
Mitigation of Cache Side-Channel Attacks in Virtualised Environments

ABDULLAH ALBALAWI

Computer Science
UNIVERSITY OF YORK

PhD Thesis

October 13, 2022

For my Family...

ABSTRACT

In this thesis, we present several proposed methods that can be integrated to mitigate the attack threats of the cache side-channel attacks in particular and the threats of microarchitectural attacks in general. These methods relied on different viewpoints to address these threats to maintain and preserve the advantages and characteristics of cloud computing. The first method uses memory deduplication features to allow the proposed defence mechanism to reach the shared physical addresses of the sensitive processes to be monitored and identify suspicious behaviours using logistic regression to classify the behaviours according to the readings extracted from the observation of the shared cache lines. This mechanism provides self-protection for the VM and disrupts attackers' results in rare cases of false negatives due to frequent access to the cache lines for monitoring. The second method relied on integrating dynamic and static analysis based on machine and deep learning algorithms. This mechanism monitors suspicious behaviour within the shared virtualised system using hardware performance counters related to the shared cache and affected by the cache side-channel attacks. If any suspicious behaviour of the VM is observed. In that case, the static analysis is run to access the disk images and RAM images of the suspicious VM to extract executable files to be checked against implicit attack characteristics (opcodes) using reverse engineering tools, and then the threat level of the VM is determined using a Softmax classification algorithm. This mechanism develops using static analysis to protect the shared systems with low system overhead and high accuracy. The third method is based on the static analysis of the microarchitectural attacks and logistic regression for classification. This mechanism is designed to ensure the integrity of the shared virtualised system.

ACKNOWLEDGEMENTS

Words are not enough to express my great gratitude to my supervisors, Dr Vassilios Vassilakis and Prof Radu Calinescu. I will be forever grateful to them for their guidance, support, and inspiration for my PhD thesis during my years of study.

I am also grateful to my dear parents for their constant support in all areas of my life and encouraging me to achieve my goals and ambitions. I also profoundly thank my beloved wife and wonderful life partner for her constant support and love and my dear daughter Danah for always being the motivator.

I also extend my sincere thanks to all my sisters and brothers, especially my dear brother Fawaz for his constant support and encouragement during my study period.

I would also like to thank all my friends, especially my dear friend Naif Alasmari, who always stood by me in many circumstances that I experienced during my studies.

Also, I would like to thank Dr Hussain Aljahdali, Dr Mostafa Taha and Dr Yuval Yarom, who were always happy to respond to my emails and answer my inquiries.

I would like to express my gratitude and appreciation to those who have encouraged me while I pursue my PhD study and accomplish my goal. I thank them for their positive impact on my personal and professional lives.

I must pay great thanks to the University of York for giving me the opportunity to be one of its PhD students, facilitating all study requirements and resources, and standing by me in many of the circumstances that occurred to me during my studies.

Also, I pay great thanks to Shaqra University in particular for the financial support for my PhD scholarship and the Ministry of Higher Education of the Kingdom of Saudi Arabia in general.

AUTHOR'S DECLARATION

I declare that this thesis is a presentation of original work and I am the sole author. This work has not previously been presented for an award at this, or any other, University. All sources are acknowledged as References. In this thesis, some materials appeared in the following published or awaiting publication papers in page ix.

SIGNED: DATE:

PUBLICATIONS

1. Albalawi, Abdullah, Vassilios Vassilakis, and Radu Calinescu. "Memory Deduplication as a Protective Factor in Virtualized Systems." *International Conference on Applied Cryptography and Network Security*. Springer, Cham, 2021.
2. Albalawi, Abdullah, Vassilios G. Vassilakis, and Radu Calinescu. "Protecting Shared Virtualized Environments against Cache Side-channel Attacks." In *Proceedings of the 8th International Conference on Information Systems Security and Privacy - ICISSP, 2022*.
3. Albalawi, Abdullah, Vassilios Vassilakis, and Radu Calinescu. "Side-channel Attacks and Countermeasures in Cloud Services and Infrastructures." In *NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2022.

TABLE OF CONTENTS

	Page
List of Tables	xv
List of Figures	xvii
1 Introduction	1
1.1 Project Area and Motivation	2
1.2 Problem Statement	5
1.3 Research Questions and Objectives	6
1.4 Thesis Hypothesis	8
1.5 Thesis Contributions	9
1.6 Thesis Structure	10
2 Literature Review	13
2.1 Cloud Computing Security	14
2.2 Side-channel Attacks	16
2.2.1 Cache Side-channel Attack	17
2.2.2 Other Microarchitectural Attacks	25
2.2.3 Cache and Microarchitectural Attacks Characteristics	26
2.2.4 Cryptographic Systems Vulnerable to Cache Attacks	28
2.3 Multi-Tenancy and Virtualisation	33
2.4 Cache Architecture	34
2.4.1 Cache Addressing	35

TABLE OF CONTENTS

2.4.2	Cache Replacement Policy	38
2.5	Memory Deduplication	39
2.6	Current Solutions and Mitigations in Cloud	41
2.7	Hardware Performance Counter (HPC)	47
2.8	Machine Learning Classification Algorithms	48
2.8.1	Logistic Regression	48
2.8.2	Softmax Regression	49
2.9	Summary	50
3	Mitigation through Memory Deduplication	51
3.1	Introduction	52
3.2	Protection Method	54
3.3	Experiments	58
3.3.1	Threat Model	59
3.3.2	Experimental Results	60
3.4	Evaluation and Comparison to Other Solutions	64
3.5	Limitations	71
3.6	Summary	71
4	Mitigation through Dynamic and Static Analysis	73
4.1	Introduction	74
4.2	Required Tools	75
4.2.1	Libguestfs library	76
4.2.2	AVML Tools and Volatility Framework	76
4.3	Method	77
4.4	Experimental Results	80
4.4.1	Monitoring Suspicious Behaviours	82
4.4.2	Static Analysis for VMs	85
4.4.3	Expansion of the Static Analysis for VMs	89
4.5	Evaluation	91

4.6	Limitations	96
4.7	Summary	97
5	Mitigation through Periodical Long-range Intervals Scan	99
5.1	Introduction	100
5.2	Method	101
5.3	Experimental Setup	102
5.4	Experimental Results and Evaluation	104
5.4.1	Microarchitectural Attacks Scan	105
5.4.2	The Solution Works in Parallel with Antivirus applications	107
5.5	Summary	110
6	Holistic Protection Solution	113
6.1	Introduction	114
6.2	Methodology	115
6.3	Experimental Setup	119
6.4	Experimental Results and Evaluation	120
6.5	Summary	124
7	Conclusion and Future Work	127
7.1	Conclusion	128
7.2	Contribution to knowledge	133
7.2.1	<i>Mitigation cache side-channel attacks through memory deduplication:</i>	133
7.2.2	<i>Mitigation through Dynamic and Static Analysis:</i>	134
7.2.3	<i>Mitigation through Periodically Long-range Intervals Scan:</i>	134
7.2.4	<i>Integrated Protection System:</i>	135
7.3	Limitations and Future Work	136
	Bibliography	141

LIST OF TABLES

TABLE	Page
2.1 Cache Attacks Implementations[1]	21
2.2 Microarchitectural Attacks Characteristics	30
2.3 Summary of Current Solutions and Mitigations in Cloud	45
3.1 Comparison to other approaches	70
4.1 Score-based threat classification	79
4.2 Intel Hardware Performance Events [2, 3]	84
4.3 Experiment results	95
4.4 Comparison to the Previous Works	96
5.1 Attacks Scan Experiment Results	106
5.2 Viruses Scan Experiment Results	107
6.1 System Overhead	124

LIST OF FIGURES

FIGURE	Page
2.1 Flush+Reload Cache Side-Channel Attacks	19
2.2 Prime+Probe Attacks	22
2.3 Flush+Reload Attacks	23
2.4 Flush+Flush Attacks	24
2.5 Attack Characteristics snippet from [4]	29
2.6 The AES Algorithm Flow Chart	31
2.7 Virtualisation Architecture	34
2.8 Direct Mapping	36
2.9 Associative Mapping	37
2.10 Set-Associated Mapping	37
2.11 Memory Deduplication Feature	40
2.12 Sigmoid Function	49
3.1 Cache Hit and Miss.	54
3.2 Normal Distribution of Cache Hit and Miss	55
3.3 Flush-based Attacks Exploiting Shared LLC.	56
3.4 Number of flushes per 5s for different attack scenarios	63
3.5 Number of Flushes per 5s for Normal and Attack scenarios with Noise	64
3.6 Number of flushes per 5s for attacks on multiple Application functions	67
3.7 Number of flushes per 5s for attacks on the GnuPG implementation of RSA	68
3.8 Obfuscation of the Attack Results	69

LIST OF FIGURES

4.1	Detection Stages	81
4.2	Experiment scenarios of all cache side-channel attacks in two cases without noise attack and with background noise. PP represents prime+probe, FF represents Flush+Flush, and FR represents Flush+Reload.	85
4.3	Number of cache misses per 15s for different attack scenarios.	86
4.4	Neural Network Model	89
4.5	Expanded Neural Network Model	90
4.6	Neural Network Classification of Benign VM	91
4.7	Neural Network Classification of Malicious VM	91
4.8	Validation Loss and Accuracy of the Neural Network Classification Model	92
4.9	Logistic Regression False Positive and Negative	93
4.10	Validation and Training loss	94
5.1	The Proposed Method	103
5.2	False Positive and Negative	105
5.3	The Roc Curve	106
5.4	Fast Scan for Attacks	108
5.5	Full Scan for Attacks	108
5.6	Scan Duration using ClamAV	110
6.1	Flush-based Attacks Detection Method	117
6.2	The Dynamic and Static Analysis Protection Method	118
6.3	The Periodically Cleansing Method	119
6.4	Systems Overhead	122
6.5	Normal Distribution of Systems Overhead	123

CHAPTER

1

INTRODUCTION

The introduction chapter provides an overview of the thesis, which proposes methods to protect shared virtualised systems against microarchitecture attacks, specifically cache side-channel attacks. This chapter defines the scope of the project and the motivations that explain the importance of these proposed solutions. It also describes the problem that the project seeks to solve by mitigating the threats of cache side-channel attacks, presents research questions related to the problem and how to design defence systems that operate at various levels to monitor suspicious activities and ensure the integrity of VMs and thus the integrity of shared virtualised systems in general, and presents the hypotheses that have been proposed and relied upon to prove their effectiveness in experiments.

1.1 Project Area and Motivation

One of the most widely used technologies nowadays is cloud computing technologies that combine computing resources and services provided to customers via the Internet. In accordance with the National Institute of Standards and Technology (NIST): *"The Cloud Computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction"* [5].

