation increases the precision of the security mechanism and decreases the FPR. For example, AdaBoost[LogReg] had a precision of 96.79% and an FPR of 0.68%. Combining it with an LSTM7 increased the precision to 99.88% and decreased the FPR by 0.02%. But this is not enough, because we still have a small number of FPs and need to define a strategy to discard them. As mentioned earlier, the potential overhead due to the system resets after a false alarm can be significant, and given the critical applications IoT is being used, frequent system resets may not be permissible.

One strategy we developed exploits the fact that the local ML sends less data than a certain amount to the remote under normal operation. This hypothesis allows us to further reduce the FPs due to misclassifications of the remote ML. For our hypothesis, as mentioned earlier, we calculate the maximum size of data sent to the remote system by normal applications. The calculated maximum values as a function of the local ML are listed in Table 4.6. A small offset can also be added to the maximum. This allow us to set the size of the local storage to this maximum value as it is most likely that under normal operation we will not need to store more samples than this in a period of Δs. If an overflown occurs, it is possible that it happened due to a malware execution, which as we have seen increases the rate of data stored locally for further evaluation.

Our hypothesis is the following:

- If the local ML sends the data before the specified Δs, then the decisions of the remote ML are correct. In this case, it is most likely an attack executed locally that flooded the local storage before Δs, and any remotely positive labelled samples are most likely TPs.
- If the local send data to the remote every Δs, then we need to pay attention to FPs, since most likely a normal application was running on the system.

To remove the FPs in this case, we use the idea proposed in [149].

The authors use two Isolation Forests after an Autoencoder, one to remove FPs and another to remove FNs. An Isolation Forest is an unsupervised ML, based on decision trees as explained in Section 2.4.2. The only parameter we need to define is the contamination, i.e., the percentage of outliers in the dataset. In their implementation, the authors use the Autoencoder for anomaly detection looking at the network traffic (data packets). Their anomaly detection mechanism targets IoT devices, but it is implemented in the fog layer, which is between the edge-devices and the cloud. The authors make the

hypothesis that the Autoencoder is accurate enough to predict most of the normal data packets as normal and most attack packets as malicious. With this in mind, any FP is outlier among all positive decisions, and any FN is outlier among all negative decisions. This allows them to use the Isolation forest to find these outliers (FN or FP) among the negative or positive Autoencoder decisions.

In our case, the outliers are the FPs among the rest of the correct predictions, in the case where the remote received the data after the expected time Δs. As previously mentioned, when the remote receives data at the expected time Δs, it is most likely normal applications were executing. In this case, the remote will predict most of the samples correctly, but there might exist some FPs. For our hypothesis, we need to pay attention to the following:

- If there is malware with a fast execution time in a period Δs, then the size of the data transmitted to the remote system may be close to the maximum. This is the case of execution between 10-11 minutes in Figure 4.10. In this case, we must not consider the true positives as FPs.

- Since we do not know the contamination parameter for the remote ML positive decisions in this Δs, we cannot train an Isolation Forest on the fly as in [149].

To solve this difficulty, we train an Isolation Forest offline with the normal HPC training data used to train the remote ML. Moreover, we contaminate this normal dataset with 10% randomly selected malicious samples used to train the local ML. After training the Isolation Forest, we deploy it after the remote ML as follows. The remote ML receives the suspicious data from the local ML. At this point, the remote ML processes the data and outputs its decisions. It also checks if the received data were transmitted at the expected Δs or earlier. If the remote received the data earlier than Δs, the decisions of the remote ML remain unchanged, since as we hypothesized before the local storage overflows only due to a malware execution. Otherwise if the remote received the data at the expected Δs, we keep the HPC data labeled as "1" from the remote ML and feed it into the Isolation Forest to remove potential FPs. If the Isolation Forest predicts the sample to be normal, we reset the label "1" to zero. On the other hand, if the Isolation Forest also predicts the sample as an outlier, we keep the label "1" and inform the local system about the existence of malware.

Isolation Forest

The Isolation Forest is an extra level of security and its goal is to further reduce the FPR closer to 0%. It is only deployed if the data are sent by the local to the remote system at the expected Δs. If this is the case, we assume that it is most probable normal applications were executing. The remote ML makes the decisions, and if positive decisions exist, we input the HPC values of these positive decisions to the Isolation Forest. The Isolation Forest was trained to recognize anomalies i.e., malicious data as samples that can easily separate from the rest. If the Isolation Forest predicts a sample as malicious, then an alert is raised. If the Isolation Forest recognizes the sample as normal, we conclude it is a FP.

Local+Remote + Isolation Forest	**F-score**	**Recall**	**Precision**	**FPR**	**Accuracy**
XGBoost + LSTM7	84.44%	73.07%	100%	0.0%	95.40%
Logistic Regression + LSTM7	84.45%	73.09%	100%	0.0%	95.41%
AdaBoost [LogReg] + LSTM7	84.45%	73.09%	100%	0.0%	95.41%
NN_4_2 + LSTM7	84.42%	73.04%	100%	0.0%	95.40%
LinearSVC + LSTM7	84.45%	73.09%	100%	0.0%	95.41%
LinearSVC + LSTM11	84.36%	72.95%	100%	0.0%	95.39%
LinearSVC + LSTM Autoencoder7	96.19%	92.66%	100%	0.0%	98.75%
LinearSVC + LSTM Autoencoder11	91.52%	84.36%	100%	0.0%	97.33%
Only using Isolation Forest	77.48%	87.88%	69.28%	8.02%	91.28%

Table 4.4: Classification metrics for the different Local-Remote ML configurations plus Isolation Forest for FP reduction.

4.3.7 Isolation Forest Strategy Evaluation

Table 4.4 presents the final classification metrics of the Local-Remote ML configuration after using the Isolation Forest technique presented above to reduce FPs. As we can see from Table 4.4, the presented strategy successfully reduces FPs to zero without any significant reduction in recall, which would mean that we misclassify TPs. For example, the combination of AdaBoost[LogReg] + LSTM7 has a recall of 73.09% and the FPRs are equal to 0.02%. On the other hand, AdaBoost[LogReg] + LSTM7 + Isolation Forest still has a recall of 73.09% and the FPR is 0%. Furthermore, we see that combining a local ML (LinearSVC in the Table) with any of the selected remote MLs plus the Isolation Forest strategy yields 100% precision and 0% FPR.

We have already mentioned that the best combination of Local-Remote ML is the one that has the highest F-score local ML and the highest precision remote ML, since we want to avoid false alarms. However, since the Isolation Forest strategy now filters the FPs, the precision of the remote ML is no longer the most important metric. Since precision for all the Local-Remote combinations is now 100%, we now need to consider the ability of the Remote ML to detect malicious samples i.e., its recall or F-score. Using Table 4.1, we can verify that the LSTM Autoencoder7 has the highest F-score among all remote MLs, but also has a high FPR. Using Table 4.2, we can also see that LSTM Autoencoder7 is the remote ML that provides the best F-score of 95.85% when combined with LinearSVC, but also has the highest FPR of 0.14% compared to all others. Since, as mentioned earlier, we want to avoid false alarms, this combination was not previously considered due to its high FPR. However, now that the Isolation Forest reduces the FPR to zero, this combination performs best in this classification problem. Using Table 4.4, we can see that the combination of LinearSVC plus LSTM Autoencoder7 has the highest F-score. Though, when using the Isolation Forest, the remote ML with the highest F-score is the one that gives the best results, as it allows us to capture the most malicious activity.

Finally, we can say that the combination of the local ML with the highest F-score and the remote ML with the highest F-score combined with the Isolation Forest performs best in our classification problem. To justify this, Table 4.5 shows the metrics for two Local-Remote configurations. The table summarizes the metrics from Table 4.4 and also adds two more columns showing the percentage of TPs detected locally and remotely. For both configurations, the local ML is AdaBoost[LogReg] (the highest F-score local ML) and the remote MLs are LSTM7 and LSTM Autoencoder7 plus the Isolation Forest. The Isolation Forest manages to set the FPs to zero, so both implementations have an FPR of zero. But the LSTM Autoencoder7 performs better on the other metrics, being the

remote ML with the highest F-score. Also, from the column indicating the percentage of positive samples detected remotely, we can see that the LSTM Autoencoder7 detects 19.58% more positive samples than the LSTM7 which is due to its ability to recognize more malicious samples. Finally, we see in Table 4.4 that using only the Isolation Forest is not successful in our classification problem. But the combination of Local+Remote ML together with the Isolation Forest works best in the context of the proposed implementation.

Local + Remote + Isolation Forest	**F-score**	**Sensitivity (recall)**	**Precision**	**FPR**	**Accuracy**	**% of Positives detected locally**	**% of Positives detected remotely**
AdaBoost[LogReg] + LSTM7	84.45%	73.09%	100%	0.0%	95.41%	67.29	5.80
AdaBoost[LogReg] + LSTM Autoencoder7	96.19%	92.66%	100%	0.0%	98.75%	67.29	25.37

Table 4.5: Classification metrics for the Local + Remote ML + Isolation Forest when local detection is activated.

From the above results we can conclude that the ideal combination of Local+Remote ML is the local ML and the remote ML with the highest F-score. This way, we can send most of the captured malicious activity for further evaluation while reducing the communication overhead by successfully filtering the normal extracted data. The highest F-score remote ML will correctly detect most malicious samples, and the Isolation Forest will decrease the FPs.

4.3.8 Evaluation of the Local MLs Overheads on the Local System

As mentioned in our motivations and explanation of the Local-Remote idea, the local ML should not cause significant overheads on the local system. The reason is that IoT devices are resource-constrained in terms of computing and memory resources. While there are detection solutions that operate with high accuracy, their implementation on such systems can be restrictive. The memory overhead is due to local storage of extracted HPC samples when remote ML solutions are proposed. The computational overhead is due to the sharing of resources between the monitor and the applications being run. Since the monitor is responsible for extracting the HPC values and making the corresponding decisions, the time spent on these tasks (HPC extraction time plus decision latency) defines the final performance overhead incurred in the local system. An ML, which includes numerous arithmetic operations for decision making, increases

the decision latency and thus the performance overhead. Though, when proposing a solution for IoT, these parameters should be of utmost interest.

Since we are most concerned with the time it takes to deliver a decision (latency), as well as performance and memory overhead, in Table 4.6 we present these metrics for each local ML used in this experiment. To measure decision latency, we measure the time that elapses from the moment the HPC measurements are available as input to the ML to the moment we obtain the final classification. To measure the performance and memory overhead, we run the same set of applications with and without the detection mechanism. For the performance overhead, we use the same methodology as in [150], which uses the *linux time* command to measure the time that elapses between each run. The *linux time* command informs us about the time elapsed between the start point and the end point of the execution of the test. To measure the memory overhead, we use the same methodology as in [151], which uses the *linux free* command to measure the memory used by the system every 0.01 seconds. The *linux free* command informs us about the memory usage of the system at that particular moment. By measuring the memory usage every 0.01 seconds, we can measure the dynamic memory usage and calculate the average memory usage during the execution of the test. At each Δs, the memory is filled with suspicious data, for which we show in the last column of Table 4.6 the calculated maximum size of the local memory needed to store the suspicious samples, as explained earlier. Every Δs the system is under the worst-case scenario for memory requirements due to the local ML detector (requiring system memory to execute) and the suspicious data ready to be sent (HPC data saved in the local storage). We add this size to the total memory used by the applications and the local ML detector, and then calculate the overhead. Before each run, we clean the memory. To reduce the noise, we conduct each experiment 1000 times and after we average.

From Table 4.6 we observe that the performance overhead of all selected local MLs is less than 1%. Also, the memory overhead of the detection mechanism is less than 0.36%. This is because of the targeted use of simple MLs locally, which involve only a few calculations to make the appropriate decisions. Though, we consider as acceptable the induced overheads for the targeted resource-constrained devices. Another interesting observation is that as the complexity increases, the latency also increases. XGBoost has the lowest decision time, which is to be expected since the decision algorithm contains only if-else statements and does not involve complex operations. LinearSVC and Logistic Regression also have low latency because they can be considered single-layer neural networks. The highest latency is observed for the Neural Networks, which is expected due to the amount of more complex operations they must perform per neuron. The larger the Neural Network, the higher the latency.

Algorithm 4: Algorithm to calculate performance overhead

```
input  : repeats ;      // How many times to run the same experiment in order to reduce
           noise
output : performance overhead
time no monitor = 0 ;              // store time elapsed when the monitor is deactivated
time monitor = 0 ;    // store time elapsed when the monitor is activated i.e., runs in
 parallel with the applications
for i := 0 to repeats do
    time = $(time ./random-execution –monitor-deactivated);
    time no monitor = time no monitor + time;
end
time no monitor = time no monitor / repeats ;                  // calculate average between runs
for i := 0 to repeats do
    time = $(time ./random-execution –monitor-activated);
    time monitor = time monitor + time;
end
time monitor = time monitor / repeats ;                        // calculate average between runs
performance overhead = 100%*(time monitor - time no monitor) / time no monitor;
return performance overhead;
```

4.3.9 Remote ML Latency Evaluation

In this last section, we experimented and present in Table 4.7 the latency of the remote ML, to get the final prediction for the received data. Table 4.7 has four columns. The first column shows the latency when the remote ML receives the maximum size that a normal application sends to the remote when the local ML is AdaBoost[LogReg]. This is the worst-case scenario when the device is in normal operation. As mentioned before, we consider that at most time the device runs normally i.e., not being attacked. In such a case, most normal samples are filtered and only a certain amount is saved in the local storage for further evaluation. In Table 4.6, we calculated this maximum and set the size of the local storage. This is the worst case scenario as on average the device under normal operation sends less data that this size. The second column shows the latency, but in this case 10k devices send the maximum data size to the remote ML. The third and fourth columns show the latency for the case when the local ML sends all the extracted HPC data (i.e., 4*60000 samples assuming that Δs is 1 minute and the monitoring interval is 1ms) for one and 10k devices. The remote ML is implemented using the Tensorflow-GPU package and a Graphics Processing Unit (GPU) NVIDIA Tesla V100 PCIe 16 GB. We find that all remote MLs can make a prediction in less than 25.66 seconds in the case of normal operation and receiving data from 10k devices. On the other hand, we see that if we need to send all extracted samples for 10k devices, the latency of the remote MLs is almost 1 minute and even higher for the LSTM Autoencoder.

Algorithm 5: Algorithm to calculate memory overhead

```
input  : repeats ;      // How many times to run the same experiment in order to reduce
         noise
input  : repeats_per_second ;       // How many times to measure memory usage per second
input  : Local storage size
output : memory overhead

repeats_per_second = 1/repeats_per_second ;       // convert to time i.e., 1/1000 is 1ms
counter = 0;
memory no monitor = 0 ;          // stored memory used when the monitor is deactivated
memory monitor = 0 ;   // stored memory used when the monitor is activated i.e., runs
 in parallel with the applications

for i := 0 to repeats do
    exec(./random-execution –monitor-deactivated) &);
    while app_still_executes do
        memory used = $(free | grep Mem | awk 'print $2 + $4 + $5') ; // 2nd arguments is the
         memory used, 4th argument is the shared memory, 5th argument is the
         buffer/cached memory
        memory no monitor = memory no monitor + memory used;
        sleep(repeats_per_second);
        counter++;
    end
    clear memory;
end
memory no monitor = memory no monitor / counter ;     // calculate average memory usage
 between runs

counter=0;
for i := 0 to repeats do
    exec(./random-execution –monitor-activated) &);
    while app_still_executes do
        memory used = $(free | grep Mem | awk 'print $2 + $4 + $5');
        memory monitor = memory monitor + memory used;
        sleep(repeats_per_second);
        counter++;
    end
    clear memory;
end
memory monitor = memory monitor / counter ;  // calculate average memory usage between
 runs
memory monitor = memory monitor + Local storage size ;    // add the Local storage size to
 reference the worst case scenario

performance overhead = 100%*(memory monitor - memory no monitor) / memory no monitor;

return memory overhead;
```

Classifier	*latency*	*Performance Overhead*	*Memory overhead (each Δs)*	*Local memory size*
XGBoost	0.52us	0.677%	0.09%	57kb
LinearSVC	1.17us	0.573%	0.04%	171kb
Logistic Regression	1.17us	0.573%	0.04%	152kb
AdaBoost [LogReg]	4.4us	0.796%	0.36%	54kb
NN_4_2	4.73us	0.727%	0.16%	45kb

Table 4.6: Evaluation of different important metrics for a limited resources system. The last column showcases the size of the local memory to be used for storing suspicious samples. It is calculated as the maximum size of data the system stores per Δs under normal operation during training. If the local memory is overflowed, then we send the suspicious data to the remote before Δs.

This shows that local filtering of the extracted data also affects the resources needed in the remote ML, although we considered there is no limit. It shows that a single GPU is sufficient to reduce the cost of the implementation. On the other hand, extracting and sending the maximum data, either during the attack when we do not set the local memory limit as proposed in Section 4.2.2, or as proposed in the SOTA works, could result in the remote ML not being able to process the amount of data to make a decision before Δs and increase the cost, requiring more resources.

Remote ML	max normal samples (54kb) for 1 device	max normal samples 10k devices	max extracted samples (5760kb) for 1 device	max extracted samples 10k devices
LSTM_7 (512,512)	0.62 sec	14.73 sec	8.96 sec	59.34 sec
LSTM_7 (256,256)	0.64 sec	15.75 sec	9.11 sec	47.21 sec
LSTM_11 (512,512)	1.06 sec	19.39 sec	12.59 sec	48.23 sec
LSTM AutoEncoder_7	1.02 sec	25.66 sec	20.48 sec	397.26 sec

Table 4.7: Latency of the different remote ML classifiers when the local ML is the AdaBoost[LogReg].

4.4 State Of the Art Comparison

In this last section, we compare our implementation with the methods proposed in SOTA for detecting microarchitectural attacks. In Table 4.8 we can see the attacks used

Detection Mechanism	Attacks	Accuracy	F-score	FPR or Precision	Overhead	System	Local or Remote
Mushtaq et al. [91] **Logistic Regression** (No Load)	Flush+Reload	99.51%		0.48% FPR	0.94%	Intel	Local
Mushtaq et al. [91] **SVM** (No Load)	Flush+Reload	98.82%		0.397% FPR	1.29%	Intel	Local
Mushtaq et al. [91] **Logistic Regression** (No Load)	Flush+Flush	91.73%		0% FPR	1.10%	Intel	Local
Mushtaq et al. [91] **SVM** (No Load)	Flush+Flush	97.42%		0% FPR	0.79%	Intel	Local
WHISPER [92] **Ensemble Learning** ***One model per malware*** **(DT, RF and SVM)** (No Load)	CacheCSA, **(F+F, F+R, P+P)**, Spectre, Meltdown	>97.05%			<8%	Intel	Local
FortuneTeller [76] **LSTM**	Spectre CacheCSA, **(F+F, F+R, P+P)**, Meltdown, Rowhammer		99.70%	0.125% FPR	3.50%	Intel	Local
Depoix et al. [152] **DNN**	Spectre CacheCSA, **(F+R, P+P)**, Meltdown	99%	97.16%	0.33% FPR			
Wei et al. [153] **OC-SVM**	Prime + Probe, Spectre, Rowhammer, **Evasive**	<98.63%		<0.5% FPR		ARM	
Wei et al. [153] **LSTM**	Prime + Probe, Spectre, Rowhammer, **Evasive**	<99.06%		<0.5% FPR		ARM	
Kuruvila et al. [154] **Random Forest**	Flush + Flush, Spectre, Meltdown, Rowhammer BashLite, PNScan	89.90%	89.91%	89.25% **Precision**	<1.22%	MIPS	Local
Wang et al. [155] **MPL**	CacheCSA, **(F+F, F+R, P+P)**, Spectre?	<98.9%	<97%	5.3% FPR	<3.2%	Intel, ARM	
Wang et al. [155] **Logistic Regression**	CacheCSA, **(F+F, F+R, P+P)**, Spectre?	<98,9%	91.90%	14.9% FPR	<3.23%	Intel, ARM	
Ours **AdaBoost** + **LSTM AutoEncoder7**	Spectre CacheCSA **(F+F, F+R, E+R, P+P)**, Meltdown, Rowhammer, **Evasive**	98.75%	96.19%	0% FPR	0.80%	ARM	Local Remote

Table 4.8: Comparison of our Local-Remote implementation to the SOTA.

in each work, the accuracy, the F-score, the FPR or precision, the performance overhead, the target system, and finally the implementation in a local or remote system. From the table, it can be seen that the complexity of the algorithm used increases as more attack libraries are used in the evaluation of the proposed detection mechanism. For example, Mushtaq et al. in [91] used simple MLs such as a logistic regression and SVM, but aimed to detect only one attack library. As attack libraries increased in their work, Mushtaq et al. in WHISPER [92] used an ensemble algorithm with decision trees, random forest, and SVM. The final decision was made using majority voting. The authors found that each algorithm by itself had good accuracy but high FPR, while combining the algorithms reduced the FPR. Using more than one classifier results in a larger overhead for the system, which in this case is $\sim$8%. Similarly, Wei et al. [153] uses an LSTM to detect a wide range of SATHV, including evasive malware, and shows an accuracy of $\sim$99.06%, but an FPR of $\sim$0.5%. The authors do not mention the overhead incurred in the local system for using such a ML locally, but based on FortuneTeller [76], we can see that their method is successful with a high F-score of 99.70%, but they cause 3.5% performance overhead in the local system. Considering that they test their solution on an Intel desktop, we expect this overhead to be more significant on resource-constrained devices. All the works referenced in Table 4.8 aims to detect attacks with high accuracy. However, as mentioned earlier, this is not the only factor, as FPR should also be considered. This is because even an FPR of close to 0% could lead to frequent system resets and consquently large overheads. On the other hand, we propose a Local-Remote mechanism that targets near 0% FPR and a high F-score. As we can see from the table, we obtain a score that is close to or exceeds that of most works since we have one of the largest sets of attack libraries in our dataset.