Cloud computing technology brings significant benefits for businesses and organizations, such as cost efficiency, scalability, and flexibility [6]. According to a Gartner analysis [7] which considered that cloud computing is among the top 10 most essential and promising aspects of technologies [8]. Cloud computing facilitates resource sharing by utilising multi-tenancy technologies to distribute computing resources owned by a third-party provider [9]. Furthermore, Cloud computing enables the provisioning and deployment of critical services, such as social media and business applications, with minimal administration effort [10].

Multi-tenancy is a critical characteristic of cloud. It enables cloud providers to maximise resource consumption by distributing a shared virtualised infrastructure across multiple users, thus reducing cost [6]. Cloud computing providers utilise automatic resource allocation mechanisms, which result in the creation of two or more VMs associated with distinct clients using the same physical machine's resources [11]. However, at times, the resources cloud is shared with a malicious user who exploits the allocation techniques or VMs placement policy to co-locate their VM with the specific target VM. In such a case, this could lead to confidentiality violation by executing co-resident attacks [12].

Despite all the benefits of multi-tenancy, it is still a source of new risks in cloud computing [13], [14]. Without appropriate security solutions designed for clouds, security issues could become the primary concern hindering adoption [6]. Moreover, virtualization that enables multi-tenancy, considered the main component of a cloud, creates significant security vulnerabilities and does not provide adequate isolation between multiple instances operating on the same host. The NIST report [13, 15] considered that the multi-tenancy is one of the most serious shortcomings and source of threats to cloud security and privacy. Likewise, several researchers [16–18] considered the multi-tenancy as a security vulnerability confronting cloud services vendors.

Also, the shared virtualised environment may be exploited to launch attacks on the shared cache that contains recently accessed data. These types of attacks expose the victim's sensitive information. Knowledge of such sensitive data leads to cracking encryption keys in almost all used encryption libraries such as AES and RSA which are well known encryption systems. The use of the shared virtualised environment is a substantial risk to the data unless there is a protection mechanism that monitors the activities within this environment periodically while maintaining the non-deterioration of the performance of the system as a whole, identifying suspicious activities and eliminating any attacking factor that may threaten the shared environments.

Although there are many proposed methods to mitigate the threats of this sort of an attack, these methods have some limitations in how a mechanism to detect or protect the

data works, and also what will steps will be taken after detecting a potential suspicious behaviour, the accuracy of detection of threat detection, and who is responsible for detection and protection operations. We also believe that data protection is the responsibility of everyone who works within the shared virtualised environments. Therefore there is a need to find integrated solutions that are reliable with high accuracy and acceptable performance and shared by everyone in order for the result to be an environment with the least possible threats that may potentially influence the security and privacy of all users of the shared virtualised environments. This research concentrates on a particular type of threat, namely the cache side-channel attacks that target the virtualisation level. In this kind of attack, the attackers have specific target VMs to extract sensitive information by exploiting various side channels (Breaking cryptography by using information leaked from physical parameters such as execution time and electromagnetic emission [19][12]. Moreover, the work presented in this thesis will introduce compatible solutions and countermeasures to detect suspicious behaviour that could indicate a cache side-channel attacks and develop security controls that maintain the advantages of multi-tenancy and sharing resources while reducing the security risks.

This thesis presents integrated solutions to provide the necessary protection for VMs that operate in a shared virtual environment in which VMs share a common computing power. We addressed the security problems from the user's and the service provider's point of view because sharing the responsibility of resources protection is everyone's responsibility. Therefore, we provided a mechanism that works inside the VM for detecting suspicious behaviour that may indicate the presence of an attack, and mechanisms that work inside the host for detecting the attacks and protecting the shared virtualised environment against malicious activities and attacks' programs such as Cache side-channel attacks with precautions being taken while there has been an indication of an attack on the shared resource. Also, a mechanism based on static analysis has been proposed to check VMs on a long-term periodic basis integrated with a well-known antivirus called ClamAV to scan VM against Microarchitectural Attacks and other threats that may be caused by viruses.

1.2 Problem Statement

Cloud computing relies on sharing computing resources over a network to reduce the cost of infrastructure. One form of sharing involves using a common pool of applications and programs that could be sensitive. Such as sharing cryptographic libraries using memory deduplication, which is a memory-saving feature to optimise memory utilisation and allows to increase the number of VMs on the same host. Despite the economical benefits of sharing computing resources, it is known to give rise to security risks if the resources are shared with a malicious user. In such a case, the malicious user can exploit the shared resources on the same physical machine as a covert channel to launch side-channel attacks. In some cases, such side-channel attacks are known to be able to crack most encryption algorithms leading to confidentiality violations [4, 12, 20–22].

There are a number of countermeasures [23–30] designed to mitigate cache side-channel attacks. Due to the cache attacks relying on sharing cache levels and time difference of accessing data from cache and main memory, most of the existing defence methods are proposed based on the following ideas: eliminating imbalance, partitioning caches, avoiding co-location, and detecting malicious activities. However, they are also known to have some shortcomings that will be discussed next. When applied to cloud computing, it requires significant changes to the computing infrastructure. This may hinder their adoption by cloud providers. Also, some of these methods may cause system performance degradation and high overhead. They also have high false rates (either false positive or negative) in detecting malicious activities. They also lack the diversity and comprehensiveness of protection against cache side-channel attacks. Moreover, they do not provide systematic procedures to exclude the malicious VM after detecting the cache attacks.

Therefore, it is necessary to design a protection system that integrates diverse lines of defence with acceptable performance and high accuracy, making it difficult to be penetrated and bypassed by attackers. While achieving all this the protection system also needs to maintain the fundamental nature of shared virtualised systems and improve security controls that enhance performance characteristics. It is also essential

to conserve the economic advantages of the shared virtualised systems while reducing side-channel attacks and microarchitectural attacks risks and providing mechanisms to monitor VMs' activities on the shared cache and design forensics workstation compatible with shared virtualised systems to analyse executable files of the suspicious VM to exclude malicious VM.

1.3 Research Questions and Objectives

In the context of providing shared virtualised systems and cloud computing systems with adequate integrated protection, this work studies and analyse effects, sources and reasons of the cache side-channel attacks and designing integrated detection and protection mechanisms that aim to provide the necessary protection for shared systems. Along these lines, this thesis seeks to answer the following research questions:

- *Question 1:* Is it possible to take advantage of memory deduplication as a protective factor to detect suspicious behavior in virtualized systems and obfuscate the attacks' results?
- *Question 2:* How can self-protection be provided to a VM for detecting suspicious activities and protecting its applications against cache side-channel attacks while there are limitations in using hardware performance counters inside the VM?
- *Question 3:* How to design a hybrid system that integrates dynamic and static analysis to detect and protect the shared virtualised system and identify malicious VMs with acceptable performance and high accuracy?
- *Question 4:* How to improve a protection system based on static analysis to be compatible with the nature of the shared virtualised system for scanning executable files against the cache side-channel attacks?

- *Question 5:* How to generalise the static analysis to scan microarchitectural attack opcodes integrated with an antivirus application for periodically scanning VMs within the shared virtualised system to prevent malware?
- *Question 6:* How to integrate the proposed solutions to design an integrated system to protect the shared virtualised system from malware and monitor suspicious activities?

Several experiments have been accomplished in the subsequent chapters to address the research questions. Questions 1 and 2 have been discussed in Chapter 3. Questions 3 and 4 have been addressed in Chapter 4, while Chapter 5 has addressed Question 5, and Chapter 6 has answered the research Question 6.

To address these research questions, the project seeks to fulfill the following objectives:

- *Objective 1:* To develop security controls that maintain the advantages of multi-tenancy while reducing the security risks due to side-channel attacks with acceptable degradation in performance, hence increase the difficulty for attackers to extract sensitive information.
- *Objective 2:* To detect abnormal behavior in the virtualised environment that could indicate cache side-channel attacks.
- *Objective 3:* To develop a comprehensive protection system to protect cloud computing against a sufficient number of microarchitectural attacks with various approaches and high accuracy.
- *Objective 4:* To provide self-protection for VMs to protect their shared application and cryptographic libraries within the shared virtualized system without fundamental changes to the system infrastructure.

The research questions and objectives will also be reviewed in the conclusion, in Chapter 7.

1.4 Thesis Hypothesis

In this section, we illustrate a number of possible hypotheses to solve the problem causing the attack.

- *H1* - It is possible to sense instability of the shared cache due to the attacks instructions on these cache lines by accessing the addresses of the sensitive functions of the cryptographic libraries in the shared cache and monitor the activities by measuring the access time of these functions.
- *H2* - Disabling memory deduplication may reduce the attack's success rate, and hence it can be used as a precaution if suspicious activities are detected.
- *H3* - Monitoring the attacker's activities and their impact on the hardware performance counters using the Linux Perf and then analysing them using machine learning classifiers, then it will be possible to detect the attack state in the virtualized system with high accuracy because the attack operations usually have a significant impact on some performance counters during the execution of the attack.
- *H4* - Analyzing the executable files of a particular VM and find the implicit characteristics of cache side-channel attacks will help identify the threat level of the suspicious VM using a softmax classifier because the attack files contain a series of instructions and opcodes that threaten the shared system.
- *H5* - Combining the H3 and H4 will reduce the system overhead and increase the detection accuracy because the H3 will indicate which suspicious VM needs to be analyzed with H4 rather than checking all the VMs in the shared virtualized system.
- *H6* - Generalizing the hypothesis H4 to other microarchitectural attacks and analyzing their implicit attributes using a logistic regression algorithm combined with an antivirus application, the protection against malicious programs will be more

comprehensive because not all antivirus applications can detect microarchitectural attacks related to the shared virtualised system.

- *H7* - If we combine the H1, H5 and H6 to operate together within the shared virtualised system, the protection system will become more comprehensive, which will positively affect the accuracy of detecting microarchitectural attacks due to the diversity of detection methods, as well as the system overhead will be acceptable because the protection system will rely on dynamic analysis continuously and restrict the use of static analysis instead of complete reliance on static analysis that consumes high system overhead.

In the subsequent chapters, several experiments will be conducted to examine the hypotheses. In Chapter 3, Hypothesis H1 will be examined. Hypothesis H2 and H5 will be discussed in Chapter 4, while Hypothesis H6 will be tested in Chapter 5, while we will discuss Hypothesis H7 in Chapter 6.

1.5 Thesis Contributions

This thesis provides new approaches for protecting shared virtualized systems against microarchitectural attacks and its contributions are outlined as follows:

- We proposed a method for protection against cache side-channel attacks by using memory deduplication and logistic regression from within the victim VM to detect suspicious activities and obfuscate the results of the attacks (Chapter 3).
- We designed a method for detecting and protecting shared virtualised systems against cache side-channel attacks by integrating a dynamic and static analysis and identifying the threat level of a particular VM using machine learning algorithms (Chapter 4).
- We developed a method for periodically cleansing shared virtualised systems against microarchitectural attacks and viruses by analyzing implicit attributes

of executable files by using a logistic regression algorithm integrated with a well-known antivirus application called ClamAV (Chapter 5).

- We designed a method that combines the above contributions to provide comprehensive and integrated protection for shared virtual systems against threats exposed to them with high accuracy and acceptable overhead on the system (Chapter 6).

1.6 Thesis Structure

The remainder of the thesis is organized as follows:

- *Chapter 2 – Literature review* – provides a review of concepts and principles related to cloud computing security and threats to cloud computing and illustrates the current countermeasures and related shortcomings.
- *Chapter 3 – Mitigation through Memory Deduplication* – presents a method for monitoring memory locations of shared sensitive applications such as cryptographic libraries using memory deduplication to fetch readings of activities on these memory locations and analyze them using a logistic regression algorithm to identify abnormal activities that indicate the state of cache side-channel attacks.
- *Chapter 4 – Mitigation through Dynamic and Static Analysis* – offers a mechanism for detecting the activities of VMs within the shared virtual system using Linux Perf to fetch the hardware performance counters readings and then analyze them using logistic regression to detect suspicious behavior. Upon detection of a suspicious VM, the VM's executable files are analyzed against the implicit characteristics of the cache side-channel attacks. Then the threat level is determined using the Softmax neural network algorithm.
- *Chapter 5 – Mitigation through Periodically Long-range Intervals Scan* – provides a technique to protect shared virtualized systems by periodically scanning VMs'

executable files against implicit attributes and opcodes of microarchitectural attacks and analyzing them using a logistic regression algorithm. Also, this method is integrated with the ClamAV antivirus application to provide comprehensive protection for shared virtualized systems.

- *Chapter 6 – Integrated Protection System* – proposes an integrated and comprehensive method that integrates all the proposed methods to provide various lines of defense with different techniques to protect the shared virtualised systems, making the attack operations more difficult to penetrate these lines of defense.
- *Chapter 7 – Conclusion* – highlights the thesis conclusions, summarises the project’s results to assess how effectively the objectives have been met, and discusses areas for future research.

CHAPTER



LITERATURE REVIEW

This chapter discusses essential topics, including concepts and principles related to the research problem; It provides overviews of the benefits and challenges of multi-tenancy systems. It also presents a study on the microarchitectural attacks that cloud computing is exposed to and possible solutions with an explanation of their limitations.

2.1 Cloud Computing Security

Cloud computing is adopted by companies, individuals, and governments to save cost, to increase efficiency, and to obtain other advantages in their business environment [31]. Despite the promising benefits of cloud computing, there are barriers that may limit its adoption in all areas, especially when it comes to information security within cloud. Adopters of cloud computing are plagued and concerned by the issue of security [32–35], as the cloud may be exposed to many security risks through the Internet or through co-residence that may expose it to more threats of malicious attacks. Concern about security in cloud computing is considered the main obstacle for the continuing growth of cloud computing.

Cloud computing is a shared system where many users share the same computing resources. Cloud computing's multi-tenancy concept and resource sharing have created new security concerns and impacts on information security [36, 37]. Cloud computing's multi-tenancy makes it possible for malicious users to run a VM and share resources with a victim VM on the same physical machine, making the malicious and victim VMs sharing resources on the same host, leading to a break in the logical isolation provided by virtualisation and a breach in confidentiality or degradation of the performance of the victim VM by launching side-channel attacks [38, 39].

In cloud computing, there are many different forms of side-channel attacks categorised based on the target hardware and how they work, for example, side attacks on the cache memory. Hence, sharing the same physical resources may facilitate side-channel attacks performed using a shared channel (covert channel) between an attacker and a victim

VMs. These vulnerabilities may lead to inadequate isolation between VMs and thus steal confidential and sensitive information from users.

Cloud computing provides clients with a management interface via which they can launch and terminate VM instances based on a configuration given by the client. To launch a VM, a client specifies a set of parameters, which are then sent to the provider's VM launch service by the client. Before deploying VMs, clients often create a cloud account and configure VM parameters such as the type of instance, disk image, or location. Then the VM launch service allocates resources for the VM and selects a specific physical host to run the new VM; this process is called VM placement. Certain factors influence the VM placement, for example, the available machines that can be used in the data centre, concurrent VM launch requests and time. By controlling these factors, an attacker is able to affect the placement of VMs on specific physical machine to locate the malicious VM with target VMs. Also, placement policy behaviour can be observable and exploited to increase the likelihood of attackers achieving co-residency[39].

The multi-tenancy in cloud computing involves multiple users, who may include the attacker and the victim, sharing the same computing resources. The side-channel attacks we consider require two main steps: first, placement of a malicious and a target VMs on the same server (co-located with the target VM), and second, extraction of information using a covert channel [40, 41].

According to Ristenpart et al. [38], an attacker should follow certain procedures and phases in order to conduct a co-resident attack and gather confidential information from the victim VM. In the first phase, the attacker creates an account with a cloud provider, then the attacker collects information about VM placement policy and discovers the cloud cartography using network probing. Next, the attacker abuses or brute forces the VM placement policy and determines whether two instances (attacker VM and target VM) are co-resident using a network-based co-resident check. After achieving multi-tenancy, In the third phase, the attacker recovers the sensitive information using a side-channel attack.

Cloud computing must be secure and adhere to strict requirements for data confidentiality, integrity, and availability. Cloud computing adopts the multi-tenancy feature to increase resource utilisation, improve performance, and reduce cost. On the other hand, multi-tenancy and virtualisation are considered the main challenges in cloud computing. Cloud computing systems produce several benefits; however, some organisations are still hesitant about shifting their setups to a cloud, mainly because of security issues and risks [42–44].

2.2 Side-channel Attacks

According to Shahid et al. [45] *"side-channel attacks are the physical attacks that use the physical process to extract the secret information of the cryptographic algorithms such as encryption key."* These attacks exploit the data leakage from a secret channel during execution processes. According to AlJahdali et al. [42] *"A side channel attack is any attack based on information gained from the physical implementation of a system. There are many side channel attacks known in the field; some of the well-known side channel attacks are timing attacks, power consumption attacks and differential fault analysis."*

Side-channel attacks are evolving in attacking computing devices until they reached cloud computing platforms, and they are also applicable and work effectively in cracking many encryption applications and other applications to identify the behaviour of the victim [46]. Side-channel attacks are not new, whereas their impact is increasing in cloud computing due to the sharing of resources. In side-channel attacks, the attacker exploits the shared hardware resources as a covert channel for recovering confidential information, such as the cryptographic key or any other sensitive information regarding the victim. The data that is leaked may come in various forms, for example, power leakage, electromagnetic leakage, and timing leakage.

In the timing attack, the attacker measures the timing of executing sensitive operations

to break encryption keys or obtain sensitive information about the victim's behaviour. This type of attack includes many scenarios and methods for carrying out these attacks. The side-channel attacks have successfully broken almost all the cryptographic algorithms today. They are a severe threat to the cloud systems [12, 47].

2.2.1 Cache Side-channel Attack

In this work, we focus on cache attacks that target the shared cache memory between users in virtualized systems, where the attacker analyzes the timing information gained from retrieving data from the cache or from the main memory [45]. When the CPU looks for data, it can be found in the main memory or in the cache. If the data is recovered from the cache, the retrieval time or amount of CPU cycles is low. However, suppose the data is not cached in memory. In that case, it must be recovered from the main memory, ensuring a relatively larger amount of time and CPU cycles would be taken to recover it. Then the recovered data will temporarily remain in the cache memory to improve the system performance if the data is retrieved next time. Consequently, the attack process depends on exploiting the time difference between recovering data either from cache or the main memory; in other words, the time difference between cache hits and cache misses[20].

The attackers take advantage of timing information to launch attacks on the victim's VM using the cache hits and cache misses to measure the CPU cycles or the time to recover the cache memory's targeted addresses. In this attack, the attacker can break the isolation between VMs [22], uncover the victim's actions, obtain information about cryptographic operations, and then break the encryption key.

The attackers can obtain sensitive information from encryption processes using timing information extracted from the shared cache memory. This may lead to breaking encryption systems, e.g., by using timing information in the TableLookUp implementation of AES. For making the encryption process easy and fast in AES, the T-Table implementation was designed in addition to the XORing process (TableLookUp operation). However, the

T-table entries will be stored in the cache memory when used in encryption processes, which leads to ease of breaking the encryption key using timing information[20, 48].

The main ways to exploit cache memory and extract sensitive data are given in section 2.2.1.1, section 2.2.1.2, and section 2.2.1.3.

The attacker takes several steps to execute cache side-channel attacks, taking advantage of shared resources and memory deduplication, as shown in Figure 2.1. The attack is carried out with the following steps: (1) In the beginning, the victim can use the shared program that contains a number of sensitive operations and functions that are loaded into the shared cache by simply entering one of them and executing one of the functions. (2) The attacker evicts these physical addresses from the cache memory by using the flush command (*clflush*) to ensure that the addresses will be retrieved from the main memory if requested next time, as a trap for the victim to find out the data retrieved and stored in the cache memory, if the victim used one of these addresses. (3) The victim may use one or more of the sensitive program's functions, and as soon as the victim uses one of them, it will be restored to the cache memory. That means that the victim has actually fallen into the trap set by the attacker. (4) The attacker retrieves all the addresses that have been flushed while keeping track of how long it takes to retrieve each of these addresses using Time Stamp Counter (*rdtsc*). (5) The attacker analyzes the results. If the retrieval time for any of the physical addresses is longer than the specified threshold, this means that none of them was used. However, if the retrieval time for any of them was less than the threshold, this means that it was used in the sensitive operations.