4.5 Summary

In this chapter, we proposed a solution for SATHV detection in low-resource devices such as IoT/IIoT. We proposed a Local-Remote implementation that attempts to minimize the performance, memory, and communication overheads in the local resource-constrained system. The proposed solution quickly alerts the system when high probability samples are detected, filters low probability samples, and stores suspicious samples for further analysis. Then, the local system communicates with a remote system on which a complex ML solution is implemented. The complex ML is able to learn more complex behaviors and better distinguish normal from malicious actions and make a decision with higher confidence than the local ML.

We also present the methodology used in our experiments and evaluate the proposed

Local-Remote idea in terms of several metrics. We show that the Local-Remote implementation increases the precision compared to only a local ML implementation. Moreover, we have shown that a simple local ML can filter normal extracted HPC data with a filtering percentage of ∼99%, effectively reducing the required bandwidth in normal operation and increasing it only in the presence of malware.

In addition, we propose a strategy to achieve a near zero FPs detection. The proposed strategy uses an Isolation Forest based on the hypothesis that the local ML sends data each Δs under normal conditions and only sends data before the expected time under attack. As we have presented, the local storage is overflown by data before Δs only during an attack, which signals the local system to send the data immediately. If the data are received by the remote ML at the expected Δs, we check that the positive detections after the remote ML are also predicted to be outliers from the Isolation Forest. Table 4.4 confirmed the hypothesis in the context of the proposed solution.

We also find out that the combination of the highest F-score local and highest F-score remote ML performs the best in our classification problem. This is because having the highest F-score local allow us to filter most of the normal samples, while it allow us to successfully label suspicious or raise an alert for malicious samples. Then the highest F-score remote has the ability to recognize most of the malicious samples, while having minimal FPR. Then, the Isolation Forest strategy applied in the context of this solution, can successfully detect the FPs, reducing the FPR to near 0% in our test case.

The proposed solution of the Local-Remote ML tries to minimize the overheads of the local system and still succeeds in detecting malware with high accuracy. As the current implementation is implemented in software, the local detection mechanism and the applications executing in the local system share resources. Even if the simple ML in the local system introduces low overheads regarding performance and memory according to Table 4.6, this can still be a problem for low resources devices. Currently, we evaluate the proposed idea using a Raspberry Pi4 which has more memory and computational resources.

Further, as the implementation is in software, it is still in danger of being targeted by malware. Today's OS are vulnerable to multiple vulnerabilities and the thousands of lines of code make the verification problem more difficult. Low resource devices might use a less complicated OS, but still, vulnerabilities exist which can endanger the detection module. To address this, a hardware implementation should be more appropriate as we will see in the next chapter.

5

A Hardware-based Local Detection Mechanism for the Security enhancement of the Local-Remote approach

The Local-Remote implementation of the detection mechanism allows us to filter extracted HPC samples, make fast decisions locally in the case of high probability samples, and finally further evaluate suspicious samples in a remote complex ML. However, our local detection mechanism is implemented in software, which makes it vulnerable to software attacks. This could allow attackers to directly attack the local detection mechanism and compromise its functionality. On the other hand, a hardware implementation limits the ability of attackers to interfere with the detection task, since the attacker have limited access rights from software to the system hardware. In this section, we analyze the security risks of a software implementation and propose a hardware implementation that further allows us to take advantage of hardware acceleration to limit the overhead and/or use complex ML locally. Finally we evaluate and compare its performance metrics and overheads compared with the software version.

5.1 Motivations of the work

The main motivations and problematics that have driven our research presented in this chapter are the following:

- A software implementation of a security mechanism is vulnerable to software attacks. A security designer should be careful when implementing the code to avoid creating bugs/breaches that can be exploited by attackers. In addition, SATHV such as Rowhammer could directly target the ML algorithm to change its implementation [51]–[53], e.g., by changing the weights, which could affect the accuracy.
- A software implementation of a security mechanism runs in parallel with the other user applications, sharing resources. Since we chose to monitor the system with high frequency, the context switching between the monitor and applications introduces overhead in addition to the time required by the ML to make a decision. Thus, we could only use simple MLs to keep the local system overheads low. Implementing the mechanism in hardware allows us to reduce the overheads since the monitor will not interfere the normal operations by sharing resources. Further, hardware acceleration could allow us to use locally complex MLs (such as DNNs) without imposing additional performance overhead on the local system.
- Finally, interrupting the CPU to run the monitor and make decisions introduces noise into the measurements because the monitoring interval is not equal to the specified frequency, but instead fluctuates around it with some noise. This adds additional noise to our data, complicating our detection problem. On the other hand, a hardware monitor minimizes this noise because it reads the registers directly at the specified time interval without interfering with the OS.

5.2 Software attacks targeting the local detection mechanism

We have already introduced in Section 4.5 that a software implementation of the local detection mechanism is vulnerable to software attacks and SATHV. This is because the detection mechanism and attack applications share resources e.g., the DRAM. In this section, we present some attacks that target the ML algorithms, as well as an attack we developed to disrupt/interfere the extraction of HPCs. Further, we list some attacks that could potentially be modified to target the local mechanism. This demonstrates the need to enhance the security of the local detection mechanism through a hardware

implementation.

Rowhammer targeting ML models

Recently, many related works study the robustness of ML models against adversarial attacks, including white [29] and black [156] boxes. Most of these works look for ways to interfere with the classification of the ML model's inputs to misclassify the sample. This includes inducing specially crafted noise into the inputs that activates certain parameters in the model, with the goal of confusing the network. In our case, adding noise to the HPC measurements means modifying the attack code to change the behavior triggered in the system. In this case, this task is not as straightforward as, for example, image classification. In the case of image classification attacks, the attackers add the specially crafted noise and thus have control over the image. In the case of a software application, the attackers have to add the noise using obfuscation techniques (i.e., specially crafted instructions that will modify the code execution) and still succeed performing their malicious activities. However, they do not have direct control over the hardware and the time available to add the required noise i.e., the monitoring interval, making this task more difficult.

Attackers could also use other techniques to circumvent detection of the attack, e.g., by tampering with model parameters such as weights, activation functions, biases, thresholds, etc. Recently, SOTA works [51]–[53] have explored the use of Rowhammer to alter model parameters. This is accomplished by bit-flipping targeted bits in the DRAM, where the model parameters are stored. [51] has shown that 10 targeted bit-flips can reduce the accuracy of a model to less than 10%. This becomes more serious in our case, as the simple models we used in the previous Chapter 4 have fewer parameters than more complex ML algorithms, making the proposed idea more vulnerable to this type of attack, as changing one parameter greatly changes the detection capability of the local ML.

In the following, we present a simple neural network example trained with MiBench, PARSEC, Spectre, and Rowhammer. The neural network can be seen in Figure 5.1. It receives as inputs the six HPC events and has three hidden layers. In the figure, we can see the original weights of the last layer W1 and W2. W2 is the targeted weight for this example, and we can also see the binary representation using the IEEE-754 floating point representation. The ML model has an accuracy of 99.71% and a confusion matrix that we can see in Figure 5.2a. If an attacker succeeds in bit-flipping the 2nd bit of the exponent of the second weight of the last layer of the network, then the classifier classifies all attack samples as FN, as can be seen in Figure 5.2b. On the other hand, if the

attacker only succeeds in bit-flipping the 1st bit of the exponent of the second weight of the last layer, the accuracy is 99.20% and the network has an accuracy matrix as seen in Figure 5.2c. This simple example shows that small changes to the weights of a simple ML can significantly alter its detection capabilities but also shows that the induced false must be precise. This is because, unlike more complex ML, which can store/learn more information in different neurons, the simple ML we presented learns only limited information, and each neuron plays an important role in the final decision.

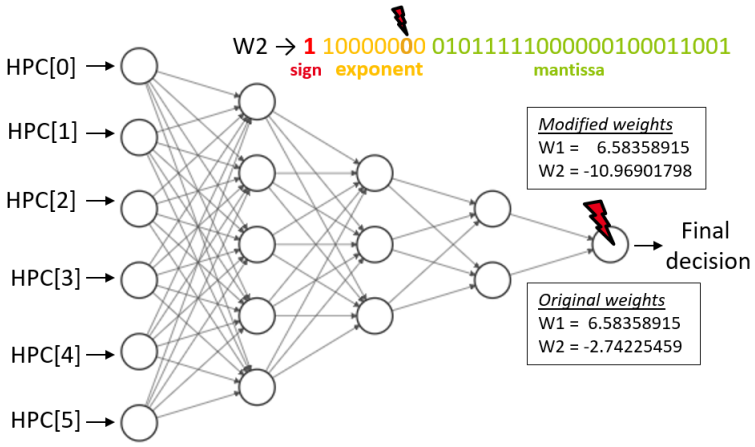
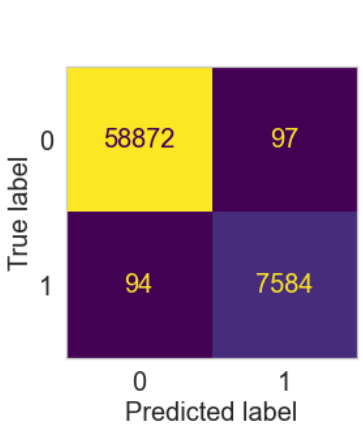
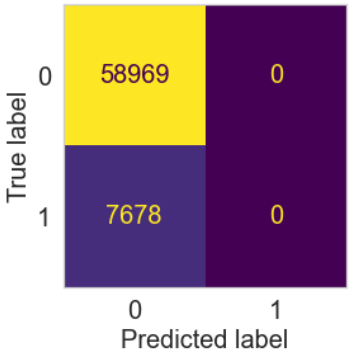


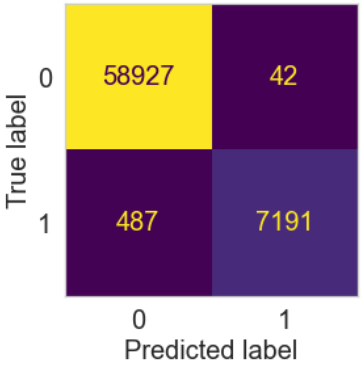
Figure 5.1: Neural Network Visualization before and after bit-flipping last layer weight value. In the figure we can see how the weight W2 changes and the original.



(a) Original Confusion Matrix.



(b) Confusion Matrix after bit-flipping the 2nd exponent bit in the last layer weight.



(c) Confusion Matrix after bit-flipping the 1st exponent bit in the last layer weight.

Figure 5.2: Confusion Matrices of a Neural Network before and after bit-flipping a weight in the last layer as shown in Figure 5.1.

This is serious security pitfall, considering Rowhammer can be performed from software and remotely without requiring physical access. Since Rowhammer is included in our threat model, detecting the attack in time we could avoid this the case where the attacker successfully faults the parameters. Further, using Error Correcting Codes

(ECC) before deploying the model could be used to verify their integrity, detecting potential tampering. But this might add some extra overhead, since we would need to do that every time the new HPC samples are extracted. Still, it showcases that software implementation can be vulnerable to software attacks and SATHV.

Software attack targeting the HPC extraction from user-space

Since our detection implementation relies in the extraction of the HPC values from the performance monitoring unit, we explored ways to interfere with this task. Under normal operation, accessing HPC counters is restricted and a kernel-module should be loaded to allow reading/modification using the internal CP15 interface [157] or the Linux perf-tool. During the development of the detection module, we discovered in the reference manual of ARM that HPCs can also be accessed using memory-mapped I/O [157, p. 161]. Since user-space application can also map in their memory space the memory-mapped I/O region (which also includes accessing peripherals), this poses a potential risk. As we will showcase, this is even a greater threat as a designer could falsely believe the detection mechanism is safe, by restricting access to the HPC counters from user-space, but an attacker could bypass this restriction by using the memory-mapped interface. In the following we will showcase four cases of how an attacker could interfere with the HPC extraction. The test cases presented require a script to be executed from user-space, without requiring the use of a kernel-module. More information could be found for all test cases in the Section A.1.

```
root@zybo45:/home/fpga/read_memory_Thesis_example# ./malware_target_PMU -c 0 --test_case 1
value of DBGDRAR is F8800003.
The value of DBGDRAR_physical_address is F8800000
The of DBGDSAR_physical_address = 90000
The CPU debug logic physical address is F8890000 i.e. DBGDRAR + DBGDSAR
Reading DBGAUTHSTATUS ... !
If FF all debugging modes are enabled!
DBGAUTHSTATUS == FF
Allow access to the PMU registers through a memory-mapped interface by writing the value 0xC5ACCE55 to PMLAR register
Enable user access register [PMUSERENR] is currently 0
If PMUSERENR[0] == 0 then user access to the PMU is disabled. Enable it ...
         Write '1' to bit 0 of the user access register [PMUSERENR] to activate it
         Successfully enabled user access to PMUs without a Kernel-Module!

--------------------------------------------------------------------------------
Case 1, Read PMU counters from user space without using a kernel-module to enable access
--------------------------------------------------------------------------------

PMU was enabled
Counter 0 configured with the event 1
Counter 1 configured with the event 4
Counter 2 configured with the event 70
Counter 3 configured with the event 10
Counter 4 configured with the event 5
Counter 5 configured with the event 50
Performance monitor results
Loop: 0, 7598 18524 20860 3947 371 603 173102
Loop: 1, 11530 24107 25661 5088 1032 897 383803
Loop: 2, 4668 10459 11633 2305 242 272 469774
Loop: 3, 11497 24137 25561 5052 1036 805 667082

 ---------------  Finish the test case ---------------
```

(a) Test Case 1.

```
root@zybo45:/home/fpga/read_memory_Thesis_example# ./malware_target_PMU -c 0 --test_case 2
--------------------------------------------------------------------------------
Case 2, Change the hardware HPC events monitored by the detection mechanism!
--------------------------------------------------------------------------------

Event of PMU Counter 0 was 63
Event of PMU Counter 1 was 64
Event of PMU Counter 2 was 65
Event of PMU Counter 3 was 66
Event of PMU Counter 4 was 67
Event of PMU Counter 5 was 69

Change the hardware HPC event by modifying PMXEVTYPER*

We modified the event of PMU Counter 0 to 63
We modified the event of PMU Counter 1 to 64
We modified the event of PMU Counter 2 to 65
We modified the event of PMU Counter 3 to 66
We modified the event of PMU Counter 4 to 67
We modified the event of PMU Counter 5 to 69

 ---------------  Finish the test case ---------------
```

(b) Test Case 2.

Figure 5.3: PMU interference, test cases 1 and 2.

Case1: Read PMU registers from user-space without using a kernel-module to enable access The first test case is the activation of HPC counters without a kernel module.

Attackers with access to the HPC register values can observe the registers and successfully figure out the monitoring interval by analysing when the value of HPC registers is reset (Algorithm 1 in Section 3.5.1). Since many of the SATHVs require access to the cycle counter to find timing differences, an attacker could read the cycle counter even if access to the PMU is originally disabled. Finally, [158] used information from the PMUs (L2 cache misses) to create a covert channel between TrustZone and userspace applications. This allowed them to find out how many cache lines have been updated, i.e., which addresses the TrustZone application used. An example of this test case can be found in Figure 5.3a. In the figure, we also see the necessary steps and register configurations that allow us to enable the HPCs via the memory-mapped interface.

Considering that an attacker can have access to the PMUs and fully know the underlying ML algorithm used by the local ML as well as the monitored HPC events, in the following we present three test case scenarios that pose a major threat to our security.

Case2: Change the HPC events monitored by the detection mechanism In this second test case, attackers could use the script to modify the monitored HPC events and configure them to their own choice as we can observe in Figure 5.3b. This is succeeded by reconfiguring the HPC registers with the configuration of their own. This could allow attackers to manipulate the inputs of the detection mechanism. For example, an attack that increases an HPC event to a range outside the normal applications distribution could be modified by the attacker with another that provides values within the range of normal values. In this way, the attacker can bypass detection and then reconfigure the monitor with the original configuration to cover the tracks of the tampering.

```
root@zybo45:/home/fpga/read_memory_Thesis_example# ./malware_target_PMU -c 0 --test_case 3

----------------------------
Case 3, Reset PMU counters!
----------------------------

PMU was enabled
Counter 0 configured with the event 1
Counter 1 configured with the event 4
Counter 2 configured with the event 70
Counter 3 configured with the event 10
Counter 4 configured with the event 5
Counter 5 configured with the event 50

Sleep 3 seconds to allow the counters to increase!

35921 , 141778 , 140379 , 22590 , 3133 , 5595

Now reset the counters!

0 , 247 , 498 , 90 , 12 , 17

 ---------------  Finish the test case ---------------
```

(a) Test Case 3.

```
root@zybo45:/home/fpga/read_memory_Thesis_example# ./malware_target_PMU -c 0 --test_case 4

------------------------------
Test 4, Disable PMU counters!
------------------------------

PMU was enabled
Counter 0 configured with the event 1
Counter 1 configured with the event 4
Counter 2 configured with the event 70
Counter 3 configured with the event 10
Counter 4 configured with the event 5
Counter 5 configured with the event 50

Sleep 3 seconds to allow the counters to increase!

73263 , 222750 , 223677 , 39196 , 5279 , 6672

Now sleep again to see if we properly DISABLED? Sleep 3 seconds to verify counters didn't increase

73263 , 222750 , 223677 , 39196 , 5279 , 6672

 ---------------  Finish the test case ---------------
```

(b) Test Case 4.

Figure 5.4: PMU interference, test case 3 and 4.

Case3: Reset PMU counters In this scenario, the attacker resets the HPC values to zero, as you can see Figure 5.4a. If an HPC event increases in a range of values outside

the normal application distribution, an attacker could reset it during the attack to reduce the final values extracted by the detection mechanism, allowing the attacker to bypass detection. This test case can be more difficult to perform but it shows us the potential tools an attacker can use trick our system.

Case4: Disable HPC counters In this last scenario, the attackers can disable the HPCs by freezing the hardware event count. This could also allow them to freeze the values of an HPC event that increases significantly during the attack so that it remains in the range of the values of normal applications. Another case is when the detection mechanism uses event sampling. In this case, the monitor extracts measurements when one of the HPC events reaches a certain value, as in [79], which extracts HPC values after every syscall, and [159], which extracts HPC values every 100k instructions. In this case, an attacker who resets the HPC values of the monitored event could cause the monitor to never extract HPC values because the specified value is never reached. For example, the attacker disables the counter at value zero for the syscalls and less than 100k for the instructions. An example can be found in Figure 5.4b. As we can see from the figure, the extracted HPC values remain the same after 3 seconds of execution.

Hardware attacks against ML models

Hardware attacks against ML models are out of the scope of this work as explained in our threat model. As previously mentioned in Chapter 1, the industrial environment provides limited access to systems, and consumer IoT devices are located in the user's home. Furthermore, if IoT devices can become targets of their own users, we consider physical protection measures exist. Examples of such protections include an integrated secure element or detection of attempts to dismantle the device. This limits physical access to the device, which allow us to not consider hardware attacks in our threat model. Such hardware attacks include, but are not limited to, fault attacks on the parameters of the ML algorithm such as the weights, activation functions, etc., using laser [160]–[162], clock glitch [163], and voltage glitch attacks [164].

In summary, we show some techniques that attackers could use to disrupt HPC extraction or modify ML model parameters when the local detection mechanism is implemented in software. We believe that this shows that a hardware implementation of the local detection mechanism should increase security against software attacks and fault attacks such as rowhammer. In the following sections, we will provide further reasoning for the necessity of a hardware implementation selection.

5.3 Local ML hardware implementation

As mentioned in the previous section, a software implementation of the local ML may be vulnerable to software attacks. On the other hand, a hardware implementation of the mechanism reduces the attack surface since the attacker is restricted from accessing the hardware mechanism from software. A dedicated hardware component with high privileges in the system allows us to limit the ability of an attacker to disrupt our detection solution. Furthermore, a dedicated hardware component relieves CPU from the performance overhead of running applications and the detection module in parallel. However, the hardware implementation comes at the price of requiring additional area, as an additional component must be integrated into the system. This additional area, though, poses a constraint on our design since it requires an Application-Specific Integrated Circuit (ASIC) or programmable logic to be in place.

As mentioned earlier, selecting the most prominent HPC events to distinguish between normal and malicious applications is an important step in developing an effective detection mechanism. In Chapter 3, we looked at the theoretical side-effects proposed in SOTA, and we also used some feature extraction methods to finally select the six highest scoring events to propose our local mechanism. Selecting the most prominent HPC events could also help us to implement less complicated ML systems locally. However, the proposed HPC events better work with the set of applications used (normal and attack applications) and are also successful in detecting evasive attacks. However, the HPC events selected by the feature extraction methods focus on specific components of the CPU, i.e., the HPC events more affected by the SATHV execution. Though, this could make them vulnerable to new attack vectors that do not affect the proposed HPCs.

Since using a local-remote allows the local monitor to be less accurate, while the remote monitor helps to increase the detection rate, we investigate two ways to select HPC events that can better scale our detection solution. First, we could also use HPC events that can give us a global view of the system i.e., using HPCs from different components. Second, we can use for scalability HPC events implemented in different architectures, though the detection mechanism could be in some sense independent of the underlying CPU architecture. This has led us to create a list of HPC events implemented for the following CPU cores: RISC-V [CVA6, Rocketchip, BOOM], ARM Cortex [A9, A53, A57, A72, M55, M85, R4, R5, R7, R8, R52], Intel [Atom, Pentium], and the latest 12th generation Intel cores for IoT devices. Table 5.1 lists the HPC events selected using the feature extraction method MI Section 2.4.4, the HPC events that can provide a global overview of Zybo, and also lists the HPC events common to all the above CPU cores. The common HPC events we found are eight, so Table 5.1 presents eight HPCs for each

selection method.

Method	HPC1	HPC2	HPC3	HPC4	HPC5	HPC6	HPC7	HPC8
Mutual Information	Instruction cache miss	L1 ITLB Miss	L1 Data Cache miss	Last Level Cache miss	ITLB allocation	DTLB allocation	Exception taken	ISB
Global view of the system (Zybo Z7-20)	Instruction cache miss	L1 Data Cache miss or Last Level Cache miss	Data TLB allocation	Exception taken	Branch mispredicted	Instructions executed	L1 Data Cache access or Last Level Cache access	LS/ST instructions
Events implemented in multiple architectures	Instruction cache miss	L1 Data Cache miss or Last Level Cache miss	Predictable branches	Exception taken	Branch mispredicted	Instructions executed	L1 Data Cache access or Last Level Cache access	Exception return

Table 5.1: HPC event selection.

As we can see from Table 5.1, HPC events that give us a global view of the behavior of the different hardware components of the CPU and HPC events implemented in different CPUs share HPCs. In addition, the listed HPC events have been used in previous SOTA work, as we presented in Table 3.1. Since a partial list of these HPC events provides a global overview of system behavior and they are implemented in several CPUs available on the market, and has already been used to detect attacks, we selected them for the implementation of the local mechanism. This allows us to propose a mechanism adapted to more CPUs but less accurate. This is because, these HPC events are not among the most prominent as determined by feature extraction methods, the local ML might be less accurate than using the best HPCs. However, the remote ML can help detect abnormal behavior by extracting the complex features from this HPC subset as well. Table 5.1 verifies that the HPC events selected by the feature extraction methods focus on specific components of the CPU.

Finally, the hardware implementation could reduce the energy consumption of the local detection mechanism because the dedicated hardware is more efficient than the general purpose CPU as discussed in [165]. In the following, we present the details of the hardware implementation.

Hardware Implementation

To implement the Local-Remote detection mechanism based on a hardware implementation of the local detection mechanism, we use the Zybo Z7-20 evaluation board. Zybo Z7-20 is an embedded software and digital circuit development board based on the Xilinx Zynq-7000 family. It contains a dual-core ARM Cortex-A9 processor running at 667MHz. The ARM Cortex-A9 processor is a 32-bit processor core that implements the ARMv7-A architecture. Zybo Z7-20 have a Processing System (PS)-Programmable Logic (PL) interface that allows us to implement the hardware monitor in PL and run

the applications in the PS. In addition, the ARM Coresight PMU component can be accessed via an AXI bus from the PL. In Figure 5.5, we introduce the Zybo architecture and the AXI bus that we can use to extract the HPC measurements. The red arrow shows the interface we can use to access the PMU from the programmable logic. The access can be done from the ARM Debug Access Port (DAP), and using an AXI master port from the PL to specify the address of the HPC registers to read.

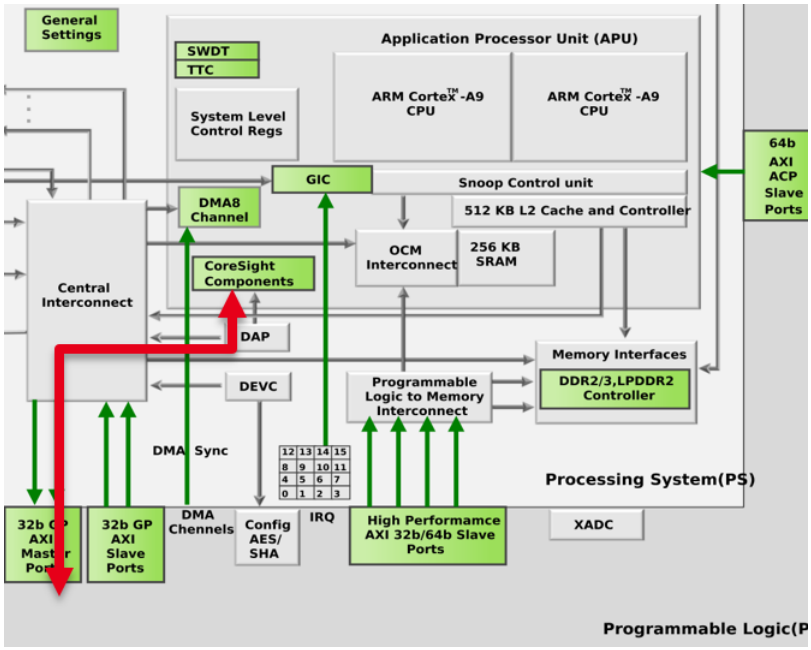


Figure 5.5: Zybo-Z7-20 architecture and the AXI bus allowing us to extract HPC values to the PL.

Monitoring Interval and Noise Reduction

As we explained earlier, monitor and applications share resources. The monitor runs with higher privileges than the other applications, which means that after the idle period (Algorithm 1 in Section 3.5.1) finishes (*sleep*(*monitoring interval*)), an interrupt is triggered and the OS schedules the monitor to run. Since this depends on the interrupt service, the OS saves the execution state of the previous application, etc, the time we let an application execute to collect HPC events varies. For example, if we use a monitoring interval of 2 ms, we get the following distribution, shown in red in Figure 5.6. As we can see from the figure, the monitoring interval is not exactly 2 ms, but has a mean value of 2.1 ms. The mean value of the monitoring interval is larger than 2 ms. This is because CPU needs some time to properly set up the execution of the monitor after the specified monitoring interval has passed. This extra time introduces additional noise in our measurements due to context switching. On the other hand, we see in the blue distribution that the hardware implementation extracts measurements without the un-

certainty shown in the software version. Nevertheless, we observe a small increase over the specified monitoring interval (5-10 clock cycles), but this is due to the AXI bus requests. The improvement achieved by the hardware implementation results from the fact that the hardware version waits from the specified clock periods of the FPGA and then immediately extracts the HPC measurements. This reduction in the noise of the HPC measurements can further improve the extracted data and help us better distinguish between attacks and normal operation.

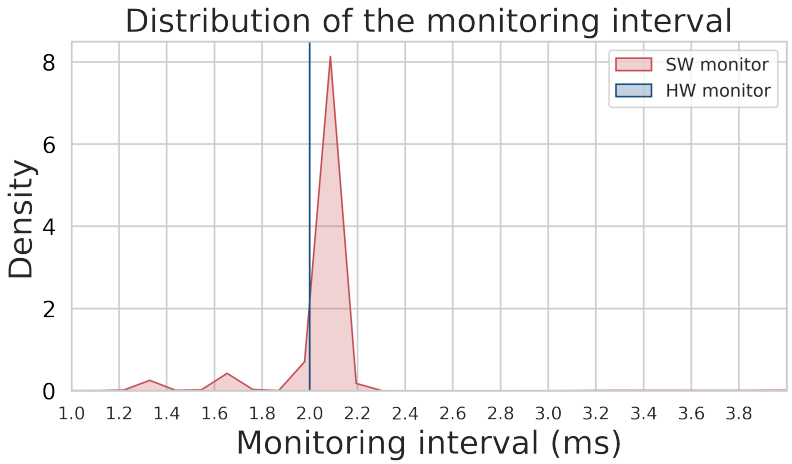


Figure 5.6: Monitor interval distribution of the SW and HW implementation of the detection module.

With the hardware implementation described above, we can minimize the shared resources between the applications and the monitor. We must mention that in this experimentation we use a 2ms monitoring interval instead of the 1ms used in Chapter 3, as this is a limitation of the current evaluation platform. Since this is a limitation of the current test platform, we do not consider the vulnerabilities we tried to reduce due to evasive malware by using a monitoring interval of 1ms. As explained in Section 3.4, reducing the monitoring interval to 1ms allowed us to decrease the time attackers have to add obfuscation techniques, and consequently their ability to bypass our monitoring. We make the hypothesis that in the final implementation, this design limitation should be considered.

HW implementation of the local detection mechanism

In the next paragraphs we will explain the basic blocks of the hardware implementation i.e., the ML algorithm, false positive filtering, and the FIFO, to store the necessary past samples in case we need them to further evaluate a suspicious sample remotely.

Machine Learning model The ML is implemented using a fixed-point representation of the model parameters. The HPC values are extracted as 32-bit registers from the

PMU of the ARM core, scaled, and also converted to a fixed-point representation. The implementation of ML was designed to maximize resource reuse and reduce energy consumption. The ML model is implemented with a pipeline that uses an adder and a multiplier. From Figure 5.7 we can see the block diagram of the ML model.

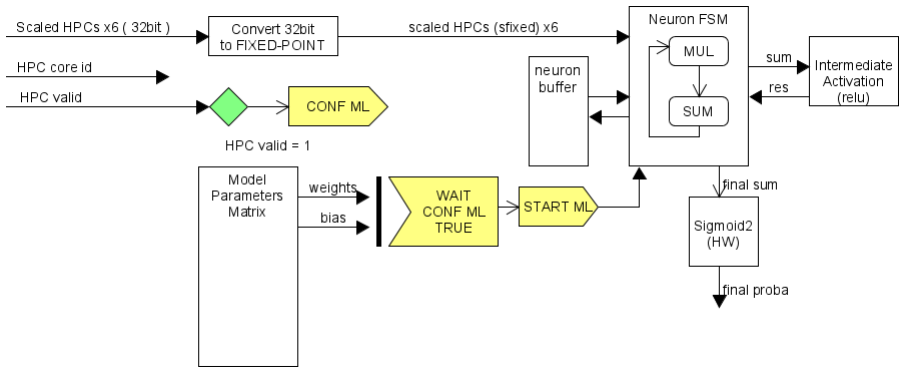


Figure 5.7: Block diagram of the HW implementation of the ML algorithm.

The ML is waiting for the HPC extraction module to signal that the measurements are valid i.e., HPC_VALID. The next step is to configure the ML with the neuron weights and bias. As mentioned in Section 2.4.1, the logistic regression model can be viewed as a single-layer neural network with one neuron. Since the neuron is the basic module for implementing more complex deep neural networks, instead of instantiating multiple neurons, we reuse the same neuron, which effectively reduces resources but increases the latency. The outputs of the neurons are stored in a buffer until they are needed as inputs for the next layer. Since each neuron has different parameters, the model parameter matrix stores the individual parameters and configures the neuron with the appropriate parameters as needed.

The activation functions are the nonlinear part of the model. For the ML model, the sigmoid is the function that converts the output of the neuron into a probability. The sigmoid is indeed one of the most important functions in our design. But due to the complexity of the sigmoid representation, special attention should be paid when implementing it in hardware. For example, the sigmoid function as represented in Equation (5.1) requires the computation of an exponent and a division. These functions require many resources to be implemented in hardware, and therefore many researchers propose a hardware implementation to approximate it. For our implementation, we reused the proposed sigmoids from [166], [167]. We denote the Equation (5.2) as *sigmoid1*, Equation (5.3) as *sigmoid2*, and Equation (5.4) as *sigmoid3*.

$$\hat{y} = \frac{1}{1 + e^{-(w^T * x + b)}} \tag{5.1}$$

$$f(x) = \begin{cases} 1, & 5 \leq \text{x} \\ 0.03125x + 0.84375, & 2.375 \leq \text{x} < 5 \\ 0.125x + 0.625, & 1 \leq \text{x} < 2.375 \\ 0.25x + 0.5, & 0 \leq \text{x} < 1 \\ 1 - f(-x), & \text{x} < 0 \end{cases} \tag{5.2}$$

$$f(x) = \begin{cases} 1, & 4 \leq \text{x} \\ -0.03577x^2 + 0.25908x + 0.5038, & 0 < \text{x} < 4 \\ 1 - f(-x), & \text{x} < 0 \end{cases} \tag{5.3}$$

$$f(x) = \begin{cases} 0, & \text{x} \leq 1 \\ 0.00247x + 0.01843, & \text{-8} \leq \text{x} < \text{-4.5} \\ 0.02415x + 0.1159, & \text{-4.5} \leq \text{x} < \text{-3} \\ 0.0639x + 0.23525, & \text{-3} \leq \text{x} < \text{-2.5} \\ 0.0831x + 0.28325, & \text{-2.5} \leq \text{x} < \text{-2} \\ 0.12891x + 0.37487, & \text{-2} \leq \text{x} < \text{-1.5} \\ 0.16351x + 0.42677, & \text{-1.5} \leq \text{x} < \text{-1} \\ 0.23674x + 0.5, & \text{-1} \leq \text{x} < 1 \\ 0.16351x + 0.57323, & 1 \leq \text{x} < 1.5 \\ 0.12891x + 0.62513, & 1.5 \leq \text{x} < 2 \\ 0.0831x + 0.71675, & 2 \leq \text{x} < 2.5 \\ 0.0639x + 0.76475, & 2.5 \leq \text{x} < 3 \\ 0.02415x + 0.88401, & 3 \leq \text{x} < 4.5 \\ 0.00247x + 0.98157, & 4.5 \leq \text{x} < 8 \\ 1, & 8 \leq \text{x} \end{cases} \tag{5.4}$$

All of the above sigmoid functions attempt to approximate the sigmoid by segmentation. As the sigmoid is approximated, there is an error between the hardware and software versions, which is added to the loss of precision due to the fixed-point representation. Figure 5.8 shows the errors of the three selected sigmoid functions. As we can see from the figure, the error of *sigmoid3* is less than 0.006 (0.6%), while *sigmoid1* and *sigmoid2* have an error of 0.018 (1.8%). As we can see from the figure, the errors have some peaks whose position will affect the final implementation, as we will present in Section 5.4.

Decision making and FP filtering The next step is decision making and FP filtering, as you can see in Figure 5.9. After we get the probability from the ML model (final proba), we compare the probability with the two thresholds, i.e., suspicious and alert, and we output two decisions. Since we might have some FPs, as explained in Chapter 3, the FP filtering is applied. The FP filtering algorithm (EWMA) is applied to both the

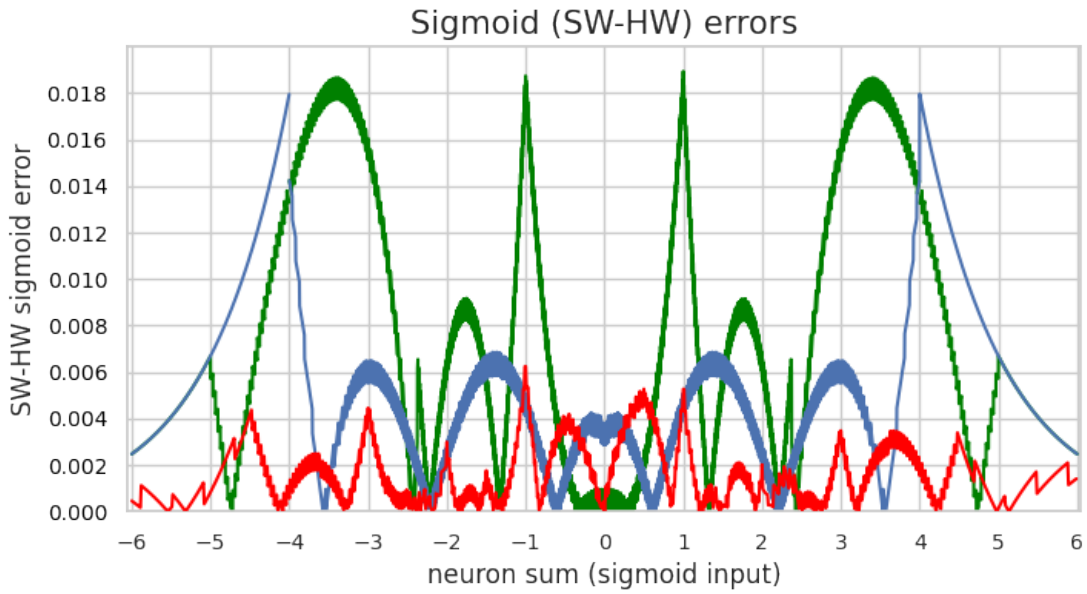


Figure 5.8: Errors introduced due to the implementation of the sigmoid function in hardware.

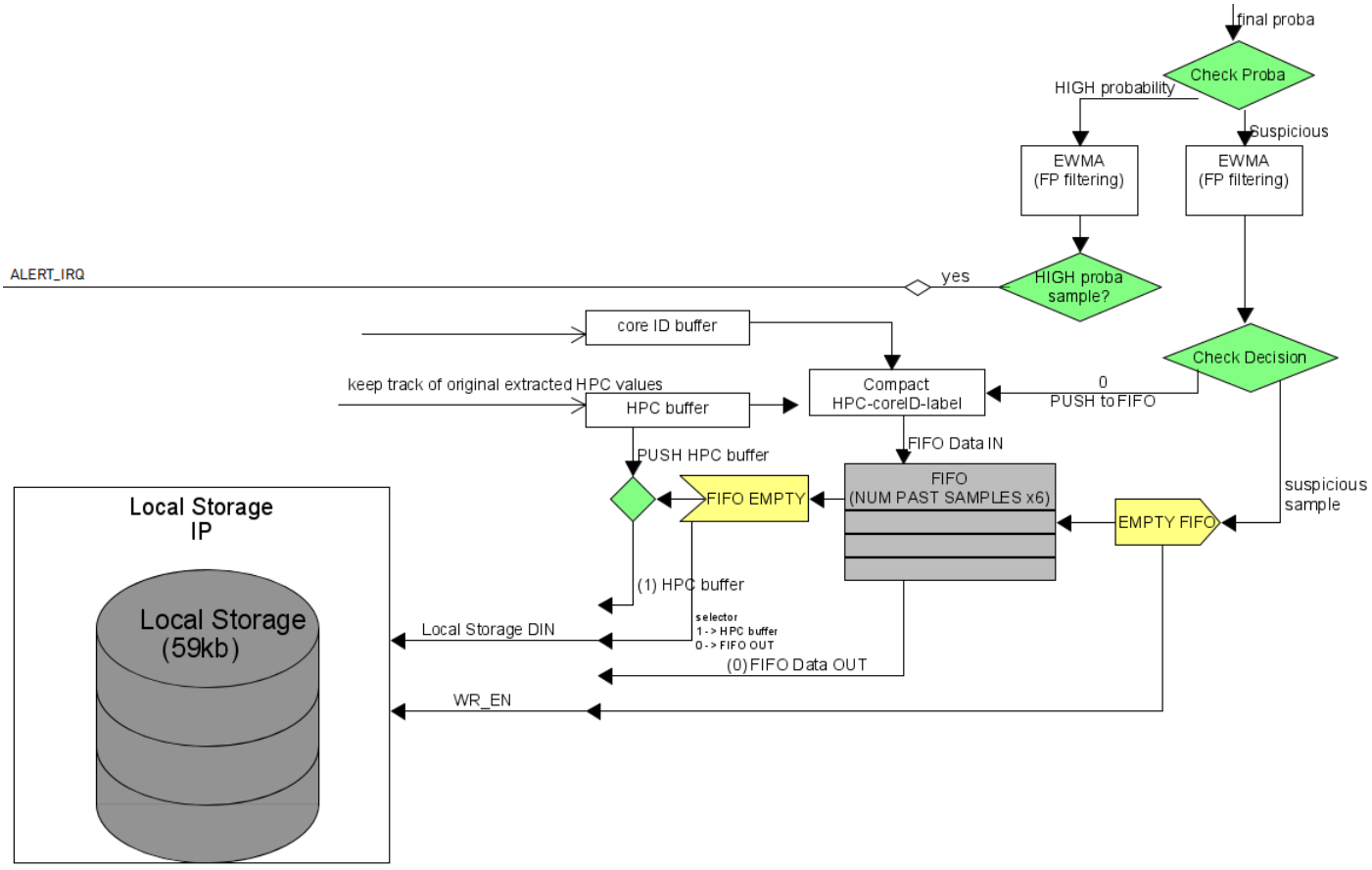


Figure 5.9: Decision making and FP filtering block diagram.

alert and the suspicious decisions. If an alert sample is detected, an Interrupt Request (IRQ) is sent to the secure processor for appropriate action (ALERT IRQ). Otherwise, if the sample is classified as normal, the sample is pushed in the FIFO. In Figure 5.9 we can see that before pushing the sample to the FIFO, we compact the samples with the ID of the core we extracted the sample and the label. The core ID and the label are added at the highest bits of the HPC_0 register. This is because we push the original 32 bit extracted values, and since at 1ms none of the HPCs reach a close to maximum values, the higher bits are always zero so we could use them to reduce the require space.