Data leakage is the result of sharing the same physical machine. Prior studies have shown the possibility and practicality of cache side-channel attacks in cloud computing. For instance, in the case of a cross-VM attack on AES implementations, Irazoqui et al.[20] found that the attack exploits the transparent Page Sharing used in virtualization environments. Suzuki et al.[49] employed memory deduplication to detect processes running on the target VM. Bernstein's attack is a cache attack that is implemented in a client-server-based environment that is non-virtualized[50]. Bernstein used a client

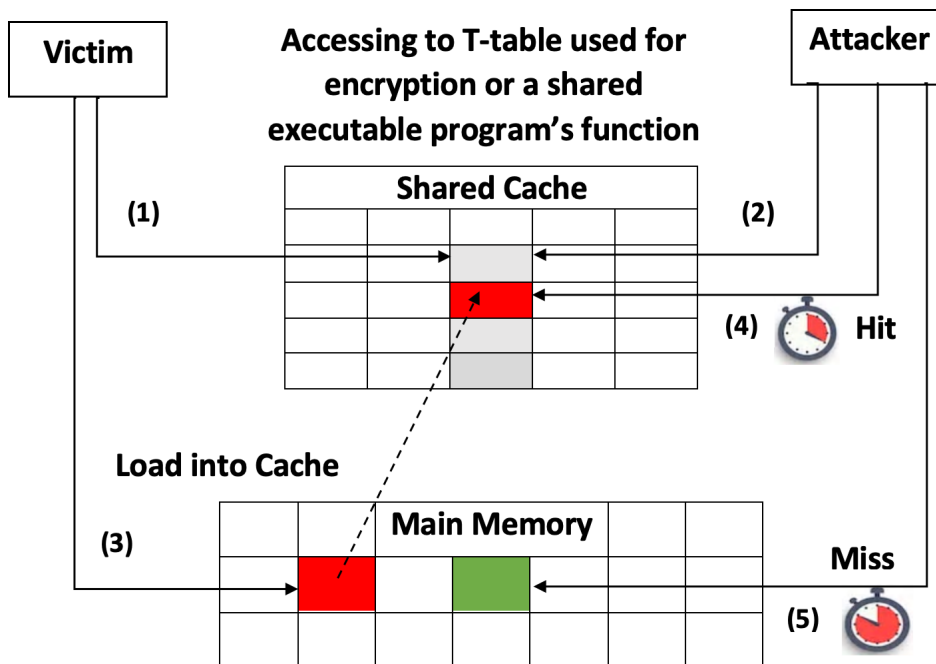


Figure 2.1: Flush+Reload Cache Side-Channel Attacks

to send UDP packets to request encryption from the server, and then the server sends back the encrypted text to the client. Bernstein was able to crack the AES encryption key using the timing information. Also, Irazoqui et al.[51] applied Bernstein's attack on OpenSSL 1.0.1 using a virtualized environment with Ubuntu 12.04 running XEN and VMware hypervisors. Irazoqui et al. implemented a cross-VM attack scenario, and were able to break an AES cryptographic key.

Multiple researchers have discussed the security issues related to cache attacks when certain information is leaked. For instance, in cache attacks on AES, Bernstein implemented a cache attack that targeted the TableLookup of OpenSSL implementation of AES, using timing information. Bernstein used two servers. One was the actual victim's server, and the other was a replica identical to the victim's server [50]. The attack was performed in two main stages: the first stage is the profiling and the second stage is the execution of the attack. In the first stage, the attacker sends a large number of packets to the identical server. The server encrypts these packets with an identified encryption key and provides the attacker with the encryption's timing information. After that, in

the second stage, a similar operation is executed again, but in this stage targeted the actual server itself using a private encryption key. Next, the timing profiles from these two stages are correlated to indicate the most likely value to be the encryption key used for the encryption process[50, 52, 53].

Yarom et al. presented the cache attack based on Flush + Reload technique to extract a secret encryption key from the GnuPG 1.4.13 implementation of RSA. The result of their cache attack was the breaking of the encryption key from an ECDSA (Elliptic Curve Digital Signature Algorithm)[4, 54]. Gullasch et al.[55] applied a cache timing attack on the OpenSSL 0.9.8n using Linux kernel 2.6.23. The attack was executed using the Flush + Reload technique to find out the memory accesses' timing information. The key was broken with a few encryption attempts during the attack. Also, Irazoqui et al.[20] implemented a cache attack with different scenarios on AES by using the Flush + Reload attack to gain the AES cryptographic key in a virtualized environment. The attack needed to enable memory deduplication in VMware ESXI 5.5.0 with several Ubuntu 12.04 64-bit guest OSs.

This review was presented based on the information related to the main topic of the thesis, so through this review, it was identified what side-channel attacks are, their goals, how to implement them, and what is the appropriate environment for such attacks. These questions are fundamental to provide a comprehensive overview of these attacks and thus comprehend and understand the goal of the thesis. Also, Table 2.1 shows the number of cache attacks, the systems that were targeted to carry out these attacks, and the exploit points for each system that was targeted. The table shows that these attacks threaten many cryptographic systems, indicating the diversity of their implementations.

Table 2.1: Cache Attacks Implementations[1]

Attack	Target	Experimental system	Exploited Features
[55]	AES(OpenSSL 0.9.8n)	Pentium M, Linux 2.6.33.4	Shared library, completely fair scheduler in Linux
[4]	RSA(GnuPG 1.4.13)	Intel Core i5-3470 (Ivy Bridge), Intel Xeon E5-2430, VMware ESXi5.1, KVM	Memory mapping or page deduplication
[20]	AES (OpenSSL 1.0.1f)	Intel i5-3320M, VMware ESXi5.5.0	Page deduplication
[54]	ECDSA(OpenSSL 1.0.1e)	Intel Core i5-3470	Memory mapping
[56]	TLS, DTLS (PolarSSL 1.3.6, CyaSSL 3.0.0, GnuTLS 3.2.0)	Intel i5-650, VMware ESXi5.5.0	Page deduplication
[57]	Keystroke,AES(OpenSSL 1.0.2)	Windows, Linux	Shared libraries
[58]	DSA in OpenSSL	Intel Haswell (Core i5-4570)	Shared libraries
[59]	ElGamal (GnuPG 1.4.13 and 1.4.18)	Xen 4.4 (Intel Xeon E5 2690), VMware ESXi 5.1 (Intel Core i5-3470)	Huge page
[22]	User behaviours	Ubuntu LTS v16.04.1, QEMU-KVM v2.6.2	Page deduplication
[21]	AES(OpenSSL)	Haswell i7-4790 CPU	Shared libraries

2.2.1.1 Prime+Probe

In this method, the attacker's VM loads the cache lines with its data. Next, it gives the victim some time to perform some encryption operations. After that, the previously loaded data retrieval time is measured by the attacker's VM. As a result, the attacker will know what data has been removed from the cache memory and, thus, recognize the cache lines used in the victim's encryption operations, as shown in Figure 2.2. This technique does not require shared libraries or page deduplication. Figure 2.2 shows the

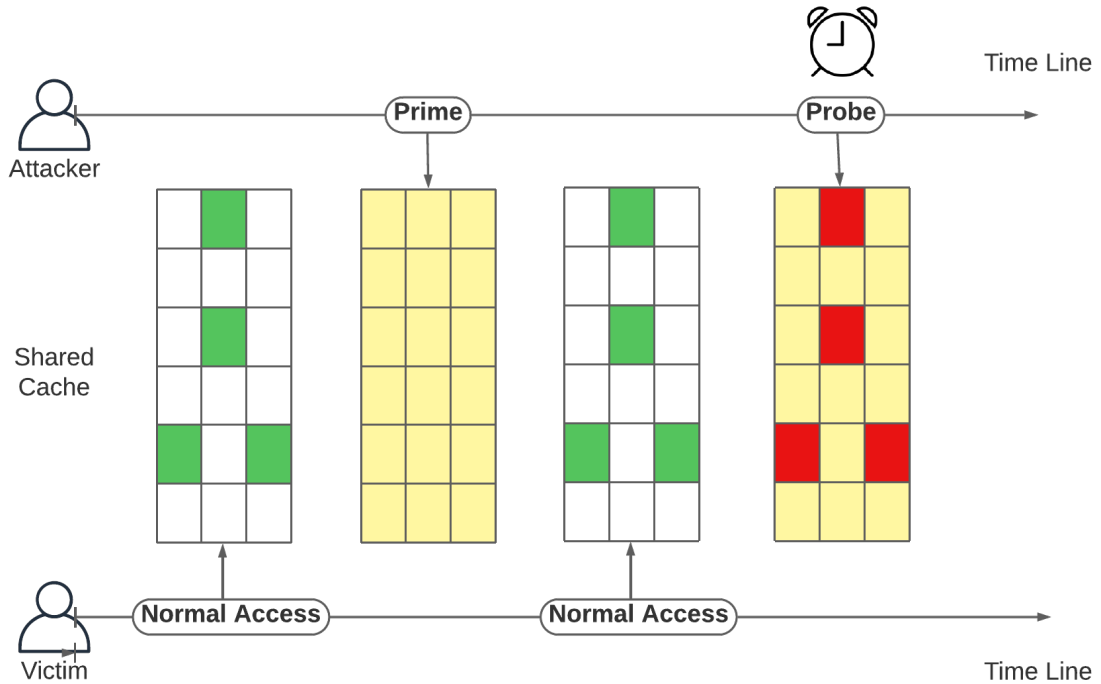


Figure 2.2: Prime+Probe Attacks

steps of the attacker using the prime+probe technique. At first, the attacker fills the shared cache memory with his own data (cache lines in yellow in the figure), then allows some time for the victim to carry out some sensitive operations (the victim used the green cache line in figure), which will be stored in the shared cache memory. After that, the attacker accesses the same data that was filled in the cache memory and, at the same time, measures the access time for each cache line. If it exceeds the specified threshold (marked in red in the figure), it means that the victim used this cache line in its sensitive operations.

2.2.1.2 Flush+Reload

The attacker's VM first flushes the required cache lines out of the cache, as shown in Figure 2.3. After that, it gives the victim time to perform some encryption operations. Next, the attacker reloads the evicted lines and measures their access time. Thus, the attacker can identify if the victim's encryption process has recovered the cache lines,