If the sample is classified as suspicious, a request to empty the FIFO is sent. The output from FIFO is sent to local storage where the samples are stored until the secure core sends them to the remote. When the FIFO is empty, the current suspicious sample is also pushed to the local storage.

When the local storage reaches its maximum capacity, a MEM_FULL IRQ is sent to the secure core to immediately read the stored samples and send them to the remote. Recall that the size of the local storage is fixed at the maximum number of samples we need to store during a Δs period in normal operation. This means that during normal operation we store less data in local storage than the maximum capacity. The local storage will only emit a MEM_FULL signal when malware is executed, because as we show in Section 4.2.2, the malware overflow the local storage before the expected Δs.

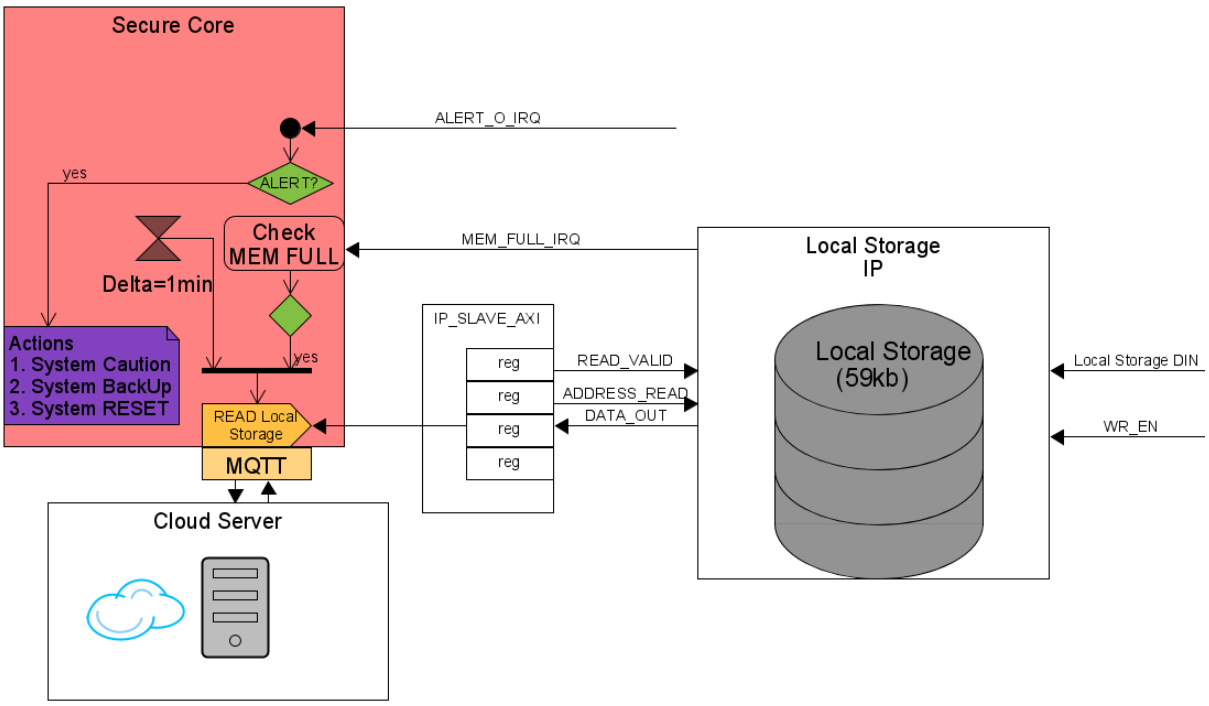


Figure 5.10: Data transmission and/or action by the secure core

Data transmission and/or action Then a secure core reads the data stored in local storage and sends it to the remote system for further analysis. This happens either every Δs, or when an interrupt is received due to the MEM_FULL IRQ. In the event of an ALERT IRQ, appropriate action may be taken, such as rolling back to a safe state, restricting the execution of sensitive information, or resetting the system. This can be viewed in Figure 5.10. The severity of the action can be determined, for example, by the frequency of alert interruptions. The protocol to be used for data transmission is MQTT. MQTT is an OASIS standard messaging protocol for the IoT, and is designed as a lightweight publish/subscribe messaging transport that is ideal for connecting remote devices with a small code footprint and minimal network bandwidth [168]. The secure core can be either an isolated physical CPU core, or a secure element as proposed in the iMRC project [131].

Figure 5.11 shows the whole block diagram of the local-remote in hardware. From the figure we can see how all of the components are connected with each other.

5.4 Hardware Local ML Evaluation

In this section, we evaluate the hardware implementation of the ML model. As mentioned earlier, the switch to hardware is accompanied by errors due to the lower precision of the number representation and the approximation of various functions. These errors can affect the accuracy of our model and its expected behavior. Furthermore, since the sigmoid functions convert the final output of the neurons into probabilities, we investigate how their errors affect the decision making for suspicious and alert samples.

5.4.1 Hardware and Software Logistic Regression Metrics

The first step is to see how each of the software and hardware versions of logistic regression performs on this classification problem. We trained the model ML offline, which gives us the model parameters and the thresholds for configuring the implementation of HW. The *suspicious threshold* was set to 0.4 (40%) and the *alert* threshold to 0.9 (90%). We must mention here that we try to train the model using quantized parameters i.e., representing the parameters during training with the fixed-point representation instead of floating-point. The training was done using QKeras [169]. QKeras is a library to train ML models using quantized representation of the model parameters, in contrast to Keras which only allows the training using floating-point numbers. During our experimentation for the current problem we observed that using the model trained using

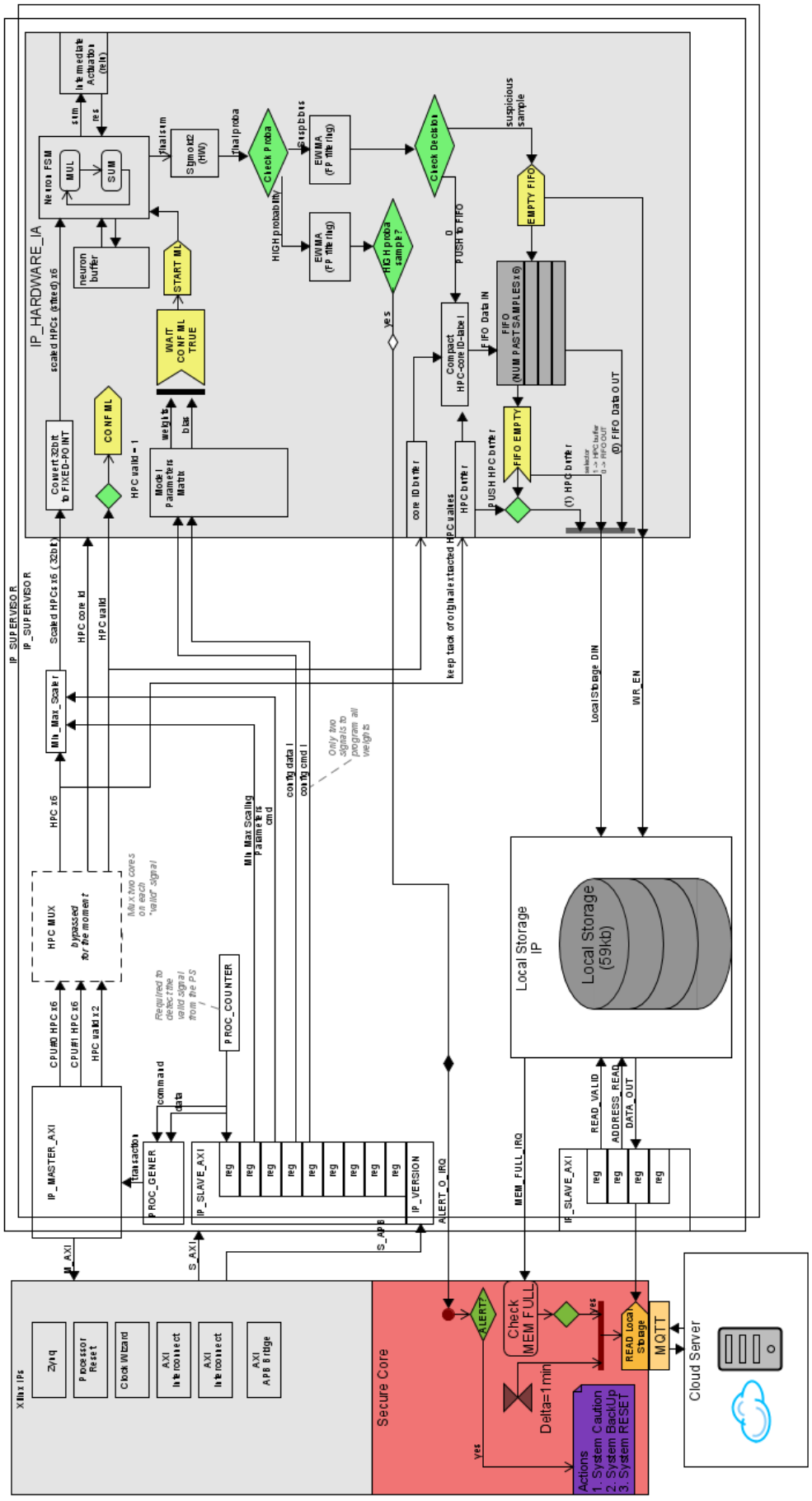
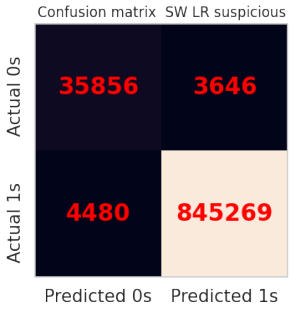


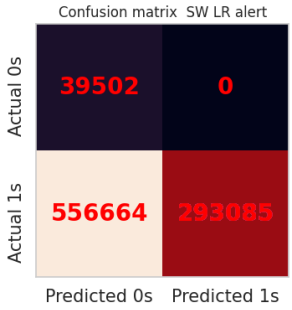
Figure 5.11: Block diagram of the HW implementation of a Logistic Regression in a local-remote design.

floating points and after converting the parameters as fixed-points gave us better results than directly using the quantized model. In this test we use a logistic regression with limited parameters, but for models with more parameters the quantized model using the fixed-point representation might provide better results that converting the floating-point parameters to fixed-point.

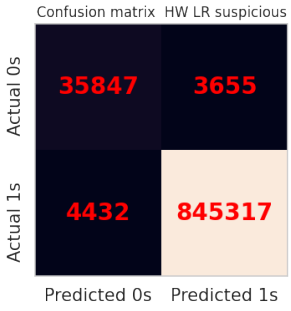
Also, due to the time complexity of evaluating the model directly in hardware, we simulate its behavior in software using Python and the same fixed-point number representation and model implementation i.e., the neuron pipeline and the hardware approximation of the sigmoids. Each step of the software simulation follows the design guidelines of the hardware. To verify the correctness of the software simulation, we verify the Python results and the results obtained by the Vivado simulator are the same. For the fixed-point representation of the inputs and model parameters, we use the Fxpmath library [170]. With Fxpmath we can represent a real number as Fxp(number, sign=True, total number of bits, fraction number of bits).



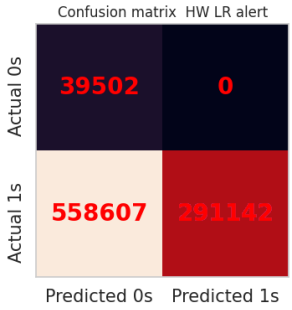
(a) Confusion Matrix for suspicious data (SW LR).



(b) Confusion Matrix for alert data (SW LR).



(c) Confusion Matrix for suspicious data (HW LR).



(d) Confusion Matrix for alert data (HW LR).

Figure 5.12: Confusion Matrices for the alert and suspicious data of the SW and HW version of the Logistic Regression model implemented locally.

In Figure 5.12 we see the confusion matrices for the suspicious and alert data of the HW and SW implementation. Out of the total 889251 samples, we can see that the HW implementation differs from the SW version in 2000 samples i.e., 57 suspicious labelled

samples and 1943 alert samples. Of greatest importance is any discrepancy in the alert data, as these are the samples for which the local ML takes direct action and which can lead to potential false alarms. Using the confusion matrices of Figure 5.12b and Figure 5.12d, it can be seen that the local system does not produce false positives in either case. This means that the 1943 samples that differ are samples that have a high probability in the SW version, but the HW local ML stores them for further evaluation instead of flagging them as alert samples. Since this classification does not result in FPs and we can still evaluate them remotely, we only lose in detection time.

On the other hand, we can see in Figure 5.12a and Figure 5.12c that the 57 samples that differ in the SW and HW implementation lead to better results for the HW implementation. This is because the HW implementation correctly flags 48 more positive samples as suspicious and 9 more normal samples as negative. This means that the HW implementation allows us to evaluate correctly more samples to the remote while successfully filtering more normal samples.

5.4.2 Hardware and Software Sigmoid Evaluation

After the previous example, it is of utmost interest to further investigate how the errors of HW may affect the behavior of local MLs. Using the equations Equation (5.2), Equation (5.3), and Equation (5.4), we can see that researchers are trying to split the software function into several linear parts or a second order equation. In selecting the sigmoids, we considered the area overhead in addition to the sw-hw error. *Sigmoid1* and *sigmoid3* have the same built-in feature of approximating the software sigmoid with multiple linear segments. *Sigmoid3* splits the function into more segments, reducing the error as we saw in Figure 5.8 to less than 0.005 (0.5%) compared to *sigmoid1*, which has an error of less than 0.018 (1.8%). This is accompanied by an increased resource overhead. Using our Vivado implementation, we can see that *sigmoid3* consumes 24% more resources than *sigmoid1*. Since area overhead is an important factor when implementing a hardware mechanism, both area and error are considered.

The local-remote proposal is sensitive around the chosen thresholds i.e., *suspicious* and *alert*. Using Figure 5.8, we can see that each sigmoid has higher errors in different segments of its input. For example, *sigmoid1* has peaks around [-4.5, -2.7], [-1.5, -0.5], [2.7, 4.5], and [0.5, 1.5], while it has low error in the other input segments. *Sigmoid2* has a small error between [-3.8, 3.8], but exceeds 0.01(1%) outside this range. Therefore, we investigate whether and how these peaks affect the classification results. For example, if a threshold is near the high error peaks, this could lead to multiple misclassifications compared to the software part. However, if a threshold is near a segment with low er-

rors, the sigmoid errors might not affect at all the hardware implementation at all.

In the next examples, we demonstrate how the sigmoid can hypothetically affect our classification results. In Figure 5.13, we can observe the hypothetical case where our algorithm returns as thresholds 0.4 for suspicious and 0.99 for alert. The red shaded area represents the range of input values for the sigmoid that are misclassified in hardware compared to software. This means that if our classification threshold is 0.99, for example, and an input is assigned a probability of 0.991 in software, it could be assigned 0.989 in hardware, resulting in a FP. Though, the red shaded areas represent the sigmoid input range that are prone to misclassification due to the hardware error. As we can see from the figure, the susceptible area for the suspicious threshold is very small, unlike the area around the 0.99 threshold. As we can see, the neuron outputs in the range [3.9, 5.4] can be misclassified compared to the software. For malicious samples, the undesirable effect is that samples with a higher probability than 0.9 in the software but map to less than the threshold in the hardware, effectively missing the malicious behavior. For normal samples, the undesirable effect is that samples that have a probability of less than 0.99 in software but are mapped higher due to the hardware error, resulting in potential false alarms.

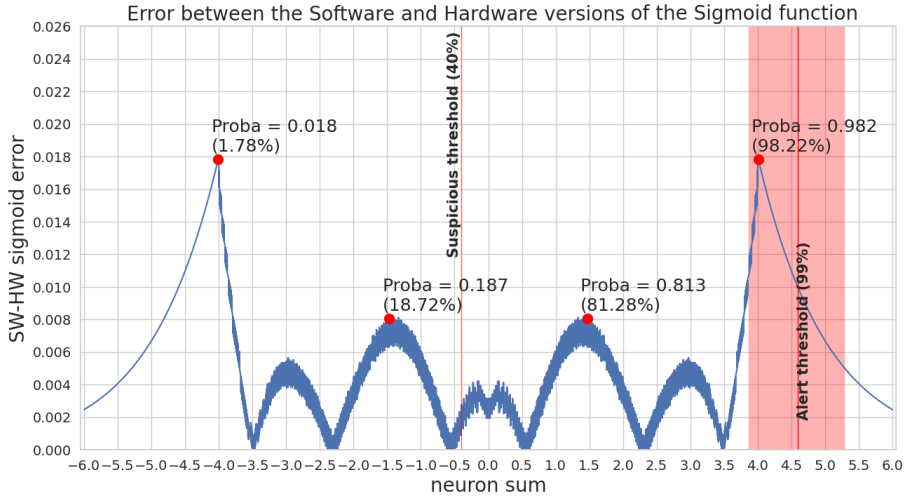


Figure 5.13: Sigmoid2 error and hypothetical area of misclassifications if alert threshold was 0.99(99%).

To test the above hypothetical reasoning in practice, we perform the classification for the software and hardware models using the alert threshold of 0.99(99%). Figure 5.14 shows the software and hardware version confusion matrices for the alert data. In this case, we have 129426 samples out of a total of 889251 samples that differ between the software ML and the hardware ML. But as we can see from the confusion matrix in Figure 5.14b, these misclassifications resulted in 129426 TPs, i.e., samples that were not correctly identified by the software version. In addition, we do not observe FP. The

above example shows that when the error of the hardware is high compared to the software version, differences in the decisions of the ML model in SW and HW apply. In the present case, the hardware performs better, but in other cases we might risk multiple false alarms. Special care should be taken when considering both the alert and suspicious thresholds, as our model may not behave as expected.

(a) Confusion Matrix for alert data (SW LR). (b) Confusion Matrix for alert data (HW LR).

Figure 5.14: Confusion Matrices for the alert data of the SW and HW version of the Logistic Regression model implemented locally when the *alert threshold* is 0.99(99%).

Which activation function is most proper for the local-remote Since, as we find in our experiments, the errors of the neurons are minimal, the major source of error is in the sigmoid activation functions. For the local-remote, we should generally use the sigmoid that has the lowest errors around the two thresholds *suspicious* and *alert*. This will allow us to be more certain that the expected software and the implemented hardware mechanism are close. We also need to consider the distribution of the training data as inputs to the sigmoid. That is, when more data are mapped around the high error points of the sigmoid, the probability of misclassification is the highest.

Figure 5.15 represents the errors distribution of *sigmoid3* and the distribution of normal and malware training data as inputs to the sigmoid. In an earlier example presented in Figure 5.12, we show that for the *alert* threshold of 0.9, we have 1943 differences between the hardware and software implementations, with most of the differences resulting in the alert data being classified as suspicious in hardware. Although sigmoid3 has a minimum error of less than 0.001(0.1%) around the 0.9 threshold, from Figure 5.15, we can see from the distribution of samples (red shaded area) that many samples are assigned around the threshold. This increases the probability that the samples are misclassified. On the other hand, in the figure we see the red bullet point at input 3.7, which corresponds to a probability of 0.9761. At this point, the error of the sigmoid is higher than at 0.9, but there are fewer samples around this point, resulting in only 1383 differences between software and hardware. The above it shows that despite the high errors of the

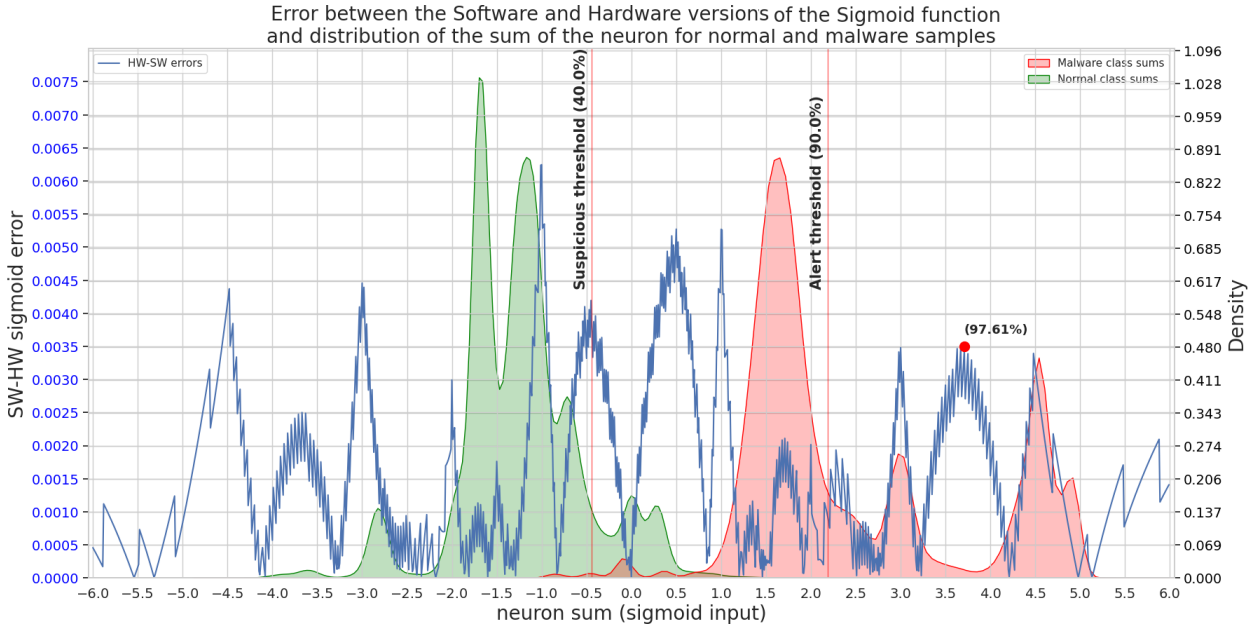


Figure 5.15: Sigmoid3 error and distribution of normal and malware data inputs to the sigmoid.

sigmoids, it is important to consider the errors only in the areas that interest us i.e., around the alert and suspicious thresholds. Further, we also need to verify that our algorithm does not map the input samples to probabilities around high peak errors us this might increase the misclassified samples.

Considering the area overhead, the errors caused by the hardware, and the current thresholds, we chose *sigmoid2* for our implementation. Using sigmoid2, we observe 1119 differences compared to the software version ML, while 1075 alert decisions are classified as suspicious data by the hardware ML and 44 normal samples are classified as suspicious by the software versions, while they are marked as normal by the hardware.

Classification metrics for the local-remote We implement the ML model using sigmoid2 and evaluated the performance of the overall design. As mentioned earlier, due to the sigmoid implementation in hardware, some alert samples were not detected locally but were classified as suspicious to be further evaluated in the remote ML. The remote ML in this case is a LSTM, which was trained with normal data only. The remote ML uses the seven past samples to predict the current sample, as explained in Chapter 4. The classification metrics for this implementation are as follows in Table 5.2:

In the Table 5.2 we can see the classification metrics for using only the local ML and for the local-remote implementation. In this case, when only the local ML is used, we chose a classification threshold of 0.5 (50%), and for the local-remote implementation, the suspicious and alert thresholds were 0.4 (40%) and 0.9 (90%). The most important

Local + Remote	F-score	Sensitivity	Precision	FPR	Accuracy	% of Positives detected locally	% of Positives detected remotely
Logistic Regression ((***only local***)) [*threshold 50%*]	96.22%	92.76% (16/16)	99.95%	0.98% (15/20)	93.04%	92.80%	N/A
Logistic Regression [*thresholds 40%, 90%*] ***+ LSTM***	98.96%	97.94% (16/16)	99.99%	0.13% (1/20)	98.02%	59.45% (12/16)	38.49% (4/16)

Table 5.2: Classification metrics for the only Local (HW) and Local (HW) + Remote ML when local detection is activated and when not.

metrics to observe is the sensitivity and FPR. Both columns, except for the score, indicate how many attacks were correctly detected and how many normal applications were classified as false alarms. Using only the local ML we see that we can detect 16 out of 16 attack vectors, but on the other hand, we trigger false alarms in 15 out of 20 normal applications. This again demonstrates our argument that simple ML might be able to effectively identify malicious behavior, but at the cost of multiple FPs. On the other hand, using a local-remote implementation, we can see that we detect all 16 attacks, while raising a false alarm on only one application. The table also shows that the local-remote system triggered a quick alarm for 12 attacks, while it required the verdict of the remote system for the remaining 4 attacks. These results show that the proposed idea can help to effectively detect attack vectors while minimizing FPR.

Hardware Overhead We have implemented the above local-remote design in Vivado and give the estimated resources used by the implementation as well as the energy consumption.

IP	BRAM_18K	DSP48E	FF	LUT	Energy overhead
Custom Logistic Regression + FP filtering + Local Storage	20 (14.28%)	10 (4.54%)	1381 (1.3%)	1140 (2.14%)	5%

Table 5.3: Resources used by the custom Logistic Regression IP, FP filtering and local storage.

We consider only the local ML model, FP filtering and the local storage, since these are the components we added. The AXI bus is a component that must be instantiated by other solutions to connect the PS and the PL. Table 5.3 presents the hardware re-

sources used by the design. The design consumes 2.14% of the available LUTs, 1.3% of the available Flip-Flops (FF), 14.28% of the available block RAM, and 4.54% of the available DSPs. While most resources remain below 5%, which we consider acceptable for this platform, the memory overhead of the block RAM is 14.28%. This is due to the local-storage that has a size of 59kb. Nevertheless, we consider this a significant improvement considering that in other cases we need to store 1406kb of data since we monitor two cores and extract measurements every 2ms. This is the case when we need to send all the extracted data in a period of Δs to the remote system for making a decision as in [88], [97]. This means that with the proposed solution we only need to store a maximum of 4.2% of the extracted data. Further, a software implementation of the local ML also requires the same memory to store the samples before transmitting them to the remote. Last, we observed that the estimated energy consumed by the whole board is 1.836W, while the FPGA consumes 11% of this amount. From this 11%, 6% is the clock generator, which means the the estimated energy overhead of the AXI interface and the hardware ML is 5% of the estimated consumed energy. We must note that the energy comparison between software and hardware was not evaluated.

More information about the resources used and the design can be found in Section A.2.

5.5 Conclusion

In this section, we presented how the local-remote implementation could be implemented in hardware. This section also analyzed the reasons that led us to consider such an implementation. Apart from the reasons for the performance overhead on the local CPU due to the shared resources between the local ML mechanism and the rest of the applications, we presented how the noise in the measurements can be improved if the HPCs are extracted from the hardware. Due to the OS interrupt service routine, the extraction does not occur exactly in the specified time interval, but exhibits noise. On the other hand, the hardware directly accesses the PMU to extract the HPCs, resulting in an almost precise monitoring interval.

The main reason that made us do this is that a software implementation is also vulnerable to software attacks. Apart from OS and its numerous discovered bugs, we have shown how Rowhammer can corrupt some model parameters to make our classifier ineffective. We also developed a simple malware that can disrupt HPCs and presented four attack scenarios. Since these reasons could allow attackers to bypass our mechanism, we proposed an implementation in hardware. Accessing the hardware is more difficult for an attacker, which increases the level of security.

Lastly, we presented the implementation and the errors due to the migration to hardware. We analyzed and evaluated how they can affect the local-remote implementation and its classification metrics. In the end, we find that the hardware overhead for most resources remains below 5%, but our design has a 14% overhead in block RAMs due to local storage of samples for further evaluation. We argue that the memory overhead also exist in the software version of the local ML, and this overhead is one of the main limitations of the proposed solution. Nevertheless, the proposed idea manages at least 95.2% data filtering in normal operation, which significantly reduces the data to be stored and eventually transferred.

6

Conclusion and Perspectives

In this chapter, we conclude this manuscript, the purpose of which was to describe our research on securing IoT devices from microarchitectural using low-level information. We provide a summary of our work and contributions. Below, we outline some limitations of the current work and some perspectives that can improve the presented work by the research community.

6.1 Summary of the contributions

The research contributions presented in this manuscript are the result of the growing interest in the security of systems. One of the most interesting research areas is the security of resource-constrained devices such as IoT and IIoT. Apart from the limited security measures implemented in these devices, their increasing complexity to meet the need for more resources has increased the attack surface. Attackers can now leverage another group of attacks in addition to traditional malware, which we refer to in this work as Software Attacks Targeting Hardware Vulnerabilities (SATHV). This group of attacks is particularly interesting because they can use side channels to extract system information and/or damage the system by inducing errors. In the State of the Art (SOTA), we find many solutions to protect against SATHV using information from Hardware Performance Counters (HPCs), but they consider server and desktop systems. While these solutions show promising results, they cannot be directly applied to systems with limited resources. These systems have many constraints that include computational and memory resources, communication bandwidth, and energy consumption. This drove our research on how the proposed solution can be applied in the context of IoT and IIoT.

Implementing efficient security solutions that have a high detection rate and a minimal False Positive Rate (FPR) while keeping implementation costs low is a challenge. Machine learning tools can be used to learn how to distinguish between malicious and normal operations, but their ability to efficiently learn how to distinguish depends directly on the input information. In this work, the input data is the extracted HPCs, and the number of inputs is limited due to the system specifications. Though, it is necessary to select the most relevant information sources to be used as inputs from a large number of available HPCs. Since the available HPCs are numerous, selecting the best one can be time consuming or require some expertise. In Chapter 3, we showed that theoretical information alone is not sufficient and an evaluation should be performed for each attack vector. The theoretical information is composed of HPCs already used in SOTA, but as we saw in Chapter 3, it depends on the targeted class of attacks. If the attack vectors considered are limited, the HPCs used are restricted to some hardware components, which can be an outfall if other attacks do not stress the monitored components. We have also shown that eviction-based attack vectors exhibit different behavior than flush-based vectors. Since SATHV can be performed using either technique, considering only flush-based vectors could allow attackers to bypass the system with eviction. This was demonstrated in Section 3.3 and a work published in NEWCAS 2021 [27].

We also discussed how attackers can circumvent the system using evasion techniques and how monitoring frequency can allow us to reduce the attacker's ability to success-

fully bypass our protections and still succeed. In addition, many works do not take into account execution environments where multiple processes are running in parallel. This could allow attackers to hide malicious activity among legitimate executions, potentially avoiding detection due to FP filtering. In addition, context switching introduces noise to traditional HPCs selected for SATHV detection, such as the cache miss ratio, since during context switching this side-effect increases due to the need to load the new application. This could lead to many FPs. Considering the possibility that attackers can bypass detection using evasive techniques, hide between legitimate applications, or use eviction techniques instead of flush-based vectors, and considering a noisy environment where multiple applications share resources, we propose a security mechanism for SATHV detection in ARMv7 systems in Section 3.5. This work was published in DSD 2021 [28].

Given this, and the fact that eviction and flush based SATHV attack vectors may have different behaviors, as well as the different behaviors and increasing complexity of modern applications, we propose a detection mechanism in Chapter 4 that takes into account the limitations of the IoT. As mentioned earlier, the computational and memory resources of the IoT are limited, as is the communication bandwidth. We observed, that our simple linear ML classifier proposed in Chapter 3 and SOTA is unable to effectively detect all attacks while minimizing the FPR. FPR is as important as detection rate in IoT devices. This is because actions taken due to false alarms can lead to increased overhead costs or, in the worst case, render the device unusable. Though simple ML systems cannot efficiently secure the system. SOTA works [76], [92] recognize this limitation of simpler ML systems and propose more complex ML models that increase their detection rate while keeping FPR low. But these solutions come at the price of adding system overheads. Though, the SOTA works [88], [98] propose detection in a remote system where the resources are "unlimited", so that the devices only need to transmit the extracted data. We also chose to monitor the system at a high frequency (1ms monitoring interval), which increases the data extracted from the system. Considering the amount of extracted data and the hundreds of IoT devices connected to the Internet, this solution may pose a problem for the networks' ability to handle all the traffic. For this reason, we have proposed a solution that uses many edge computing techniques in the context of security. Our main idea is to filter the extracted data and transmit only the malicious patterns. Also, by setting two confidence levels (alert and suspicious), we can detect malicious operations locally when we are confident that an attack is being carried out. When the mechanism is less confident, the extracted samples are stored for later transmission to a remote complex mechanism for further evaluation. In this way, we were able to reduce the communication overhead by filtering 99.32% of the extracted

data in normal operation, successfully detecting the attacks, and minimizing the FPR to near zero percent.

Finally, we argue in Chapter 5 that the software implementation of Chapter 4 is not secure against software attacks. We demonstrated four test case scenarios that an attacker could use to disrupt the extracted measurements, effectively bypassing the mechanism. We also simulated a Rowhammer error on a parameter of the last neuron of a neural network, rendering our mechanism inefficient. These test cases, as well as the numerous software bugs in the code of OS, can compromise a software-implemented mechanism. This led us to implement our local-remote idea in hardware, where access rights are limited and we can also reduce system overhead since applications and monitor do not share resources. We also evaluated the hardware implementation and outlined how migrating from software to hardware can affect our work due to errors in number representation and function approximations.

6.2 Limitations and tracks for improvements

In this section, we discuss the limitations we found in our work and we give some perspectives for improvement.

Evasive strategies In Chapter 3 we presented some methods on how to implement evasive malware. The techniques used were originally proposed in [138] for Spectre attacks, which we use to implement evasive SATHV for each class such as CacheCSA, Meltdown, and Spectre. These techniques include inserting NOP instructions, *sleep*(), or normal operations at specific times within the malware's execution, with the goal of modifying each behavior closer to normal. We have found that monitoring the system with high frequency decreases the success rate of developing successful and evasive SATHV. This is because the inserted code introduces noise into the malware. When these techniques are used more frequently, the malware fails to extract the secret or cause an error. For example, a potential cause for this in CacheCSA is that when putting the attack for *sleep*(), the OS will most probably schedule another application to execute which can affect the cache state.

In this work, we chose a monitoring interval of 1 ms. The monitoring interval of 1 ms was chosen because it allows us to monitor the system quickly, reducing the time an attacker has to create evasive malware and limiting the overhead caused by monitoring. However, as more systems are equipped with higher-frequency CPU cores, the more time the attackers have to create evasive malware because the faster CPU cores can

perform the sensitive tasks faster, giving them more time to create evasive malware. This could be avoided by monitoring the system at a higher granularity (i.e., 500us). However, this impacts performance overhead due to more frequent context switches between the monitor-ML and applications, and further the extracted data increases linearly. Though attackers could exploit this limitation to successfully bypass the security solution in systems with higher operating frequency.

Another evasion technique that could be used by the attackers is the use of a compiler. Modern compilers are very complex and can produce code with many specifications. Moreover, our model is based on machine learning techniques that take input and output their decisions. If attackers have knowledge about the system, it is possible that they can be bypassed by specially crafted inputs [171], [172]. Moreover, in this work we have only considered simple ML models that have a limited number of parameters. While more complex ML models can learn more information, the simplicity of the models used in this work allows only limited learning capacity. This means that attackers need to expend less effort to specify a set of inputs that they can use to bypass the protection mechanisms. It is then possible for a compiler to be developed that takes these specifications into account to produce code that successfully evades the detection while successfully performing its malicious activities. SATHV-sensitive tasks rely on assembler code defined by the attacker, which complicates the above hypothesis, but the capabilities of modern compilers can still enable success of this strategy.

Data transmission In this work we consider the local mechanism responsible for sending the data to the remote system using a secure element, such the one proposed in [131]. This secure element reads the locally stored data and transmits it to the remote while receiving the alert interrupt to perform appropriate actions. This establishes a kind of secure connection between the edge device and the server, However, if attackers successfully perform a DDoS attack and the network becomes unavailable, it is solely in the ability of the local ML, to detect possible threats. In our experiments, we observed that as the number of normal and malware applications used increases, it becomes more challenging to find an alert threshold that does not trigger false alarms. This applies that the device only filters normal data while storing suspicious data for further analysis. In the case of a DDoS attack, a missing alert threshold could compromise the device. However, in real-world scenarios, a certain number of FPs is allowed depending on the criticality of the application. In such a case, we could use a threshold that allows some FPs.

Threat model In chapter 5, we also identify a number of software attacks that can compromise the system, while we do not consider hardware attacks that require physical access. For this reason, we propose the use of a hardware implementation in an FPGA. FPGAs are increasingly used to implement ML because of the hardware acceleration they provide [173]. They can be easily reprogrammed so that we can update our local ML to keep up with new attack vectors. But an FPGA is less energy efficient than an Application-Specific Integrated Circuit (ASIC) [174]. Energy is an important factor when proposing an implementation for IoT devices since most of them are battery powered, but energy consumption has not been evaluated in this work. On the other hand, an ASIC is application specific, so it is more difficult to upgrade it's local ML configuration. Considering this, we proposed the implementation in an FPGA.

However, to program the FPGA, a bitstream file must be transferred to the target device. This can be a vulnerability, however, as the bitstream could be tampered with and hardware rootkits could be implemented in the FPGA. In [175], the authors outlined a number of scenarios to tamper with the AXI bus in a SoC FPGA implementation. Such a rootkit could also be used to corrupt the extracted HPC data since we also use the AXI. Though, to avoid this vulnerability, a secure connection and verification should be used.

These are some limitations of the hardware implementations known to the research and industrial community. Further, IoT are known to not receive the appropriate updates. Though, how to configure the hardware implementation securely was not researched in this work.

Summary In the previous paragraphs we listed some of the limitations we identified during our work and could potentially allow attackers to compromise the security of the device. Improving or considering them could allow the implementation of a more robust mechanism.

6.3 Long term perspectives

In this section we list some intuitions that can lead the research community to improve this work.

Transmission frequency, Δs A first intuition is the Δs, which was used to define the time interval in which the local system sends data to the remote system. In this work we have set it to one minute, but further evaluation needs to be done that takes into account

communication overhead, detection latency, memory overhead, and energy consumption. As we saw in Chapter 5, the IP that consumes the most resources in the FPGA is local storage. Faster transmission of data can reduce the overhead of local memory, but could also increase energy consumption due to frequent transmission. Less frequent transfer of data can increase both the local memory overhead as more data is stored and the decision latency that could allow the attackers to successfully perform their malicious activities in time. A balance between all factors should be explored to choose the best parameter.

Actions on a received alert A second intuition is the actions to take when an alert interruption is signaled. In this work, we have mainly considered a system reset. However, we have identified a number of actions that may be of greatest interest. The basic parameter to be studied is the frequency of alarms. If the frequency of alarms is very low, the system could avoid running sensitive content or move the suspicious application to an isolated environment. If an alarm occurs with a medium frequency, a rollback to a safe state could be performed. In addition, we could verify the integrity of the firmware to identify if new applications, services, kernel-modules have been installed. Finally, if the system receives alarms with a high frequency, only then a system reset might be required. The set of actions and how to apply them depending on the frequency of alerts and/or suspicious data being stored for further evaluation is of the most interest.

From SATHV to Malware Detection Until now, we have assumed that our attack library includes only SATHV. But IoT devices, as mentioned in Section 2.1, are vulnerable to other malware families. This poses a limitation of our current threat model.

Further increasing the threat model to include generic malware poses a great challenge for the current implementation. Since our current solution, which uses simple ML algorithms locally, may not be sufficient to detect attacks locally without raising many false alarms but may only be used for filtering, more complex ML solutions could be used that can recognize more behaviors. These complex ML solutions used locally can still be considered less complex and resource demanding than the more complex remote ML solution. Considering that hardware parallelization can help us efficiently implement these solutions in hardware, and the performance overhead on CPU is significantly reduced while resources are reduced through pipelining and reuse of resources.

In addition, since there are many malware families that all have different behaviors, we propose to use a local anomaly detection algorithm. The local ML algorithm is trained using only normal training data. This allows it to detect most normal behaviors as normal and classify deviations as suspicious. The local ML in this case is a CNN trained to

use seven past samples to predict the current sample. If the prediction (reconstruction) error is below an initial threshold (suspicious), the sample is classified as trustworthy. If the prediction error is higher than a second threshold (alert), an alarm is triggered. If the prediction error of the sample is between the threshold for suspicious and the threshold for alert, further analysis should be performed remotely.

To evaluate the proposed idea, we use MiBench benchmark as our normal applications and also an application that regularly communicates with a server. This combination could emulate better the real-world IoT behavior were a limited number of applications are executed and a Internet connection is established to exchange data. For our attack libraries, we use SATHV and a set of malware developed as part of the iMRC [131] project. The set of malware developed for iMRC includes DoS attacks, modification of application responses, data-logger using memory dump, data-logger to send data to a malicious website, Man-in-the-Middle injection, reverse shell, and Cryptolocker. The total number of malware applications is 37, out of which 25 are SATHV and 12 belong to generic malware families, while the total number of normal applications is 29. The platform used in this experimentation is a Raspberry Pi3. To train and test our implementation we use a 70-30% split of the normal applications, while all malware are used for testing.