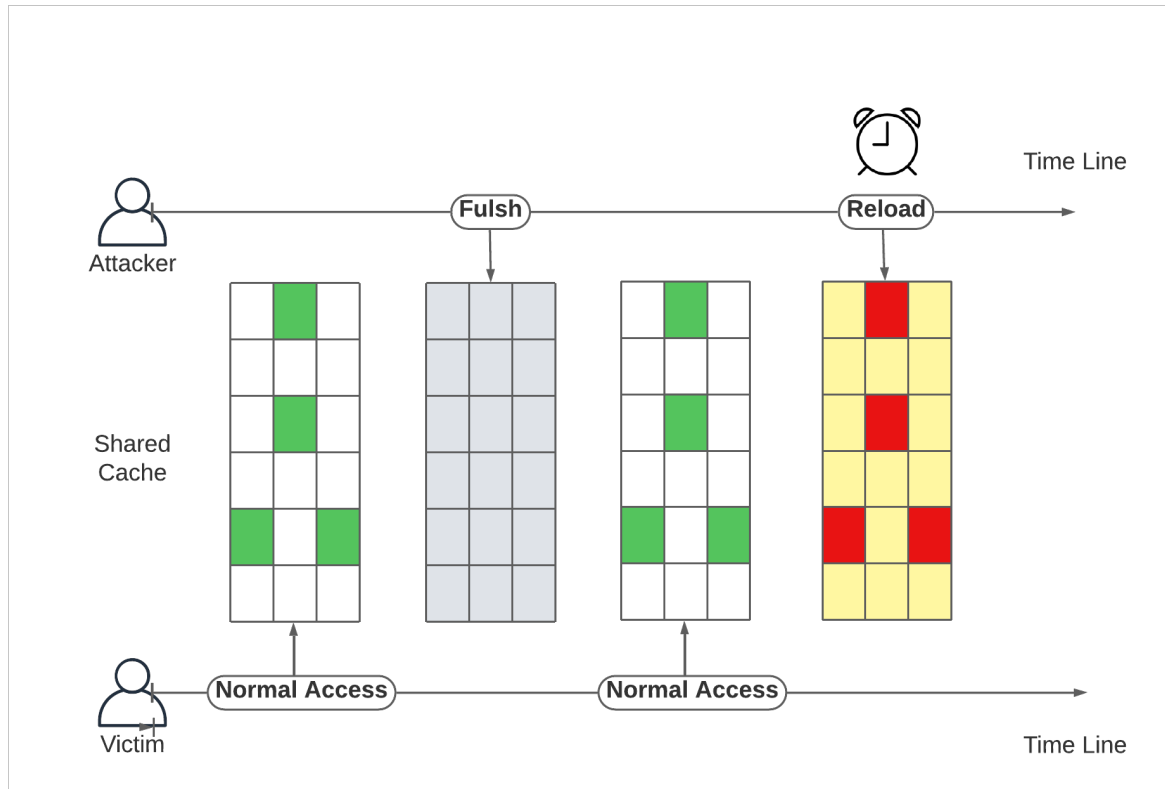


Figure 2.3: Flush+Reload Attacks

using the timing information. This technique relies on shared libraries and memory deduplication. Figure 2.3 represents the steps of the attacker using the flush+reload technique. In this attack, the attacker first flushes a specific cache lines (marked gray in the figure) out of the shared cache, then allows some time for the victim to perform sensitive operations (the victim used green cache lines in the figure), after which the attacker reloads the same cache line (marked yellow in the figure) that was flushed before. At the same time, the attacker measures the access time of the cache line. If the access time is shorter than the specified threshold, this means that the victim used the same cache line in the sensitive operations (marked in red in the figure).

2.2.1.3 Flush+Flush

In this approach, as shown in Figure 2.4, the attacker's VM first flushes the required memory lines out of the cache. After that, it gives the victim time to perform some

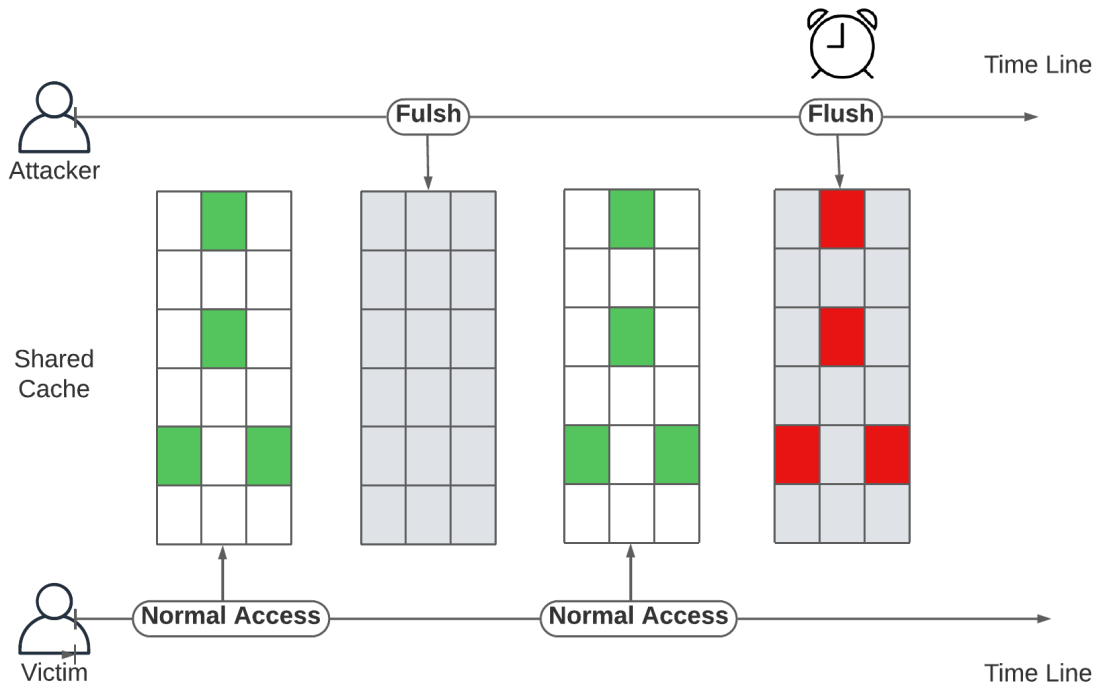


Figure 2.4: Flush+Flush Attacks

encryption operations. Next, the attacker's VM flushes again the previous memory lines and measures the flush instructions' execution time, bypassing direct cache accesses. The attacker can use this type of attack to break a cryptographic key. This technique relies on shared libraries and memory deduplication. Figure 2.4 illustrates the steps of the attacker using the flush+flush technique. In this attack, the attacker first flushes a specific cache lines out of the shared cache (marked gray in the figure), then allows some time for the victim to perform sensitive operations (the victim used green cache lines in the figure), after which the attacker flushes again the same cache line (this is the only difference from the previous flush+reload technique). At the same time, the attacker measures the access time of the cache line. If the access time is longer than the specified threshold, this means that the victim used the same cache line in the sensitive operations (marked in red in the figure).

2.2.2 Other Microarchitectural Attacks

This section reviews some other microarchitectural attacks that threaten the shared virtualised systems due to the nature and structure of these systems. These microarchitectural attacks share some properties, techniques, and attack environments with cache side-channel attacks.

2.2.2.1 Spectre Attack

The Specter Attack exploits an essential optimization technique adopted by modern CPUs called Speculative Execution. Speculative execution occurs when a CPU retrieves data that will most likely be required later, rather than waiting until it definitely requires it. The attacker can observe the data from region not allowed to be accessed in the memory that leads to reveal the victim's process. To perform the attack, first, the attacker performs flush instructions to evict the desired cache lines and the target branch instruction address. The attacker effect the CPU branch predictor many times using proper inputs for the conditional branch. After which, the attacker inputs an invalid value for the conditional branch to cause a wrong prediction, thereby loading sensitive data into the shared cache. Finally, the attacker observes and keeps track of accessing time of the cache lines. If certain cache lines have a short access time, the data is considered sensitive [60, 61].

2.2.2.2 Meltdown Attack

Meltdown attack is a microarchitectural attack that abuses speculative execution feature in modern CPUs to leak data that is stored in kernel memory. This is a very similar attack to the spectre attack, except that it does not rely on branch prediction, and aims to read the kernel memory from the userspace [60]. The attacker achieves the meltdown attack in the following manner. First, the attacker runs a code to read a byte (secret value) from privileged memory to implement a faulting instruction that will throw an exception of segmentation fault. Although it throws the segmentation fault exception, the the byte (secret value) is in the cache, after the attacker multiplies, the byte, the cache

page size, and use it as an index into the block of allocated memory. After which, the attacker iterates through and observes the time taken to read, thus revealing confidential data [62].

2.2.2.3 Rowhammer Attack

Rowhammer attack approach exploits the electrical interaction of the DRAM rows with each other causing them to leak part of the charge when continuously accessing adjacent rows. The attacker takes advantage of this point by repeatedly accessing a DRAM row until it causes the bit flips from one to zero or vice versa. The attacker takes the following steps to carry out this attack: first, the attacker selects a DRAM row that is next to the DRAM row that is to be flipped. After which, the attacker repeatedly accesses the DRAM row to affect the adjacent rows, thus leaking their charge. Finally, the attacker evicts the accessed DRAM out of the cache to guarantee that the subsequent access will be to the DRAM row [63].

2.2.3 Cache and Microarchitectural Attacks Characteristics

The implicit characteristics of microarchitectural side-channel attacks explain how these attacks have been designed and how they work. Figure 2.5 shows the characteristics of microarchitectural side-channel attacks (opcode). As described by Irazoqui et al.[64] [26], the code and programs of microarchitectural side-channel attacks contain implicit characteristics and instructions that may distinguish them to some extent which lead to revealing them when analyzing these attacks' codes. Table 2.2 shows a set of characteristics (opcodes) of microarchitectural attacks and their function, as well as the microarchitectural attacks that use each.

The attacker is likely to misuse unprivileged information and the legal use instructions to launch microarchitectural side-channel attacks. This information can inform the the attacker to design and program attack scripts by utilizing instructions that are able to, evict the cache memory, measure time for retrieving data precisely, locking the memory

bus, and bypassing cache access as well, as shown in Figure 2.5. All these scripts are then compiled into executable files within the shared virtualized environments to perform the attacks. However, identifying these scripts of attacks' is possible by disassembling the executable files of the attack and recognizing the interior implicit characteristics and instructions on how they have been built.

The microarchitectural side-channel attacks comprise of certain characteristics that require to be incorporated in their design. Below we review these characteristics as discussed in [64] [26].

- **High-Resolution Timers:** As shown in Figure 2.5 (line 7 and line 12), a set of microarchitectural side-channel attacks rely on timing information for retrieving data from the cache, the RAM, the last level of cache, and the first and second level cache precisely. Hence, it is required to use an instruction, such as Time Stamp Counter (*rdtsc*), that records the timing information efficiently and has adequate precision to distinguish between data retrieval times.
- **Memory Barriers:** As shown in Figure 2.5 (line 5, 6, 8, and 11), memory barriers includes two type of instructions, *mfence*, and *lfence*. The attacker may use these instructions to serialize all store and load activities that happened prior to *mfence* and *lfence* instructions in the program instruction stream. In other words, these instructions can be used to suspend out-of-order execution and collect precise timing information of retrieving data from the cache and RAM. *mfence* and *lfence* instructions can also be included in the attacks' scripts [65, 66].
- **Cache Evictions:** As shown in Figure 2.5 (line 14), the attacker is able exploit the eviction instructions to evict the required cache line out of the cache using *Cflush* instruction as a trap for the victim to retrieve the data from the RAM and waits for a while until the victim retrieves this cache line. Therefore, the attacker realizes that the evicted and flushed cache line has been used by measuring the data

retrieval time. The desired cache line is removed from the entire cache memory (all cache levels) using the *Cflush* instruction [67].