The classification metrics can be viewed in Table 6.1.

Local + Remote	**F-score**	**Sensitivity**	**Precision**	**FPR**	**Accuracy**	**% of Positives detected locally**	**% of Positives detected remotely**
CNN [***thresholds 75%, 99.85%***] ***+ LSTM20***	22.07%	12.40% (22/25 SATHV) & (11/12 generic malware)	∼100%	∼0.0% (1/9)	30.13%	1.76% (11/37)	10.64% (22/37)

Table 6.1: Classification metrics for the Local CNN (HW) + Remote ML when considering generic malware.

As we can see from Table 6.1 we are able to detect 22 out of 25 SATHV and 11 out of 12 generic malware. Although the table shows 0% FPR, there are 2 FP samples that result in 1 normal application being detected as malware. Considering the previously mentioned actions and since the frequency of these FPs is low, restricting the execution of sensitive content to the target device may not result in a high overhead. The low statistics despite the high detection rate is due to the fact that we perform anomaly detection on both the local and remote system, which leads us to keep only samples that deviate strongly from normal behavior. Moreover, in classification where we specify the target label, the ML algorithm can better learn to distinguish between normal and malicious behavior, while

in anomaly detection it only learns the normal behavior. In addition, anomaly detection allows us to build a model that can detect zero-day vulnerability attacks with greater success since the behavior of the attacks will most probably differ from the normal execution. But all of the above intuitions depend heavily on the model's ability to learn normal behavior. Last, under normal operation we only send 8.82kB and under attack 871.62kB of data per minute when the total size of data extracted is 5760kB. The above shows that the proposed idea has the potential to be used in generic malware detection as well.

To implement the local complex ML (CNN) in hardware, we use a library developed by the Conseil Européen pour la Recherche Nucléaire (CERN) that allows Python code to be implemented directly in hardware via their IP. The library is called hls4ml and is available at [176]. With this library, we can translate Python code into a Hardware Description Language (HDL) and also apply many optimizations depending on latency or area requirements. This library can give us a first intuition of the implementation. In our case, we set the maximum reuse of resources to reduce the required area, since the ML should provide us with a decision every 1 ms, i.e., before the new sample arrives for evaluation. The resource usage can be seen in Figure 6.1, while the comparison with our custom IP can be seen in Table 6.2.

```
================================================================
== Utilization Estimates
================================================================
* Summary:
+-----------------+---------+-------+--------+-------+-----+
|       Name      | BRAM_18K| DSP48E|   FF   |  LUT  | URAM|
+-----------------+---------+-------+--------+-------+-----+
|DSP              |        -|      -|       -|      -|    -|
|Expression       |        -|      -|       0|     34|    -|
|FIFO             |        0|      -|     606|   3180|    -|
|Instance         |        2|      1|    9234|  10157|    -|
|Memory           |        -|      -|       -|      -|    -|
|Multiplexer      |        -|      -|       -|     36|    -|
|Register         |        -|      -|       6|      -|    -|
+-----------------+---------+-------+--------+-------+-----+
|Total            |        2|      1|    9846|  13407|    0|
+-----------------+---------+-------+--------+-------+-----+
|Available        |      280|    220|  106400|  53200|    0|
+-----------------+---------+-------+--------+-------+-----+
|Utilization (%)  |    ~0   |   ~0  |       9|     25|    0|
+-----------------+---------+-------+--------+-------+-----+

+ Detail:
    * Instance:
    +-------------------------------------------------------+-------------------------------------------------------+---------+-------+------+------+-----+
    |                        Instance                       |                         Module                        | BRAM_18K| DSP48E|  FF  |  LUT | URAM|
    +-------------------------------------------------------+-------------------------------------------------------+---------+-------+------+------+-----+
    |Block_proc_U0                                          |Block_proc                                             |        0|      0|     2|    11|    0|
    |conv_1d_cl_array_array_ap_fixed_4u_config5_U0          |conv_1d_cl_array_array_ap_fixed_4u_config5_s           |        0|      0|  1366|  1163|    0|
    |conv_1d_cl_array_array_ap_fixed_8u_config2_U0          |conv_1d_cl_array_array_ap_fixed_8u_config2_s           |        0|      0|  1419|  1438|    0|
    |dense_array_array_ap_fixed_16_1_5_3_0_20u_config10_U0  |dense_array_array_ap_fixed_16_1_5_3_0_20u_config10_s   |        1|      0|  3035|  2610|    0|
    |dense_array_array_ap_fixed_16_1_5_3_0_6u_config13_U0   |dense_array_array_ap_fixed_16_1_5_3_0_6u_config13_s    |        1|      1|  1635|  1477|    0|
    |linear_array_array_ap_fixed_20u_linear_config11_U0     |linear_array_array_ap_fixed_20u_linear_config11_s      |        0|      0|     3|   384|    0|
    |linear_array_array_ap_fixed_4u_linear_config6_U0       |linear_array_array_ap_fixed_4u_linear_config6_s        |        0|      0|    13|   175|    0|
    |linear_array_array_ap_fixed_6u_linear_config14_U0      |linear_array_array_ap_fixed_6u_linear_config14_s       |        0|      0|   219|   345|    0|
    |linear_array_array_ap_fixed_8u_linear_config3_U0       |linear_array_array_ap_fixed_8u_linear_config3_s        |        0|      0|    13|   247|    0|
    |pooling1d_cl_array_array_ap_fixed_4u_config8_U0        |pooling1d_cl_array_array_ap_fixed_4u_config8_s         |        0|      0|   342|   495|    0|
    |relu_array_array_ap_fixed_20u_relu_config12_U0         |relu_array_array_ap_fixed_20u_relu_config12_s          |        0|      0|   645|   968|    0|
    |relu_array_array_ap_fixed_4u_relu_config7_U0           |relu_array_array_ap_fixed_4u_relu_config7_s            |        0|      0|   207|   328|    0|
    |relu_array_array_ap_fixed_8u_relu_config4_U0           |relu_array_array_ap_fixed_8u_relu_config4_s            |        0|      0|   335|   516|    0|
    +-------------------------------------------------------+-------------------------------------------------------+---------+-------+------+------+-----+
    |Total                                                  |                                                       |        2|      1|  9234| 10157|    0|
    +-------------------------------------------------------+-------------------------------------------------------+---------+-------+------+------+-----+
```

Figure 6.1: Vivado resource usage of the local ML (CNN) for SATHV and generic malware detection after synthesis.

IP	BRAM_18K	DSP48E	FF	LUT
Custom Logistic Regression	0	10 (4.54%)	1316 (1.23%)	1051 (1.97%)
CNN (keras-hls4ml) ***(8 Conv1d, 4 Conv1d, 20 Dense, 6 Dense)***	2 (0.45%)	1 (0.35%)	9846 (9%)	13407 (25%)

Table 6.2: Resource comparison between our custom Logistic Regression IP and the hardware model produced using a keras-hls4ml CNN.

As we can see from Table 6.2, the local ML used for anomaly detection increases the resources used in the hardware considering the Flip-Flops (FF) and Look-Up Tables (LUTs) while it used less Digital Signal Processing (DSPs) elements. Our custom IP was based in logistic regression, which can be considered a one-layer NN. For this task we used a CNN, which can be described as a DNN, which increases the hidden layers and the resources used. Though, increasing the complexity of the detection problem, increases the complexity of the models to be used, and consequently the overheads, in this case the hardware resources. To decrease the resources used, there exist strategies such as model pruning i.e., cutting off connections that do not contribute too much in the final result, and/or quantization. Bu these come at the cost of a loss in precision. As the local ML does not to be as precise as the remote ML, these drawbacks can be further studied and evaluated.

Combining HPCs and other sources of information Further, HPCs have shown promising results in detecting SATHV and generic malware. In a number of works, they have been used to propose detection mechanisms [59], [61], [78], [177]. However, despite the promising results, there are a number of reasons that may reduce their effectiveness. First, their counts are not deterministic, i.e., in different runs of the same application, the extracted values are not the same. [81] showed that HPCs can provide deterministic results in cases where the test environment is tightly controlled. But such a control environment can be difficult to set up in real case scenarios. This indeterminacy can lead to many misclassifications in cases where malware and normal applications have very similar behavior. Machine learning algorithms can learn to separate the two, but due to the noise between runs, this can lead to samples incorrectly crossing the "detection line". Other works have used API calls for malware detection [71], [72]. API calls can provide us with higher level information about how the malware works, such as creating and opening files, listening on a port, changing permissions, etc. Combining the two sources of information and/or other can lead to better results. The challenge is to combine the high-frequency extracted HPC data with the less frequent API calls while

reducing the dimensionality, e.g., using Linear Discriminant Analysis (LDA) to create simpler ML models. We believe this area could be interesting for malware analysis.

Past and future samples used for prediction Finally, in most of this work we have used ML models that use the previous samples to predict the current one. We have explained how a LIFO can act as a buffer for past samples before storing them in a local storage in case of a suspicious sample. But, it might be more interesting to predict future m samples instead of just the current sample. This could allow an anomaly detection mechanism trained only on normal data to learn more information about the execution of normal applications, thus reducing FPs. In a general case, an application performs a series of actions to prepare execution for a future series of actions. Learning past behavior and trying to predict the m future patterns, rather than just the current one, can reduce FPs because we are not just deciding the legitimacy of one step, but rather multiple future steps. However, predicting m future samples would require transferring more samples from the local system to the remote system. In the solution proposed in this work, we observed that the samples labeled as suspicious in the local system are close to each other, which reduces the memory overhead since the required samples from the past were often already stored due to a sample previously labeled as suspicious. Thus, it would be interesting to test this hypothesis and implement the logic of transferring only the necessary samples.

6.4 Final words

In this research, we have investigated and evaluated different approaches and finally proposed a solution that can be useful for SATHV and malware detection on resource-constrained devices. We hope that this work will help the community better understand the limitations of IoT devices in the context of malware detection. We also hope that this work can be inspiring and that the community will continue to look for ways to better apply detection techniques to the IoT and further improve our work.

Appendices

A

Appendix A

A.1 HPC modification by a user-space application

Case1: Read PMU registers from user-space without using a kernel-module to enable access As mentioned earlier, access to the PMU registers is restricted by default for applications in user space. Access could be enabled by a kernel module. If the attacker cannot load a malicious kernel module to enable access, this could be accomplished by writing a user-space script that must carefully change the values of certain registers.

The first step is to find out the value of the DBGDRAR register, which defines the physical base address of a 4KB-aligned memory-mapped debug component usually a ROM table that locates and describes the memory-mapped debug components in the system. This tells us where the debug components are mapped in memory. Then we need to read the DBGDSAR register that defines the offset from the base address defined by DBGDRAR of the physical base address of the debug registers for the processor. The physical address of the debugging logic CPU is then DBGDRAR + DBGDSAR. With this address as the base address of the PMU registers, we can access them by writing to the corresponding offset.

Then they must write the value "0xC5ACCE55" into the EDLAR resister, which allows or disallows access to the external debug registers through a memory mapped interface. As described in the documentation [157], this register sets the optional software lock that provides a lock to prevent access to the debug registers via the memory mapped interface. Using this lock mechanism reduces the risk of accidentally corrupting the contents of the debug registers. However, it cannot prevent all accidental or malicious corruption. Also, we need to write any value except "0xC5ACCE55" to the OSLAR register, that provides a lock for the debug registers, to unlock them. In the next step, we

need to write to the PMLAR register, which is the PMU Lock Access Register. This register also provides a software lock for writes to the PMU registers through the memory mapped interface. If we write "0xC5ACCE55", we can write to the PMU registers through the memory-mapped interface. These are the basic steps to enable modification of the PMU registers through the memory mapped interface.

After we have enabled access through the memory mapped interface, all we need to do is write a "1" to PMUSERENR. PMUSERENR enables or disables user mode access to the PMU. This now allows us to access the PMUs directly with assembly code from user space via the registers of the system control coprocessor (CP15).

```
root@zybo45:/home/fpga/read_memory_Thesis_example# ./malware_target_PMU -c 0 --test_case 1
value of DBGDRAR is F8800003.
The value of DBGDRAR_physical_address is F8800000
The of DBGDSAR_physical_address = 90000
The CPU debug logic physical address is F8890000 i.e. DBGDRAR + DBGDSAR
Reading DBGAUTHSTATUS ... !
If FF all debugging modes are enabled!
DBGAUTHSTATUS == FF
Allow access to the PMU registers through a memory-mapped interface by writing the value 0xC5ACCE55 to PMLAR register
Enable user access register [PMUSERENR] is currently 0
If PMUSERENR[0] == 0 then user access to the PMU is disabled. Enable it ...
         Write '1' to bit 0 of the user access register [PMUSERENR] to activate it
         Successfully enabled user access to PMUs without a Kernel-Module!

--------------------------------------------------------------------------------
Case 1, Read PMU counters from user space without using a kernel-module to enable access
--------------------------------------------------------------------------------

PMU was enabled
Counter 0 configured with the event 1
Counter 1 configured with the event 4
Counter 2 configured with the event 70
Counter 3 configured with the event 10
Counter 4 configured with the event 5
Counter 5 configured with the event 50
Performance monitor results
Loop: 0, 7598 18524 20860 3947 371 603 173102
Loop: 1, 11530 24107 25661 5088 1032 897 383803
Loop: 2, 4668 10459 11633 2305 242 272 469774
Loop: 3, 11497 24137 25561 5052 1036 805 667082

 --------------  Finish the test case ---------------
```

Figure A.1: PMU interference, test case 1.

Case2: Change the HPC events monitored by the detection mechanism In this test case, in order to modify the hardware events monitored by our local detection system, an attacker must enable memory-mapped access to the PMU, as explained previously, and also do the following:

The attackers must modify the PMXEVTYPER$_i$ register that determines which event increments event counter i. By writing the hexadecimal value of a different HPC event configuration to this register, they effectively change the monitored behavior.

Case3: Reset PMU counters In this test case, after enabling memory access to the PMUs discussed earlier, the attacker must make the following changes:

```
root@zybo45:/home/fpga/read_memory_Thesis_example# ./malware_target_PMU -c 0 --test_case 2
_____________________________________________________________________________________
Case 2, Change the hardware HPC events monitored by the detection mechanism!
_____________________________________________________________________________________

Event of PMU Counter 0 was 63
Event of PMU Counter 1 was 64
Event of PMU Counter 2 was 65
Event of PMU Counter 3 was 66
Event of PMU Counter 4 was 67
Event of PMU Counter 5 was 69

Change the hardware HPC event by modifying PMXEVTYPER*

We modified the event of PMU Counter 0 to 63
We modified the event of PMU Counter 1 to 64
We modified the event of PMU Counter 2 to 65
We modified the event of PMU Counter 3 to 66
We modified the event of PMU Counter 4 to 67
We modified the event of PMU Counter 5 to 69

 ---------------  Finish the test case ---------------
```

Figure A.2: PMU interference, test case 2.

The attacker must modify the configuration of the PMCR register, which provides details about the PMU implementation, including the number of counters implemented, and configures and controls the counters. To reset all counters, we need to write a "1" in bit [1] and bit [2]. This will reset the cycle counter and any other HPC event counter to zero.

```
root@zybo45:/home/fpga/read_memory_Thesis_example# ./malware_target_PMU -c 0 --test_case 3

---------------------------
Case 3, Reset PMU counters!
---------------------------

PMU was enabled
Counter 0 configured with the event 1
Counter 1 configured with the event 4
Counter 2 configured with the event 70
Counter 3 configured with the event 10
Counter 4 configured with the event 5
Counter 5 configured with the event 50

Sleep 3 seconds to allow the counters to increase!

35921 , 141778 , 140379 , 22590 , 3133 , 5595

Now reset the counters!

0 , 247 , 498 , 90 , 12 , 17

 ---------------  Finish the test case ---------------
```

Figure A.3: PMU interference, test case 3.

Case4: Disable HPC counters For this final test case, the attacker must make the following changes to disable counting of events by the HPC registers:

The attacker must modify the configuration of register PMCNTENCLR, which disables the cycle count register PMCCNTR and all implemented event counters HPC_i. Reading this register indicates which counters are enabled. To disable the HPC counters, the attacker must write the value "0x00000000" to the 32-bit register. Bit [31] disables the cycle counter, while bit[5:0] disables the six HPC event counters.

As a final step, to hide their manipulation, the attackers must perform the following steps:

The attackers must write "0x00000000" in PMUSERENR to disable access to the PMU from user space. Then, they must write "0x00" in PMLAR and also EDLAR to disable access to the PMUs from the memory-mapped interface.

```
root@zybo45:/home/fpga/read_memory_Thesis_example# ./malware_target_PMU -c 0 --test_case 4

----------------------------
Test 4, Disable PMU counters!
----------------------------

PMU was enabled
Counter 0 configured with the event 1
Counter 1 configured with the event 4
Counter 2 configured with the event 70
Counter 3 configured with the event 10
Counter 4 configured with the event 5
Counter 5 configured with the event 50

Sleep 3 seconds to allow the counters to increase!

73263 , 222750 , 223677 , 39196 , 5279 , 6672

Now sleep again to see if we properly DISABLED? Sleep 3 seconds to verify counters didn't increase

73263 , 222750 , 223677 , 39196 , 5279 , 6672

 --------------  Finish the test case ---------------
```

Figure A.4: PMU interference, test case 4.

A.2 Hardware Local ML implementation

We implemented the hardware version of our local-remote in Vivado. In the following we provide more details about the implementation.

Figure A.5 presents the block diagram of the hardware design. The IP is clocked at the 100MHz AXI clock. The ip_supervisor_0 module implements the IP_HARDWARE_IA and the local storage presented in Figure 5.7. The module processing_system7_0 is the instantiate of the ARM Cortex A9 CPU available in the Zybo Z7-20 evaluation platform. The remaining modules implement the connection to the AXI bus and the clock generator.

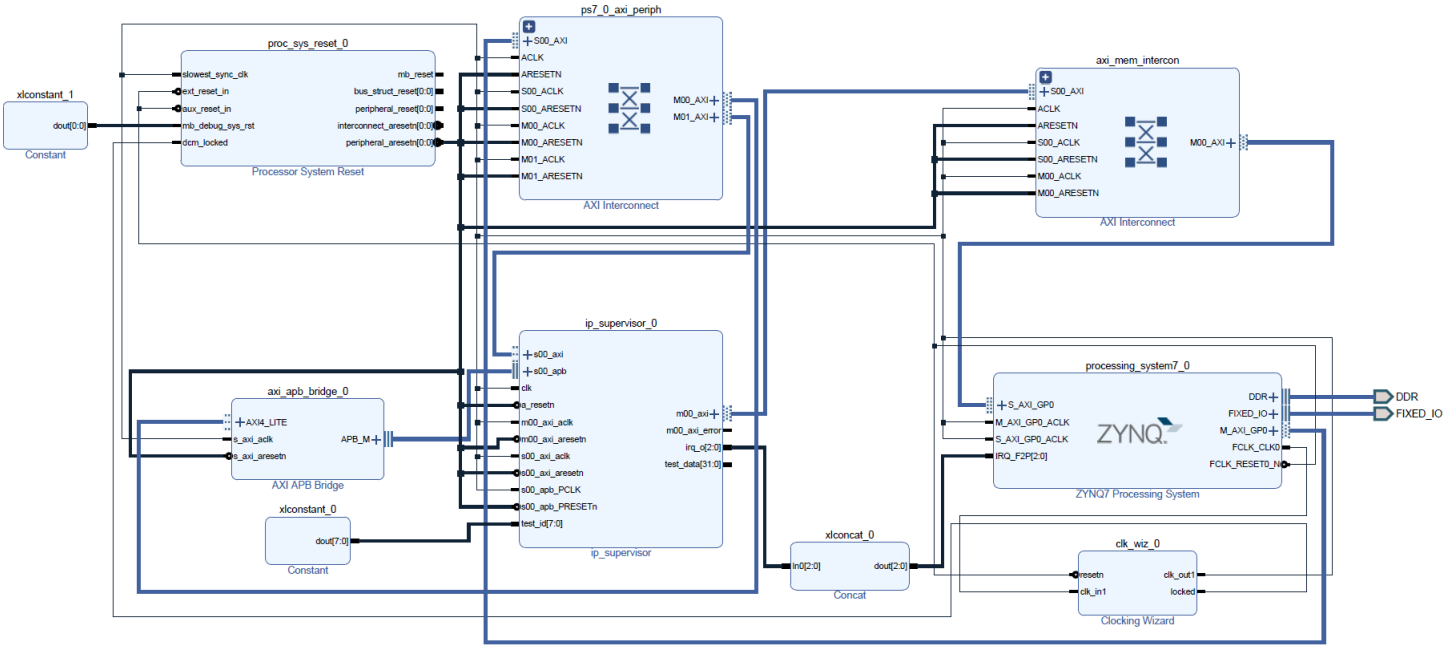


Figure A.5: Vivado block diagram of the proposed design.

In Figure A.6 we represent the hardware resources consumed by the design. We consider only the hardware_ia_top_module and the ip_fifo_inst (highlighted in yellow and green in the figure), since these are the components we added. In the figure we can observe the resources used by all the different components of the design.

Name	Slice LUTs (53200)	Slice Registers (106400)	F7 Muxes (26600)	F8 Muxes (13300)	Slice (13300)	LUT as Logic (53200)	LUT as Memory (17400)	Block RAM Tile (140)	DSPs (220)	Bonded IOPADs (130)	BUFGCTRL (32)	MMCME2_ADV (4)
design_1_wrapper	3611	3804	72	9	1416	3527	84	20.5	10	130	3	1
design_1_i (design_1)	3611	3804	72	9	1416	3527	84	20.5	10	0	3	1
axi_apb_bridge_0 (design_1_axi_apb_bridge_0_0)	47	61	0	0	22	47	0	0	0	0	0	0
axi_mem_intercon (design_1_axi_mem_intercon_0)	175	231	0	0	86	165	10	0	0	0	0	0
clk_wiz_0 (design_1_clk_wiz_0_0)	0	0	0	0	0	0	0	0	0	0	3	1
ip_supervisor_0 (design_1_ip_supervisor_0_0)	2868	2826	72	9	1087	2856	12	20.5	10	0	0	0
U0 (design_1_ip_supervisor_0_0_ip_supervisor)	2868	2823	72	9	1087	2856	12	20.5	10	0	0	0
hardware_ia_top_module_inst (design_1_ip_superv	1051	1316	18	9	496	1039	12	0	10	0	0	0
ip_fifo_inst (design_1_ip_supervisor_0_0_ip_fifo)	89	65	0	0	44	89	0	20	0	0	0	0
ip_master_axi_inst (design_1_ip_supervisor_0_0_ip_	530	428	0	0	170	530	0	0.5	0	0	0	0
ip_slave_axi_inst (design_1_ip_supervisor_0_0_ip_s	116	230	0	0	82	116	0	0	0	0	0	0
ip_version_apb_inst (design_1_ip_supervisor_0_0_i	15	23	0	0	5	15	0	0	0	0	0	0
proc_sys_reset_0 (design_1_proc_sys_reset_0_0)	17	33	0	0	12	16	1	0	0	0	0	0
processing_system7_0 (design_1_processing_system7_	0	0	0	0	0	0	0	0	0	0	0	0
ps7_0_axi_periph (design_1_ps7_0_axi_periph_0)	506	653	0	0	231	445	61	0	0	0	0	0
xlconcat_0 (design_1_xlconcat_0_0)	0	0	0	0	0	0	0	0	0	0	0	0

Figure A.6: Estimated resource utilization of the hardware design.

Finally, Figure A.7 shows the results for the energy consumption as extracted by the Vivado estimator.

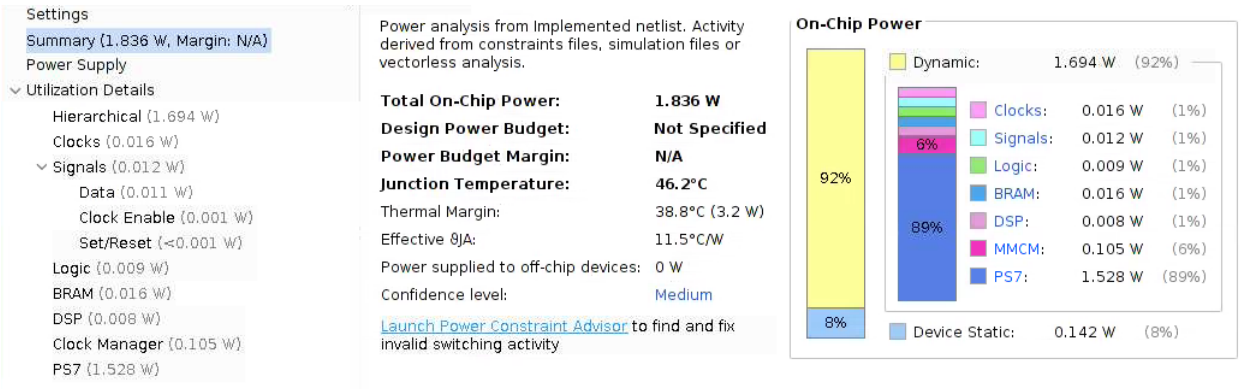


Figure A.7: Estimated energy consumption of the hardware design.

LIST OF FIGURES

List of Tables

ABBREVIATIONS

API	Interface de Programmation d'Applications
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
AUC	Area Under Curve
CacheSCA	attaques de cache par canal auxiliaire
CacheSCA	Cache Side-Channel Attacks
CFI	Control Flow Integrity
CNNs	Convolutional Neural Networks
DDoS	déni de service distribué
DDoS	Distributed Denial-of-Service
DFI	Data Flow Integrity
DNN	Deep Neural Network
DRAM	mémoire dynamique synchrone à accès aléatoire
DRAM	Dynamic Random Access Memory
DTLB	Data Translation Look-aside Buffer
ECC	Error Correcting Codes
EdA	l'Etat de l'Art
EWMA	Exponentially Weighted Moving Average
FF	Flip-Flops
FIFO	First In First Out
FN	False Negative

FNR . . . . . False Negative Rate

FP . . . . . False Positive

FPGA . . . . . Field Programmable Gate Array

FPR . . . . . taux de faux positifs

FPR . . . . . False Positive Rate

GPU . . . . . Graphics Processing Unit

HPCs . . . . . compteurs de performance du matériel

HPCs . . . . . Hardware Performance Counters

HW . . . . . Hardware

IIoT . . . . . l'IoT industriel

IIoT . . . . . Industrial Internet of Things

IoT . . . . . l'internet des objets

IoT . . . . . Internet of Things

IQR . . . . . Interquartile Range

IRQ . . . . . Interrupt Request

ISA . . . . . Instruction Set Architecture

JTAG . . . . . Joint Test Action Group

LLC . . . . . Last Level Cache

LSTM . . . . . Long Short-Term Memory

MI . . . . . Mutual Information

ML . . . . . apprentissage automatique

ML . . . . . Machine Learning

MSE . . . . . Mean Square Error

NN . . . . . Neural Network

OS . . . . . Système d'Exploitation

OS . . . . . Operating System

PL . . . . . Programmable Logic

PMU . . . . . Performance Monitoring Unit

PS . . . . . Processing System

RFE . . . . . Recursive Feature Elimination

RNN . . . . . Recurrent Neural Network

ROC . . . . . Receiver Operating Characteristic

SATHV . . . . . attaques logicielles visant les vulnérabilités matérielles

SATHV . . . . . Software Attacks Targeting Hardware Vulnerabilities

SIGSEGV . . . . . Signal Segmentation Violation

SOC . . . . . System on Chip

SOTA . . . . . State Of The Art

SVM . . . . . Support Vector Machine

SW . . . . . Software

TLB . . . . . Translation Look-aside Buffer

TN . . . . . True Negative

TP . . . . . True Positive

TPR . . . . . True Positive Rate

BIBLIOGRAPHY

[1] C. Kolias, G. Kambourakis, A. Stavrou, and J. Voas, "Ddos in the iot: mirai and other botnets", *Computer*, **50**, *7*, pp. 80–84, 2017.

[2] N. Latto, *What are meltdown and spectre?*, Avast Academy - Security - Other threats, Accessed 7 December 2020, Sep. 2020. [Online]. Available: `https://www.avast.com/c-meltdown-spectre`.

[3] G. U. of Technology, *Meltdown and spectre vulnerabilities in modern computers leak passwords and sensitive data*, website page, Found in Questions and Answers - Can my antivirus detect or block this attack, 2018. [Online]. Available: `https://meltdownattack.com/`.

[4] R. Langner, "Stuxnet: dissecting a cyberwarfare weapon", *IEEE Security & Privacy*, **9**, *3*, pp. 49–51, 2011.

[5] Y. Kim, R. Daly, J. Kim, *et al.*, "Flipping bits in memory without accessing them: an experimental study of dram disturbance errors", *ACM SIGARCH Computer Architecture News*, **42**, *3*, pp. 361–372, 2014.

[6] G. Canivet, P. Maistri, R. Leveugle, J. Clédière, F. Valette, and M. Renaudin, "Glitch and laser fault attacks onto a secure aes implementation on a sram-based fpga", *Journal of cryptology*, **24**, *2*, pp. 247–268, 2011.

[7] S. Tajik, H. Lohrke, F. Ganji, J.-P. Seifert, and C. Boit, "Laser fault attack on physically unclonable functions", *in 2015 workshop on fault diagnosis and tolerance in cryptography (FDTC)*, IEEE, 2015, pp. 85–96.

[8] F. Majéric, B. Gonzalvo, and L. Bossuet, "Jtag fault injection attack", *IEEE Embedded Systems Letters*, **10**, *3*, pp. 65–68, 2017.

[9] K. Rosenfeld and R. Karri, "Attacks and defenses for jtag", *IEEE Design & Test of Computers*, **27**, *1*, pp. 36–47, 2010.

[10] N. Beringuier-Boher, K. Gomina, D. Hely, *et al.*, "Voltage glitch attacks on mixed-signal systems", *in 2014 17th Euromicro Conference on Digital System Design*, IEEE, 2014, pp. 379–386.

[11] T. Korak, M. Hutter, B. Ege, and L. Batina, "Clock glitch attacks in the presence of heating", *in 2014 Workshop on Fault Diagnosis and Tolerance in Cryptography*, IEEE, 2014, pp. 104–114.

[12] P. Qiu, D. Wang, Y. Lyu, and G. Qu, "Voltjockey: breaching trustzone by software-controlled voltage manipulation over multi-core frequencies", *in Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 195–209.

[13] A. Tang, S. Sethumadhavan, and S. Stolfo, "Clkscrew: exposing the perils of security-oblivious energy management", *in 26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 1057–1074.

[14] D. Gruss, D. Bidner, and S. Mangard, "Practical memory deduplication attacks in sandboxed javascript", *in European Symposium on Research in Computer Security*, Springer, 2015, pp. 108–122.

[15] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The spy in the sandbox: practical cache attacks in javascript and their implications", *in Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 1406–1418.

[16] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical", *in 2015 IEEE symposium on security and privacy*, IEEE, 2015, pp. 605–622.

[17] P. Kocher, J. Horn, A. Fogh, *et al.*, "Spectre attacks: exploiting speculative execution", *in 2019 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2019, pp. 1–19.

[18] M. Lipp, M. Schwarz, D. Gruss, *et al.*, "Meltdown: reading kernel memory from user space", *in 27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 973–990.

[19] I. You and K. Yim, "Malware obfuscation techniques: a brief survey", *in 2010 International conference on broadband, wireless computing, communication and applications*, IEEE, 2010, pp. 297–300.

[20] J.-M. Cioranesco, J.-L. Danger, T. Graba, *et al.*, "Cryptographically secure shields", *in 2014 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, IEEE, 2014, pp. 25–31.

[21] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan, "The sorcerer's apprentice guide to fault attacks", *Proceedings of the IEEE*, **94**, *2*, pp. 370–382, 2006.

[22] N. Homma, Y.-i. Hayashi, N. Miura, *et al.*, "Em attack is non-invasive?-design methodology and validity verification of em attack sensor", *in International Workshop on Cryptographic Hardware and Embedded Systems*, Springer, 2014, pp. 1–16.

[23] V. Beroulle, P. Candelier, S. D. Castro, *et al.*, "Laser-induced fault effects in security-dedicated circuits", *in IFIP/IEEE international conference on very large scale integration-system on a chip*, Springer, 2014, pp. 220–240.

[24] H. Sayadi, Y. Gao, H. Mohammadi Makrani, *et al.*, "Towards accurate run-time hardware-assisted stealthy malware detection: a lightweight, yet effective time series cnn-based approach", *Cryptography*, **5**, *4*, p. 28, 2021.

[25] M. S. Mahmoud and A. A. Mohamad, "A study of efficient power consumption wireless communication techniques/modules for internet of things (iot) applications", 2016.

[26] I. Sourmey, *The impact of the communication technology protocol on your iot application's power consumption*, Saftbatteries - Energizing IoT, Accessed 16 September 2022, Jan. 2020. [Online]. Available: `https://www.saftbatteries.com/energizing-iot/impact-communication-technology-protocol-your-iot-application%E2%80%99s-power-consumption`.

[27] **Polychronou, Nikolaos Foivos**, P.-H. Thevenon, P. Maxime, and V. Beroulle, "Securing iot/iiot from software attacks targeting hardware vulnerabilities", *in 2021 19th IEEE International New Circuits and Systems Conference (NEWCAS)*, IEEE, 2021, pp. 1–4.

[28] **Polychronou, Nikolaos Foivos**, P.-H. Thevenon, M. Puys, and V. Beroulle, "Madman: detection of software attacks targeting hardware vulnerabilities", *in 2021 24th Euromicro Conference on Digital System Design (DSD)*, IEEE, 2021, pp. 355–362.

[29] F. Tramer, N. Carlini, W. Brendel, and A. Madry, "On adaptive attacks to adversarial example defenses", *Advances in Neural Information Processing Systems*, **33**, pp. 1633–1645, 2020.

[30] K. L. Dempsey, L. A. Johnson, M. A. Scholl, *et al.*, "Information security continuous monitoring (iscm) for federal information systems and organizations", 2011.

[31] *Learn about malware and how to protect all your devices against it*, `https://www.kaspersky.com/resource-center/preemptive-safety/what-is-malware-and-how-to-protect-against-it`, Accessed: 2022-08-23.

[32] *What is malware + how to prevent malware attacks in 2022*, `https://us.norton.com/internetsecurity-emerging-threats-malware.html`, Accessed: 2022-08-23.

[33] O. Or-Meir, N. Nissim, Y. Elovici, and L. Rokach, "Dynamic malware analysis in the modern era—a state of the art survey", *ACM Computing Surveys (CSUR)*, **52**, *5*, pp. 1–48, 2019.

[34] K. Hu, A. N. Nowroz, S. Reda, and F. Koushanfar, "High-sensitivity hardware trojan detection using multimodal characterization", *in 2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2013, pp. 1271–1276.

[35] M. Potkonjak, A. Nahapetian, M. Nelson, and T. Massey, "Hardware trojan horse detection using gate-level characterization", *in 2009 46th ACM/IEEE Design Automation Conference*, IEEE, 2009, pp. 688–693.

[36] S. McConnell, *Code complete*. Pearson Education, 2004.

[37] V. Van Der Veen, Y. Fratantonio, M. Lindorfer, *et al.*, "Drammer: deterministic rowhammer attacks on mobile platforms", *in Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 1675–1689.

[38] E. Tromer, D. A. Osvik, and A. Shamir, "Efficient cache attacks on aes, and countermeasures", *Journal of Cryptology*, **23**, *1*, pp. 37–71, 2010.

[39] O. Acıiçmez, B. B. Brumley, and P. Grabher, "New results on instruction cache attacks", *in International Workshop on Cryptographic Hardware and Embedded Systems*, Springer, 2010, pp. 110–124.

[40] C. Percival, *Cache missing for fun and profit*, 2005.

[41] O. Acıiçmez and W. Schindler, "A vulnerability in rsa implementations due to instruction cache analysis and its demonstration on openssl", *in Cryptographers' Track at the RSA Conference*, Springer, Berlin, Heidelberg, 2008, pp. 256–273.

[42] O. Aciiçmez, "Yet another microarchitectural attack: exploiting i-cache", *in Proceedings of the 2007 ACM workshop on Computer security architecture*, 2007, pp. 11–18.

[43] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation leak-aside buffer: defeating cache side-channel protections with {tlb} attacks", *in 27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 955–972.

[44] G. Didier and C. Maurice, "Calibration done right: noiseless flush+ flush attacks", *in International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer, 2021, pp. 278–298.

[45] D. Wang, Z. Qian, N. Abu-Ghazaleh, and S. V. Krishnamurthy, "Papp: prefetcher-aware prime and probe side-channel attack", *in Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 1–6.

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

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

[48] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "{Armageddon}: cache attacks on mobile devices", *in 25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 549–564.

[49] G. Irazoqui, T. Eisenbarth, and B. Sunar, "A shared cache attack that works across cores and defies vm sandboxing–and its application to aes", *in 2015 IEEE Symposium on Security and Privacy*, IEEE, 2015, pp. 591–604.

[50] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer. js: a remote software-induced fault attack in javascript", *in International conference on detection of intrusions and malware, and vulnerability assessment*, Springer, 2016, pp. 300–321.

[51] F. Yao, A. S. Rakin, and D. Fan, "Deephammer: depleting the intelligence of deep neural networks through targeted chain of bit flips", *in 29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, S. Capkun and F. Roesner, Eds., USENIX Association, 2020, pp. 1463–1480. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/yao.

[52] S. Hong, P. Frigo, Y. Kaya, C. Giuffrida, and T. Dumitraș, "Terminal brain damage: exposing the graceless degradation in deep neural networks under hardware fault attacks", *in 28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 497–514.

[53] A. S. Rakin, Z. He, and D. Fan, "Bit-flip attack: crushing neural network with progressive bit search", *in Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2019, pp. 1211–1220.

[54] Y. Xiao, X. Zhang, Y. Zhang, and R. Teodorescu, "One bit flips, one cloud flops:{cross-vm} row hammer attacks and privilege escalation", *in 25th USENIX security symposium (USENIX Security 16)*, 2016, pp. 19–35.

[55] P. Kocher, J. Horn, A. Fogh, *et al.*, "Spectre attacks: exploiting speculative execution", *Communications of the ACM*, **63**, 7, pp. 93–101, 2020.

[56] S. Gibbs, *Meltdown and spectre: "worst ever" cpu bugs affect virtually all computers*, Data and computer security, Accessed 28 September 2022, Jan. 2018. [Online]. Available: https://web.archive.org/web/20180106114401/https://www.theguardian.com/technology/2018/jan/04/meltdown-spectre-worst-cpu-bugs-ever-found-affect-computers-intel-processors-security-flaw.

[57] ARM, *Speculative processor vulnerability*, Arm Security Center, Accessed 28 September 2022. [Online]. Available: https://developer.arm.com/Arm%5C%20Security%5C%20Center/Speculative%5C%20Processor%5C%20Vulnerability.

[58] Intel, *Affected processors: transient execution attacks & related security issues by cpu*, Software Security Guidance, Accessed 28 September 2022. [Online]. Available: https://www.intel.com/content/www/us/en/developer/topic-technology/

software-security-guidance/processors-affected-consolidated-product-cpu-model.html.

[59] J. Demme, M. Maycock, J. Schmitz, *et al.*, "On the feasibility of online malware detection with performance counters", *ACM SIGARCH Computer Architecture News*, **41**, *3*, pp. 559–570, 2013.

[60] M. Risks, *Malware risks and mitigation report. bits-the financial services roundtable (2011)*, https://www.nist.gov/system/files/documents/itl/BITS-Malware-Report-Jun2011.pdf, 2011.

[61] A. Gupta, "Assessing hardware performance counters for malware detection", Ph.D. dissertation, Boston University, 2017.

[62] B. Herzog, S. Reif, J. Preis, W. Schröder-Preikschat, and T. Hönig, "The price of meltdown and spectre: energy overhead of mitigations at operating system level", *in Proceedings of the 14th European Workshop on Systems Security*, 2021, pp. 8–14.

[63] A. Naway and Y. Li, "A review on the use of deep learning in android malware detection", *arXiv preprint arXiv:1812.10360*, 2018.

[64] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications", *ACM Transactions on Information and System Security (TISSEC)*, **13**, *1*, pp. 1–40, 2009.

[65] C. Bresch, "Approches, stratégies, et implémentations de protections mémoire dans les systèmes embarqués critiques et contraints", Ph.D. dissertation, Université Grenoble Alpes, 2020.

[66] H. Hu, C. Qian, C. Yagemann, *et al.*, "Enforcing unique code target property for control-flow integrity", *in Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1470–1486.

[67] C. Song, B. Lee, K. Lu, W. Harris, T. Kim, and W. Lee, "Enforcing kernel security invariants with data flow integrity.", *in NDSS*, 2016.

[68] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, "Preventing memory error exploits with wit", *in 2008 IEEE Symposium on Security and Privacy (sp 2008)*, IEEE, 2008, pp. 263–277.

[69] C. Bresch, D. Hély, R. Lysecky, S. Chollet, and I. Parissis, "Trustflow-x: a practical framework for fine-grained control-flow integrity in critical systems", *ACM Transactions on Embedded Computing Systems (TECS)*, **19**, *5*, pp. 1–26, 2020.

[70] N. Christoulakis, G. Christou, E. Athanasopoulos, and S. Ioannidis, "Hcfi: hardware-enforced control-flow integrity", *in Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, 2016, pp. 38–49.

[71] G. Liang, J. Pang, and C. Dai, "A behavior-based malware variant classification technique", *International Journal of Information and Education Technology*, **6**, *4*, p. 291, 2016.

[72] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware guard extension: using sgx to conceal cache attacks", *in International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer, 2017, pp. 3–24.