- **Memory Access Lock:** Attack scripts could also contain memory bus locking instructions to ensure that the processor has exclusive ownership of the shared memory for the execution duration. The bus locking instruction consists of the *Lock* prefix and the following instructions as *ADC*, *ADD*, *AND*, *BTC*, *BTR*, *BTS*, *CMPXCHG*, *DEC*, *FADDL*, *INC*, *NEG*, *NOT*, *OR*, *SBB*, *SUB*, *XADD*, *XOR*. However, The *XCHG* instruction does not require the *Lock* prefix [68].
- **Non-temporal Memory Access:** Non-temporal instructions allow the processor not to write data into the cache. Thus retrieving data from the memory directly when requested. These instructions include *movnti* and *movntdq* instructions [69, 70].
- **CPU Affinity Assignment:** Some microarchitectural attacks require CPU affinity to achieve the co-residency on the same CPU core to share a specific cache level with a target process. Therefore, the attacks scripts may have function calls that accomplish the CPU affinities, such as *sched_setaffinity*[26].
- **Instruction Iteration:** In some microarchitectural attacks, the attacker needs to repeat some of the instructions mentioned above to execute the attack successfully. Hence some of these instructions may be placed inside a *Loop*.
- **Mmap() Function:** Attackers using *mmap()* function to load the target program into memory with as a Read Only file, then memory deduplication feature scans and removes the replica files to be one copy shared between users; this is a critical requirement for the Flush+Reload and Flush+Flush attacks.

2.2.4 Cryptographic Systems Vulnerable to Cache Attacks

Many cryptographic algorithms have been exposed to side-channel attacks, for example. RSA, AES, Digital Signature Algorithm, Elliptic Curve Cryptography and ElGamal.


```

1 int probe(char *adrs) {
2     volatile unsigned long time;
3
4     asm __volatile__ (
5         " mfence          \n"
6         " lfence          \n"
7         " rdtsc           \n"
8         " lfence          \n"
9         " movl %%eax, %%esi \n"
10        " movl (%1), %%eax  \n"
11        " lfence          \n"
12        " rdtsc           \n"
13        " subl %%esi, %%eax  \n"
14        " clflush 0(%1)    \n"
15        : "=a" (time)
16        : "c" (adrs)
17        : "%esi", "%edx");
18    return time < threshold;
19 }

```

Figure 2.5: Attack Characteristics snippet from [4]

This section discusses the most important cryptographic algorithms, which are the most extensively employed in secure systems. The objective is to provide sufficient context to comprehend the side-channel attacks.

2.2.4.1 Advanced Encryption Standard (AES)

AES or advanced encryption standards is one of the most widely used block cipher to encrypt and decrypt sensitive information. There are three different forms to encrypt 128-bit blocks of data with a symmetric-key block cipher: with 128-bit key size, 10 rounds of encryption, or 192-bit key size, 12 rounds of encryption. With 256-bit key size, 14 rounds of encryption. AES is comprised of 4 main functions [71]; **SubBytes** for substituting the bytes according to a substitution table. **ShiftRows** for shifting rows of the state. **Mixcolumns** for mixing the columns of the state. **AddRoundKey** to XOR the state with the round key. Each round implements these 4 functions except the last round as shown in Figure 2.6, MixColumns function is not used. To speed up the encryption

Table 2.2: Microarchitectural Attacks Characteristics

Characteristics	Task	Attack(s)
<code>rdtsc</code>	Timing information	Prime+Probe, Flush+Reload, Flush+Flush, Specter, and Melt-down
<code>mfence</code> , and <code>lfence</code>	Instructions serialization	Prime+Probe, Flush+Reload, Flush+Flush, Specter, Melt-down, and Rowhammer
<code>clflush</code>	Cache line eviction	Flush+Reload, Flush+Flush, Specter, Meltdown, and Rowhammer
<code>lock prefix</code>	Memory access lock	Prime+Probe, Flush+Reload, Flush+Flush, Specter, Melt-down, and Rowhammer
<code>monvnti</code> , and <code>movntdq</code>	Preventing caching data	Rowhammer
<code>sched_setaffinity</code>	CPU affinity	Prime+Probe, Flush+Reload, and Flush+Flush
<code>loop</code>	Instruction iteration	Prime+Probe, Flush+Reload, Flush+Flush, Specter, Melt-down, and Rowhammer
<code>Mmap()</code>	Load file to the mwmory	Prime+Probe, Flush+Reload, and Flush+Flush

execution, the `SubBytes`, `ShiftRows` and `Mixcolumns` functions have been combined into 4 lookup tables (T-tables) in various AES implementation such as OpenSSL.

2.2.4.2 Rivest–Shamir–Adleman (RSA)

RSA is a well known asymmetric key algorithm; The encryption and decryption process is accomplished with two different keys; the encryption process is accomplished with a public key and the decryption process is accomplished with a private key in the RSA system. RSA encryption relies on factors and large prime numbers, and its operations are based on modular exponentiations. RSA starts its function by generating the keys; It picks two prime numbers that are huge and distinct p and q , then calculating $n = p \times q$ and *totient* function that is $\phi(n) = (p - 1) \times (q - 1)$. After that selecting a positive integer

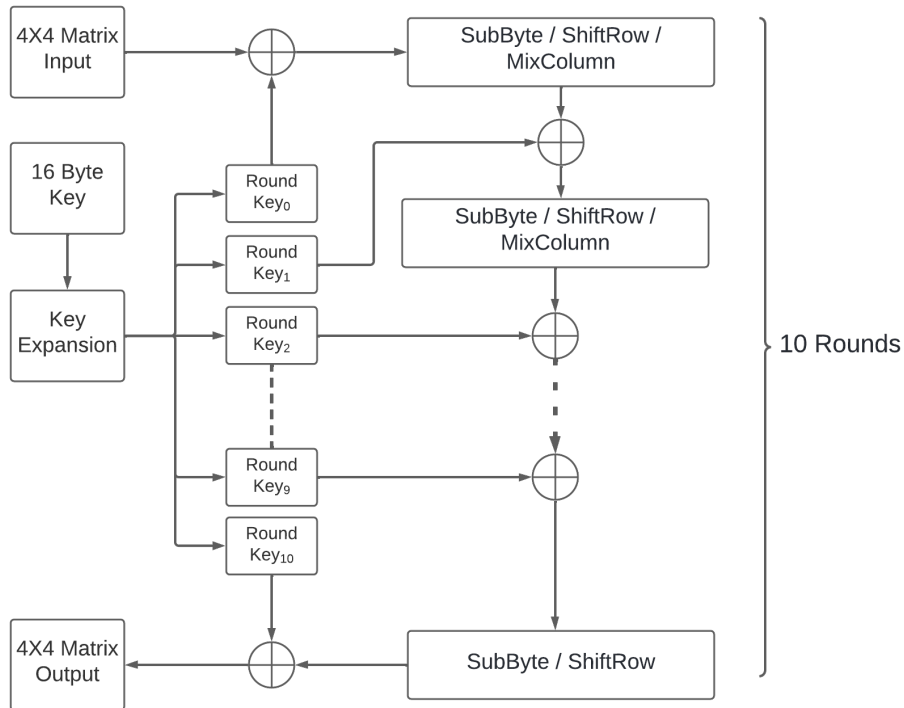


Figure 2.6: The AES Algorithm Flow Chart

e such that e is co-prime to $\phi(n)$ that means; $gcd(e, \phi(n)) = 1$ and $1 < e < \phi(n)$. The pair (n, e) make up the encryption key. Finally, calculating d that equals $d = e^{-1} \bmod \phi(n)$. The pair (n, d) make up the decryption key [71].

Plaintext P is encrypted with the encryption key (public key) by executing the $C = P^e \bmod n$ operation, and the decryption of the ciphertext C is calculated with the decryption key (private key) by performing $P = C^d \bmod n$. The square and multiply algorithm can be used to execute modular exponentiation. It can be used to process the key's bits and execute square and multiply operations regarding bits [71]. Multiplication is performed only when a key bit is one; otherwise, it is not, and a square is always performed, as shown in Algorithm 1.

2.2.4.3 ElGamal Encryption

ElGamal encryption is an asymmetric key cryptosystem, it uses private and public keys to communicate between two users to encrypt and decrypt the message between them [72,

Algorithm 1 Square and Multiply

Input: Ciphertext: C , Modulus: N and Private key: $K = (k_{m-1}, k_{m-2}, \dots, k_0)$ **Output:** Plaintext: $M = C^k \bmod N$

```
1:  $R = C$ 
2: for  $i = m - 2$  to  $0$  do
3:    $R = R^2 \bmod N$ 
4:   if  $k_i = 1$  then
5:      $R = R * C \bmod N$ 
6:   end if
7: end for
8: return  $R$ 
```

73]. ElGamal encryption has three steps: key generation, encryption and decryption:

Key generation: In order to create the encryption key, we must select a large prime number (p), and we also select the decryption key (d). After that, we choose a number for the second part (e_1) for the encryption key, so that the third and final part of the encryption key (e_2) is calculated by $e_2 = e_1^d \bmod p$ so the public key is the value of (e_1, e_2, p). **Encryption:** in encryption process we have to select an integer random number (r), then we calculate the first part of the ciphertext (c_1) that equals $c_1 = e_1^r \bmod p$. After that, we calculate the second part of the ciphertext (c_2) by $c_2 = (m * e_2^r) \bmod p$ (while (m) denotes to the plaintext), so the ciphertext is the value of (c_1, c_2). **Decryption:** first step in this stage is calculating k value that equals $k = c_1^d \bmod p$ and then we calculate the plaintext that equals $m = (c_2 * k^{-1}) \bmod p$, and then the plaintext is the value of (m).

2.2.4.4 Elliptic Curve Cryptography (ECC)

Elliptic Curve Cryptography (ECC) is an asymmetric key cryptosystem. it provides equal security with a smaller key size compared with other algorithms. An elliptic curve is a curve defined by $y^2 = x^3 + ax + b$ [74].

If we have $E_q(a, b)$, which is an elliptic curve with parameters (a), (b), and (q) that is a prime number, also we have G , which is a base point in the elliptic curve $E_q(a, b)$, we can use elliptic curve cryptography to encrypt and decrypt messages by the following steps:

Users Key generation : User A selects a private key n_A while $n_A < n$ (n is the curve limit) then user A calculates the public key p_A that equals $p_A = n_A * G$. User B also selects a private key n_B while $n_B < n$ (n is the curve limit) then user A calculates the public key p_B that equals $p_B = n_B * G$. Then, user A calculates the secret key using user B public key and vice versa for user B secret key $K = n_A * p_B$ And $K = n_B * p_A$. **Encryption :** When we encrypt a message (m), we should encode it into a point on elliptic curve p_m , and then we select a random positive number (k), so the cipher point will be $c_m = \{kG, p_m + kp_B\}$ this cipher point will be sent to the receiver. **Decryption :** To decrypt the cipher point c_m the receiver (user B) multiplies the x-coordinate (kG) with the receiver's secret key $kG * n_B$, then subtract it from y-coordinate $p_m + kp_B - kG * n_B$, then the receiver get the (p_m) point.

2.3 Multi-Tenancy and Virtualisation

The main goal of multi-tenancy is to share resources, which is the fundamental base of cloud computing. It allows many users to be served and saves costs, alongside other benefits such as, higher utilization, easier maintenance and deployment. For example, on Github, each user can share the main software application and codes, but each user's data is isolated from the others. Users can customise some features in their accounts, but unauthorised users cannot modify the core application and codes. However, multi-tenancy presents severe security threats and opens doors for possible privacy leaks[75]. Moreover, it presents the possibility of deploying a malicious VM on the same physical hardware, thus risking the extraction of secret information via side-channels[39]. Indeed, multi-tenancy could lead to a cloud security issue since it provides malicious parties with the right to access shared resources[76]. Different views exist on multi-tenancy in cloud computing; although some see it as an advantage, others see it as a risk that necessitates a solution[13]. Multi-tenancy is enabled by virtualisation that makes one physical machine act as many machines. Virtualisation uses a piece of software called Hypervisor (abstraction layer) or Virtual Machine Manager (VMM), which can create

a virtualisation layer and enable many operating systems for operating on one host concurrently, as shown in Figure 2.7. VMM observes the performance of guest operating systems that share virtualized hardware resources [77], such as VMware ESX. Thus, we can run many machines in one physical host with lower cost, higher performance and faster maintenance.

Virtualisation's primary goal is to manage workload by making computing resources more flexible, accessible, and cost-effective. Applying virtualisation can take place at both the software and hardware levels [78]. Although virtualisation has positive effects on a cloud computing environment, researchers [76] suggest eliminating the virtualisation layer to prevent multi-tenancy (which is considered a vulnerability and a source of security concerns) because virtualisation enables the running of many operating systems and applications concurrently on the same physical host [78], which an attacker could exploit to achieve co-resident attacks on the same server[76].

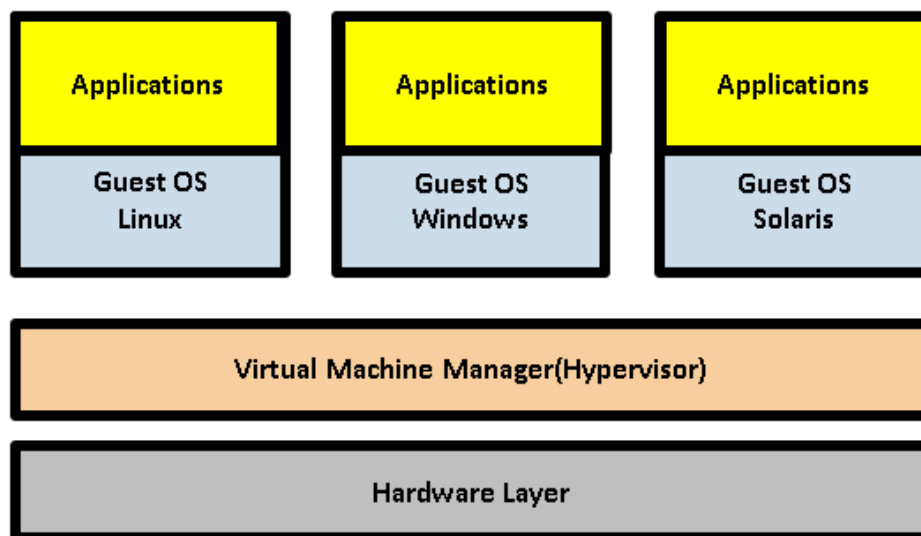


Figure 2.7: Virtualisation Architecture

2.4 Cache Architecture

The cache memory is a hierarchical collection of small-size and fast memory levels; some of these levels are reserved for a specific CPU core, while others, such as the last level

of cache, are accessible by entire CPU cores. cache is often situated between both the CPU and main memory. The primary objective of the cache memory idea is to accelerate data retrieval in the system, as obtaining data from the main memory takes longer than obtaining data from the cache memory[79].When the CPU requires data to be retrieved, it first checks all available cache memory levels. Assume the data was fetched from a cache memory location. This is referred to as a cache hit, and the retrieval time is quite faster in comparison to getting data from main memory. Retrieving data from the main memory instead of the cache is called a cache miss [20]. When data is retrieved from main memory, the CPU retrieves the entire block of data into the cache memory in order to take advantage of spatial locality for enhancing system performance.

The cache consists of three levels: L1, L2, and last level of cache (LLC). These levels of cache have different storage capacities and different characteristics as well. L1 and L2 are smaller storage capacity and faster than the LLC, and they are assigned for one core only in the CPU. In contrast, all the CPU cores share the LLC. The cache memory is split into blocks of the same size. These blocks are organized into sets with the same index[23].

2.4.1 Cache Addressing

Cache addressing or cache mapping refers to the techniques used for mapping a memory block from the main memory into the cache memory. There are three basic types of cache mapping, as outlined below.

2.4.1.1 Direct Mapping

In this technique, the main memory is divided into pages that are equal in size to the cache frame. Each location in the cache can store one copy of the main memory page, as shown in Figure 2.8. In other words, it maps a particular block of main memory into only one possible line of the cache. For example, if we have cache memory split into equal-sized 128 cache lines, and we have block 8 of main memory, then we can map block

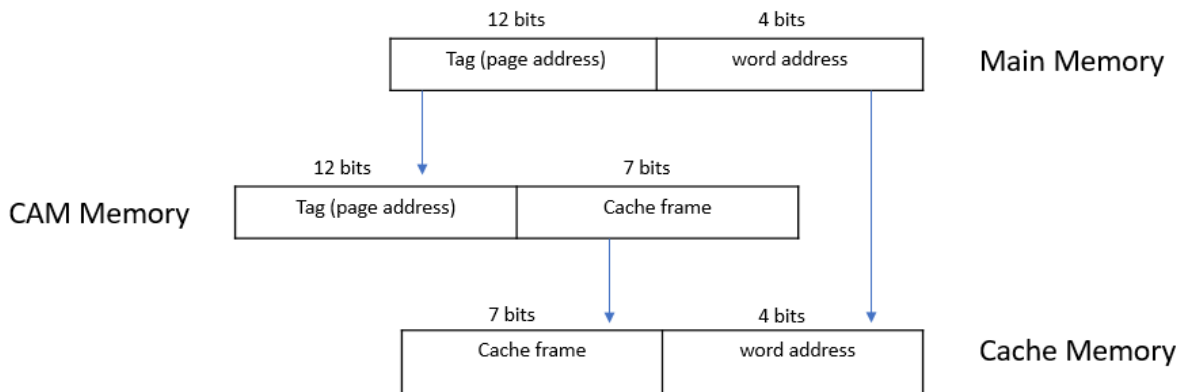


Figure 2.9: Associative Mapping

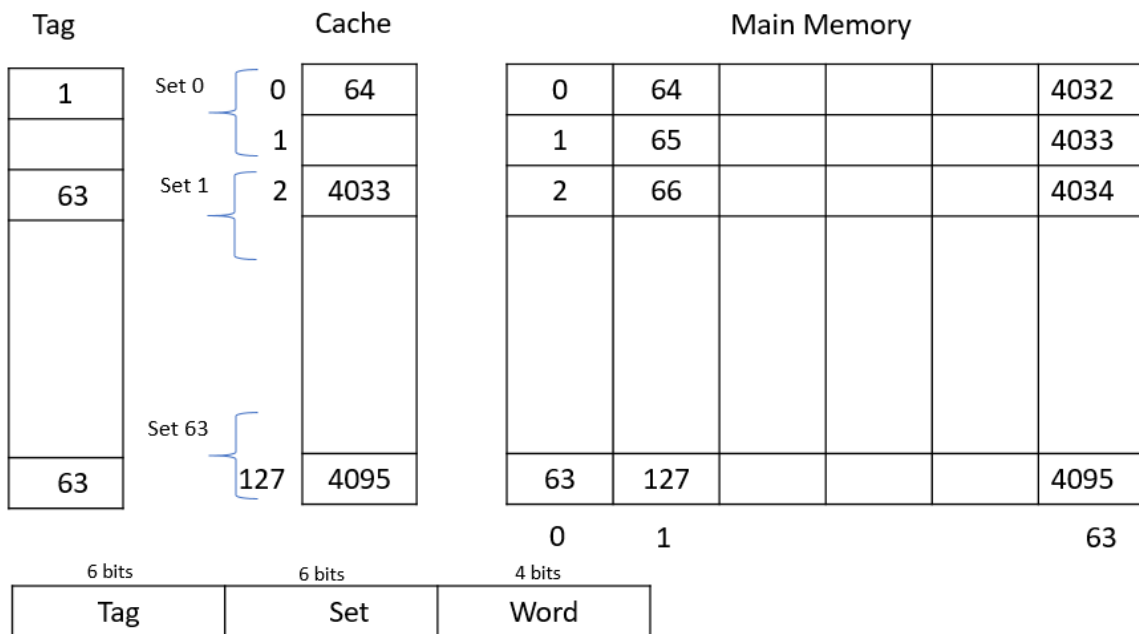


Figure 2.10: Set-Associated Mapping

when we want to map memory block number 8 into the cache using this technique, we can map it into a set number = $8 \bmod 64$ only of the cache. In this technique, the replacement policy is required to replace the occupied cache lines if there are no available cache lines in the specific set.

2.4.2 Cache Replacement Policy

Cache memory depends on the cache placement policy in case the cache is full of data, and there is no space to store more data required by the processor to perform operations. Therefore, algorithms are designed to evict cache block out of the cache to allow more data to be stored to improve overall system performance. We review some of these algorithms as follows:

- *First-In First-Out (FIFO)*: This algorithm replaces the cache block that has remained inside the cache for a longer period. This algorithm is also called the Round Robin algorithm [81].
- *Last-In First-Out (LIFO)*: This algorithm expels the cache block that has been stored in the cache recently so that it deals with the cache memory as a stack.
- *Least Recently Used (LRU)*: This algorithm removes the least recently accessed cache block to store new data. The ageing bit is used to keep track of each cache block's history [81].
- *Pseudo-Least Recently Used (PLRU)*: This algorithm uses approximate measures of ageing bit to replace the cache block to enhance Least Recently Used performance [82].
- *Random Replacement (RR)*: This algorithm randomly selects the cache block to be evicted to allow a new cache block to be stored in the cache [83].

These algorithms are used if the cache is full of data and there is no space to store a new cache block. Therefore, it is necessary to know these algorithms that may positively or negatively affect the success of cache attacks.

2.5 Memory Deduplication

Memory deduplication is a feature that increases memory utilization on physical servers running multiple VMS by allowing the data used by several VMs to be shared between these VMs. The redundant copies of the data are removed from the memory. Table 2.11 shows memory deduplication task to save memory, so if multiple memory pages have the same content, then the hypervisor only keeps one copy of these to be shared between users. Memory deduplication was introduced around the mid-1990s, and it has been implemented in virtual platforms recently [84]. For example, Red Hat proposed the KSM technique to implement memory deduplication in the Linux kernel. KSM has merged into the Linux kernel since version 2.6.32 in 2009 [49]. This feature is supported and offered in almost all the hypervisors, for instance, VMWare ESX and Linux KVM [85]. This feature's central concept is that if there are many identical pages in the content over multiple VMs, the hypervisor removes all copies while keeping only one shared copy between them. While memory deduplication increases memory efficiency, it has a significant influence on virtualized systems security.