[73] T. Willhalm and R. Dementiev, *Intel performance counter monitor - a better way to measure cpu utilization*, [Online]. Available: https://www.intel.com/content/www/us/en/develo counter-monitor.html Accessed on 7.9.2022, 2017.

[74] A. Developer, *Cortex-a9 technical reference manual r4p1, about the performance monitoring unit*, [Online]. Available: https://developer.arm.com/documentation/ddi0388/i/perform monitoring-unit/about-the-performance-monitoring-unit Accessed on 7.9.2022.

[75] F. EmbedDev, *Risc-v instruction set manual, volume i: risc-v user-level isa , 20191214 - december 2019, hardware performance counters*, [Online]. Available: https://five-embeddev.com/riscv-isa-manual/latest/counters.html#hardware-performance-counters Accessed on 7.9.2022.

[76] B. Gulmezoglu, A. Moghimi, T. Eisenbarth, and B. Sunar, "Fortuneteller: predicting microarchitectural attacks via unsupervised deep learning", *arXiv preprint arXiv:1907.03651*, 2019.

[77] M. Ozsoy, K. N. Khasawneh, C. Donovick, I. Gorelik, N. Abu-Ghazaleh, and D. Ponomarev, "Hardware-based malware detection using low-level architectural features", *IEEE Transactions on Computers*, **65**, *11*, pp. 3332–3344, 2016.

[78] N. Patel, A. Sasan, and H. Homayoun, "Analyzing hardware based malware detectors", *in 2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, IEEE, 2017, pp. 1–6.

[79] S. P. Kadiyala, P. Jadhav, S.-K. Lam, and T. Srikanthan, "Hardware performance counter-based fine-grained malware detection", *ACM Transactions on Embedded Computing Systems (TECS)*, **19**, *5*, pp. 1–17, 2020.

[80] Z. He, A. Rezaei, H. Homayoun, and H. Sayadi, "Deep neural network and transfer learning for accurate hardware-based zero-day malware detection", *in Proceedings of the Great Lakes Symposium on VLSI 2022*, 2022, pp. 27–32.

[81] V. Weaver and J. Dongarra, "Can hardware performance counters produce expected, deterministic results", *in Proceedings of Third Workshop on Functionality of Hardware Performance Monitoring*, 2010.

[82] M. Chiappetta, E. Savas, and C. Yilmaz, "Real time detection of cache-based side-channel attacks using hardware performance counters", *Applied Soft Computing*, **49**, pp. 1162–1174, 2016.

[83] J. Cho, T. Kim, S. Kim, M. Im, T. Kim, and Y. Shin, "Real-time detection for cache side channel attack using performance counter monitor", *Applied Sciences*, **10**, *3*, p. 984, 2020.

[84] C. Li and J.-L. Gaudiot, "Online detection of spectre attacks using microarchitectural traces from performance counters", *in 2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, IEEE, 2018, pp. 25–28.

[85] M. Payer, "Hexpads: a platform to detect "stealth" attacks", *in International Symposium on Engineering Secure Software and Systems*, Springer, 2016, pp. 138–154.

[86] B. A. Ahmad, "Real time detection of spectre and meltdown attacks using machine learning", *arXiv preprint arXiv:2006.01442*, 2020.

[87] A. Garcia-Serrano, "Anomaly detection for malware identification using hardware performance counters", *arXiv preprint arXiv:1508.07482*, 2015.

[88] Z. He, A. Raghavan, G. Hu, S. Chai, and R. Lee, "Power-grid controller anomaly detection with enhanced temporal deep learning", *in 2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, IEEE, 2019, pp. 160–167.

[89] J. J. Blount, D. R. Tauritz, and S. A. Mulder, "Adaptive rule-based malware detection employing learning classifier systems: a proof of concept", *in 2011 IEEE 35th Annual Computer Software and Applications Conference Workshops*, 2011, pp. 110–115. DOI: 10.1109/COMPSACW.2011.28.

[90] H. Edquist, P. Goodridge, and J. Haskel, "The internet of things and economic growth in a panel of countries", *Economics of Innovation and New Technology*, **30**, *3*, pp. 262–283, 2021.

[91] M. Mushtaq, A. Akram, M. K. Bhatti, M. Chaudhry, V. Lapotre, and G. Gogniat, "Nights-watch: a cache-based side-channel intrusion detector using hardware performance counters", *in Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*, 2018, pp. 1–8.

[92] M. Mushtaq, J. Bricq, M. K. Bhatti, *et al.*, "Whisper: a tool for run-time detection of side-channel attacks", *IEEE Access*, **8**, pp. 83 871–83 900, 2020.

[93] Z. Tong, Z. Zhu, Z. Wang, L. Wang, Y. Zhang, and Y. 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*), IEEE, 2020, pp. 919–926.

[94] M. Schwarz, M. Lipp, D. Moghimi, *et al.*, "Zombieload: cross-privilege-boundary data sampling", *in Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 753–768.

[95] Z. He, T. Miari, H. M. Makrani, M. Aliasgari, H. Homayoun, and H. Sayadi, "When machine learning meets hardware cybersecurity: delving into accurate zero-day malware detection", *in 2021 22nd International Symposium on Quality Electronic Design* (*ISQED*), IEEE, 2021, pp. 85–90.

[96] K. Ott and R. Mahapatra, "Hardware performance counters for embedded software anomaly detection", *in 2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress*(*DASC/PiCom/DataCom/CyberSciTech*), 2018, pp. 528–535. DOI: 10.1109/DASC/PiCom/DataCom/CyberSciTec.2018.00101.

[97] P. Krishnamurthy, R. Karri, and F. Khorrami, "Anomaly detection in real-time multi-threaded processes using hardware performance counters", *IEEE Transactions on Information Forensics and Security*, **15**, pp. 666–680, 2019.

[98] X. Wang, S. Chai, M. Isnardi, S. Lim, and R. Karri, "Hardware performance counter-based malware identification and detection with adaptive compressive sensing", *ACM Transactions on Architecture and Code Optimization* (*TACO*), **13**, *1*, pp. 1–23, 2016.

[99] D. G. Kleinbaum, K. Dietz, M. Gail, M. Klein, and M. Klein, *Logistic regression*. Springer, 2002.

[100] T. G. Nick and K. M. Campbell, "Logistic regression", *Topics in biostatistics*, pp. 273–301, 2007.

[101] M. A. Hearst, S. T. Dumais, E. Osuna, J. Platt, and B. Scholkopf, "Support vector machines", *IEEE Intelligent Systems and their applications*, **13**, *4*, pp. 18–28, 1998.

[102] C.-W. Hsu, C.-C. Chang, C.-J. Lin, *et al.*, *A practical guide to support vector classification*, 2003.

[103] scikit learn, *Support vector machines - complexity*, [Online]. Available: https://scikit-learn.org/stable/modules/svm.html#complexity Accessed on 8.9.2022.

[104] Y. Freund and R. E. Schapire, "Experiments with a new boosting algorithm", *in ICML*, 1996.

[105] S. B. Kotsiantis, "Decision trees: a recent overview", *Artificial Intelligence Review*, **39**, *4*, pp. 261–283, 2013.

[106] T. Chen and C. Guestrin, "Xgboost: a scalable tree boosting system", *in Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 2016, pp. 785–794.

[107] C. M. Bishop, "Neural networks and their applications", *Review of scientific instruments*, **65**, *6*, pp. 1803–1832, 1994.

[108] R. Miikkulainen, J. Liang, E. Meyerson, *et al.*, "Evolving deep neural networks", *in Artificial intelligence in the age of neural networks and brain computing*, Elsevier, 2019, pp. 293–312.

[109] S. Albawi, T. A. Mohammed, and S. Al-Zawi, "Understanding of a convolutional neural network", *in 2017 international conference on engineering and technology (ICET)*, Ieee, 2017, pp. 1–6.

[110] B. Zhao, H. Lu, S. Chen, J. Liu, and D. Wu, "Convolutional neural networks for time series classification", *Journal of Systems Engineering and Electronics*, **28**, *1*, pp. 162–169, 2017.

[111] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors", *nature*, **323**, *6088*, pp. 533–536, 1986.

[112] F. T. Liu, K. M. Ting, and Z.-H. Zhou, "Isolation forest", *in 2008 eighth ieee international conference on data mining*, IEEE, 2008, pp. 413–422.

[113] S. Hochreiter and J. Schmidhuber, "Long short-term memory", *Neural computation*, **9**, *8*, pp. 1735–1780, 1997.

[114] A. Sherstinsky, "Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network", *Physica D: Nonlinear Phenomena*, **404**, p. 132 306, 2020.

[115] P. Malhotra, A. Ramakrishnan, G. Anand, L. Vig, P. Agarwal, and G. Shroff, "Lstm-based encoder-decoder for multi-sensor anomaly detection", Jul. 2016.

[116] B. Zhou, A. Gupta, R. Jahanshahi, M. Egele, and A. Joshi, "Hardware performance counters can detect malware: myth or fact?", *in Proceedings of the 2018 on Asia conference on computer and communications security*, 2018, pp. 457–468.

[117] L. Batina, B. Gierlichs, E. Prouff, M. Rivain, F.-X. Standaert, and N. Veyrat-Charvillon, "Mutual information analysis: a comprehensive study", *Journal of Cryptology*, **24**, *2*, pp. 269–291, 2011.

[118] G. Ver Steeg, *Non-parametric entropy estimation toolbox*, `https://github.com/gregversteeg/NPEET`, 2019.

[119] **Polychronou, Nikolaos-Foivos**, P.-H. Thevenon, M. Puys, and V. Beroulle, "Research for a new solution for the detection of malwares in iot/iiot devices", *Journees C2 2022*, 2022, Available: `https://jc2-2022.inria.fr/files/2022/01/JC2-2022_paper_23.pdf`.

[120] **Polychronou, Nikolaos-Foivos**, P.-H. Thevenon, M. Puys, and V. Beroulle, "A comprehensive survey of attacks without physical access targeting hardware vulnerabilities in iot/iiot devices, and their detection mechanisms", *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, **27**, *1*, pp. 1–35, 2021.

[121] T. A. Akyildiz, C. B. Guzgeren, C. Yilmaz, and E. Savas, "Meltdowndetector: a runtime approach for detecting meltdown attacks", *Future Generation Computer Systems*, **112**, pp. 136–147, 2020.

[122] T. Micro, "Detecting attacks that exploit meltdown and spectre with performance counters", *Retrieved August*, **15**, p. 2018, 2018.

[123] M. Ghasempour, M. Lujan, and J. Garside, *Armor: a run-time memory hot-row detector.(2015)*, [Online]. Available: http://apt.cs.manchester.ac.uk/projects/ARMOR/RowHamm Accessed on 26.6.2020, 2015.

[124] D. Gruss, M. Lipp, M. Schwarz, *et al.*, "Another flip in the wall of rowhammer defenses", *in 2018 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2018, pp. 245–261.

[125] E. Lee, I. Kang, S. Lee, G. E. Suh, and J. H. Ahn, "Twice: preventing row-hammering by exploiting time window counters", *in Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 385–396.

[126] C. Li and J.-L. Gaudiot, "Detecting malicious attacks exploiting hardware vulnerabilities using performance counters", *in 2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, IEEE, vol. 1, 2019, pp. 588–597.

[127] C. Canella, J. Van Bulck, M. Schwarz, *et al.*, "A systematic evaluation of transient execution attacks and defenses", *in 28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 249–266.

[128] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "{Drama}: exploiting {dram} addressing for {cross-cpu} attacks", *in 25th USENIX security symposium (USENIX security 16)*, 2016, pp. 565–581.

[129] S. M. Seyedzadeh, A. K. Jones, and R. Melhem, "Counter-based tree structure for row hammering mitigation in dram", *IEEE Computer Architecture Letters*, **16**, *1*, pp. 18–21, 2016.

[130] P. Stewin, "A primitive for revealing stealthy peripheral-based attacks on the computing platform's main memory", *in International Workshop on Recent Advances in Intrusion Detection*, Springer, 2013, pp. 1–20.

[131] P. Thevenon, S. Riou, D. Tran, *et al.*, "Imrc: integrated monitoring & recovery component, a solution to guarantee the security of embedded systems", *J. Inter-*

net Serv. Inf. Secur., **12**, *2*, pp. 70–94, 2022. DOI: 10.22667/JISIS.2022.05.31.070. [Online]. Available: https://doi.org/10.22667/JISIS.2022.05.31.070.

[132] S. Peng, Q. Zhou, and J. Zhao, "Detection of cache-based side channel attack based on performance counters", *in Proc. 3rd Int. Conf. Artif. Intell. Ind. Eng.(AIIE)*, 2017, pp. 377–381.

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

[134] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: a free, commercially representative embedded benchmark suite", *in Proceedings of the fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538)*, IEEE, 2001, pp. 3–14.

[135] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: characterization and architectural implications", *in Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008, pp. 72–81.

[136] M. Lipp, "Libflush, 2016", *URL https://github. com/IAIK/armageddon.(cited on p. 29)*,

[137] J. Singh and J. Singh, "Challenge of malware analysis: malware obfuscation techniques", *International Journal of Information Security Science*, **7**, *3*, pp. 100–110, 2018.

[138] C. Li and J.-L. Gaudiot, "Challenges in detecting an "evasive spectre"", *IEEE Computer Architecture Letters*, **19**, *1*, pp. 18–21, 2020.

[139] B. K. Iwana and S. Uchida, "An empirical survey of data augmentation for time series classification with neural networks", *Plos one*, **16**, *7*, e0254841, 2021.

[140] S. Narkhede, "Understanding auc-roc curve", *Towards Data Science*, **26**, *1*, pp. 220–227, 2018.

[141] **Polychronou, Nikolaos-Foivos**, P.-H. Thevenon, M. Puys, and V. Beroulle, "A system for detecting malwares in a resources constrained device", 2022, patent number DD22030 ST.

[142] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: vision and challenges", *IEEE internet of things journal*, **3**, *5*, pp. 637–646, 2016.

[143] C. Habib, A. Makhoul, R. Darazi, and R. Couturier, "Real-time sampling rate adaptation based on continuous risk level evaluation in wireless body sensor networks", *in 2017 IEEE 13th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, IEEE, 2017, pp. 1–8.

[144] J. Azar, A. Makhoul, M. Barhamgi, and R. Couturier, "An energy efficient iot data compression approach for edge machine learning", *Future Generation Computer Systems*, **96**, pp. 168–175, 2019.

[145] T. Wang, H. Ke, X. Zheng, K. Wang, A. K. Sangaiah, and A. Liu, "Big data cleaning based on mobile edge computing in industrial sensor-cloud", *IEEE Transactions on Industrial Informatics*, **16**, *2*, pp. 1321–1329, 2019.

[146] Y. J. Kim, C.-H. Park, and M. Yoon, "Film: filtering and machine learning for malware detection in edge computing", *Sensors*, **22**, *6*, p. 2150, 2022.

[147] C. ARM, "Arm cortex-a72 technical reference manual", *ARM, Dec*, 2008. [Online]. Available: `https://developer.arm.com/documentation/100095/0003`.

[148] O. I. Provotar, Y. M. Linder, and M. M. Veres, "Unsupervised anomaly detection in time series using lstm-based autoencoders", *in 2019 IEEE International Conference on Advanced Trends in Information Theory (ATIT)*, IEEE, 2019, pp. 513–517.

[149] K. Sadaf and J. Sultana, "Intrusion detection based on autoencoder and isolation forest in fog computing", *IEEE Access*, **8**, pp. 167 059–167 068, 2020.

[150] Y. Mao, X. Chen, and Y. Luo, "Hvsm: an in-out-vm security monitoring architecture in iaas cloud", pp. 185–192, 2014. DOI: `10.1049/cp.2014.1285`.

[151] Y. Gao, M. Kim, S. Abuadbba, *et al.*, "End-to-end evaluation of federated learning and split learning for internet of things", *arXiv preprint arXiv:2003.13376*, 2020.

[152] J. Depoix and P. Altmeyer, "Detecting spectre attacks by identifying cache side-channel attacks using machine learning", *Advanced Microkernel Operating Systems*, **75**, 2018.

[153] S. Wei, A. Aysu, M. Orshansky, A. Gerstlauer, and M. Tiwari, "Using power-anomalies to counter evasive micro-architectural attacks in embedded systems", *in 2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, IEEE, 2019, pp. 111–120.

[154] A. P. Kuruvila, X. Meng, S. Kundu, G. Pandey, and K. Basu, "Explainable machine learning for intrusion detection via hardware performance counters", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2022.

[155] H. Wang, H. Sayadi, S. M. P. Dinakarrao, A. Sasan, S. Rafatirad, and H. Homayoun, "Enabling micro ai for securing edge devices at hardware level", *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, **11**, *4*, pp. 803–815, 2021.

[156] Y. Wang, Y.-a. Tan, W. Zhang, Y. Zhao, and X. Kuang, "An adversarial attack on dnn-based black-box object detectors", *Journal of Network and Computer Applications*, **161**, p. 102 634, 2020.

[157] C. ARM, "A9 mpcore technical reference manual", *Revision: r4p1*, 2010.

[158] H. Cho, P. Zhang, D. Kim, *et al.*, "Prime+ count: novel cross-world covert channels on arm trustzone", *in Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 441–452.

[159] M. B. Bahador, M. Abadi, and A. Tajoddin, "Hpcmalhunter: behavioral malware detection using hardware performance counters and singular value decomposition", *in 2014 4th International Conference on Computer and Knowledge Engineering (ICCKE)*, IEEE, 2014, pp. 703–708.

[160] Y. Liu, L. Wei, B. Luo, and Q. Xu, "Fault injection attack on deep neural network", *in 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, IEEE, 2017, pp. 131–138.

[161] J. Breier, X. Hou, D. Jap, L. Ma, S. Bhasin, and Y. Liu, "Practical fault attack on deep neural networks", *in Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2204–2206.

[162] D. Jap, Y.-S. Won, and S. Bhasin, "Fault injection attacks on softmax function in deep neural networks", *in Proceedings of the 18th ACM International Conference on Computing Frontiers*, 2021, pp. 238–240.

[163] W. Liu, C.-H. Chang, F. Zhang, and X. Lou, "Imperceptible misclassification attack on deep learning accelerator by glitch injection", *in 2020 57th ACM/IEEE Design Automation Conference (DAC)*, IEEE, 2020, pp. 1–6.

[164] A. S. Rakin, Y. Luo, X. Xu, and D. Fan, "Deep-dup: an adversarial weight duplication attack framework to crush deep neural network in multi-tenant FPGA", *in 30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, M. Bailey and R. Greenstadt, Eds., USENIX Association, 2021, pp. 1919–1936. [Online]. Available: `https://www.usenix.org/conference/usenixsecurity21/presentation/rakin`.

[165] M. Vestias and H. Neto, "Trends of cpu, gpu and fpga for high-performance computing", *in 2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, IEEE, 2014, pp. 1–6.

[166] I. Tsmots, O. Skorokhoda, and V. Rabyk, "Hardware implementation of sigmoid activation functions using fpga", *in 2019 IEEE 15th International Conference on the Experience of Designing and Application of CAD Systems (CADSM)*, IEEE, 2019, pp. 34–38.

[167] Z. Li, Y. Zhang, B. Sui, Z. Xing, and Q. Wang, "Fpga implementation for the sigmoid with piecewise linear fitting method based on curvature analysis", *Electronics*, **11**, *9*, p. 1365, 2022.

[168] MQTT, *Mqtt: the standard for iot messaging*, MQTT, Accessed 25 October 2022, 2022. [Online]. Available: https://mqtt.org/.

[169] Google, *Qkeras*, https://github.com/google/qkeras, 2022.

[170] F. Alcaraz, *Fxpmath*, https://github.com/francof2a/fxpmath, 2022.

[171] B. Biggio, I. Corona, D. Maiorca, *et al.*, "Evasion attacks against machine learning at test time", *in Joint European conference on machine learning and knowledge discovery in databases*, Springer, 2013, pp. 387–402.

[172] B. Biggio, B. Nelson, and P. Laskov, "Poisoning attacks against support vector machines", *arXiv preprint arXiv:1206.6389*, 2012.

[173] A. Itagi, S. Krishvadana, K. P. Bharath, and M. Rajesh Kumar, "Fpga architecture to enhance hardware acceleration for machine learning applications", *in 2021 5th International Conference on Computing Methodologies and Communication (ICCMC)*, 2021, pp. 1716–1722. DOI: 10.1109/ICCMC51019.2021.9418015.

[174] S. Gandhare and B. Karthikeyan, "Survey on fpga architecture and recent applications", *in 2019 International Conference on Vision Towards Emerging Trends in Communication and Networking (ViTECoN)*, IEEE, 2019, pp. 1–4.

[175] E. Benhani, L. Bossuet, and A. Aubert, "The security of arm trustzone in a fpga-based soc", *IEEE Transactions on Computers*, **68**, *8*, pp. 1238–1248, 2019.

[176] L. L. Fast Machine, *Hls4ml*, https://github.com/fastmachinelearning/hls4ml, 2022.

[177] D. Kumar, "Hardware-assisted online defense against malware and exploits", *Nanyang Technological University*, 2016.