Accessing the same data or the same library, especially the shared cryptographic libraries, can be potentially manipulated to leak confidential data of the victim VM's encryptions processes. Thus, memory deduplication can be exploited by attackers for obtaining sensitive information regarding encryption processes. For example, assume we have an attacker VM, and a victim VM reside on the same host, and they share the same memory, and the host enabled the memory deduplication to save memory. The attacker is able to know if a replica executable file is loaded in the victim's VM memory by mapping the executable file into memory and waiting for memory deduplication to take effect. Then the memory deduplication removes the replica executable file and keeps just one copy to be shared between the attacker and the victim. In this case, the attacker can know which function of the shared executable file has been performed by the victim VM using timing information of accessing specific cache lines related to the shared executable file, thus revealing the victim's behaviour. The exploit of the memory deduplication is

accomplished when the attacker's VMs prime or flush the cached data and give the victim's VM some time to access or reload any of these data to be cached again. Hence, the accessed cache data leaks information regarding the victim's activities [20, 85].

Several kinds of memory deduplication mechanisms are implemented by popular hypervisors, such as the Kernel Based-Virtual Machine (KVM) hypervisor. KVM is a hypervisor software applied on the Linux kernel, which employs the Kernel Samepage Merging (KSM) technique. KSM explores or scans all VMs in the virtualized environment looking for all pages that have the same content, and if it finds pages with the same content, they are then deduplicated and shared [86].

In virtualized systems that support the memory deduplication feature, pages may be data, libraries, or executable files shared between two VMs. One of these devices may be a malicious VM. In this case, the malicious VM is able to discover the victim's behavior. The attacker must load an executable file or a page into the memory and then wait for pages with the same content to be scanned, deduplicated, and shared with the victim's VM, which has the same page. Then the attacker can recognize the victim VM's actions and operations by writing on the shared page. If it takes longer, this means that the page has been shared and ready to launch the attack on the victim's VM using this shared page, resulting in sensitive information disclosure[85].

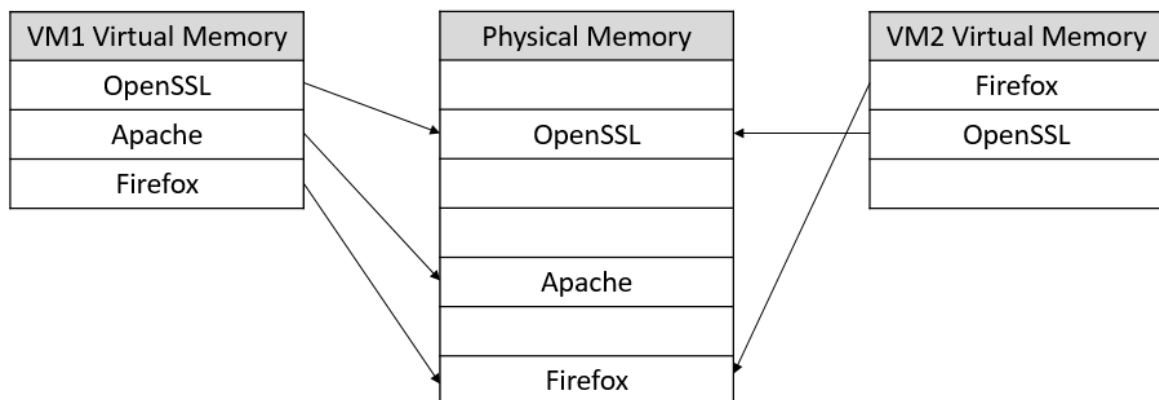


Figure 2.11: Memory Deduplication Feature

2.6 Current Solutions and Mitigations in Cloud

There are many countermeasures against cache side-channel attacks. Due to the cache attacks relying on sharing cache levels and the time difference between accessing data from the cache and main memory, most of the existing defence methods have been proposed based on the following ideas: eliminating imbalance, partitioning caches, avoiding co-location, and detecting malicious activities. However, they have the following shortcomings. Firstly, applying them to cloud computing requires significant changes to the computing infrastructure. This may hinder their adoption by cloud providers. Also, some of these methods may cause system performance degradation and high overhead. They also have a high false rate (positive and negative) in detecting malicious activities. Moreover, they lack the diversity and comprehensiveness of protection against cache side-channel attacks and the provision of preventive measures during analysis processes to detect malicious activities and malicious programs. This section reviews prior works focusing on detecting and preventing cache side-channel attacks and other microarchitectural attacks and identifying their limitations, which has urged us to focus on these limitations and consider them in our work.

A set of studies have relied on machine learning for detecting cache side-channel attacks by analysing malicious behaviour in order to identify side-channel attacks and their specific pattern. They include Zhang et al.[30], who presented CloudRadar, a detection mechanism used to decrease cache-based side-channel attacks in cloud systems significantly. It uses a combination of a signature detection method and anomaly-based detection supported by a hardware performance counter such as *perf* used in the Linux Kernel. CloudRadar uses a database to store signatures for use in comparison to identify suspicious behaviour. However, CloudRadar is unable to identify attacks that exhibit only minor changes from existing attacks because CloudRadar depends on signature-based detection, so it can only identify a specific pattern of attack program behaviour. If there is even a slight change in the attack program's behaviour pattern, the attack will be undetected, meaning that this method only detects attacks known with a specific pattern.

Some detection approaches have taken advantage of Hardware Performance Counters (HPCs) in various manners to profile the behaviour of a malicious program, thus detecting cache side-channel attacks. The motivation for utilising HPCs is to obtain information from hardware such as the CPU to detect attacks. Chiappetta et al.[24] introduced several detection mechanisms that rely on hardware performance counters. The proposed mechanisms are able to detect two kinds of cache side-channel attacks, Flush+Reload and Prime+Probe attacks, and fulfil their essential purpose. However, the mechanisms should expand their scope to include other cache attacks. When detection approaches utilise hardware performance events to detect cache attacks, the events must be carefully selected and, after examination, be analysed afterwards because there are side-channel attacks, Flush + Flush, which do not affect these events. Therefore, it is difficult to detect this type of attack. [23, 27, 87] utilised Intel Cache Monitoring Technology (Intel CMT) for collecting hardware events and then analysing these events using different machine learning. They can detect prime+probe and Flush+Reload effectively; however, their mechanisms cannot detect a Flush+Flush attack, one of the stealthiest cache attacks, because such an attack does not produce cache misses [21] to be monitored by their mechanisms.

Some studies have suggested solutions based on supervised deep learning, such as Cho et al.[88], and unsupervised deep learning, such as Gulmezoglu et al. [25]. These studies utilised performance counters provided by Intel Performance Counter Monitor (Intel PCM) to profile cache attack behaviour and analyse the counters' events. They detect these attacks after, or profile the expected behaviours of benign applications and then detect abnormal behaviours using unsupervised deep learning. However, these detection mechanisms require very complicated computation and much training data, especially in the case of unsupervised deep learning models.

There have been approaches that do not depend on machine learning but rather other ideas, such as the approaches suggested by Wang et al. study [89] and Irazoqui et al.[26], who relied on static analysis of malicious programs and applications to detect

them. Irazoqui et al. proposed MASCAT, a mechanism used for microarchitectural attack detection by static analysis of attacks' executable files. MASCAT utilises static analysis to scan an attack's elf files, searching for implicit opcodes of an attack that are usually present in the design. However, MASCAT has a set of limitations that may hinder its adoption as a suitable solution for virtualised environments, as it contains a high percentage of false positives. It also creates significant overhead in the system. Moreover, it can be used inside a local device only to scan executable files, meaning that it is not usually used to detect and protect against malicious programs in shared virtualised environments to scan a VM's disk and RAM. Total reliance on static analysis consumes a lot of system power and causes a high overhead if it is used on a large group of virtual machines. Therefore, these solutions may not be suitable for shared virtual systems.

Some studies have proposed mechanisms [28, 29] that add noise to distract the time difference between the cache hit and the cache miss in shared systems or eliminate fine-grained timers using fuzzy timers instead of high-resolution clocks to deal with this risk. However, implementing these solutions is not attractive because it requires more modifications to the hypervisor. Also, they are not feasible for applications that require fine-grained timing information [59]. Further solutions have been proposed to reduce the probability of sharing the same resources between the victim and the attacker by designing VM placement policies[90]; The fundamental concept is to restrict the number of servers allowed for each account to use, hence reducing the attacker's exposure to target VMs. This policy increases the possibility of co-locating VMs associated with the same user account, making it challenging to complete co-residence with the target VM. However, this policy has apparent limitations related to workload balance and power consumption [91]. Reducing the probability of sharing the same resources can be achieved by dividing the cache into several zones and assigning one for each VM, thus leading to partial isolation of VMs [92]. This approach may effectively isolate caches between distinct processes performing sensitive functions [93]. However, it limits the number of VMs that use shared cache on the same host, and it requires significant changes in the current cloud model to be adopted effectively [91].

Some microarchitectural attacks may affect the performance counters of hardware; they have a clear impact within systems and so they can be detected by analysing these counters. Therefore, some studies have relied upon performance counters to detect some of these attacks, such as [94] that introduced a detection mechanism that could identify Spectre attacks by detecting side attacks using a neural network model. It also collects data from hardware performance counters to monitor cache activities, as a Specter attack produces distinct patterns for cache activities. Moreover, Aweke et al. [95] developed the ANVIL that uses hardware performance counters to obtain information regarding memory accesses and cache miss rate to indicate the frequently accessed rows in the DRAM and then refreshes adjacent rows to protect from rowhammer attacks. Many studies have proposed solutions based on machine learning, such as [96] that introduced a mechanism that can analyse the access activities of the DRAM to detect a rowhammer attack based on machine learning.

Although various approaches for cache attacks and malware detection have been introduced, these approaches have significant limitations. Some of these approaches are not comprehensive enough for a sufficient number of cache attacks. Also, some are not attractive because they require fundamental changes in the infrastructure of the environment. Moreover, some of them do not give consistent results in certain circumstances. Furthermore, mechanisms based on static analysis perform static analyses frequently and do not require a start-up condition, thus increasing the load on a system. Moreover, they are ineffective in detecting and protecting shared virtualised environments against various side-channel attacks. Additionally, most of the aforementioned methods require improvement in terms of performance and results. Finally, even the most popular antivirus programmes fail to detect any of the threats we have studied (cache side-channel attacks) [26, 64].

Our work focuses on detecting at least several of the most significant cache attacks to extract accurate detection results for such attacks with acceptable system performance. We address the limitations of current detection methods by proposing approaches that

monitor and detect any abnormal behaviour of a VM periodically and then perform static analysis of the detected VM's executable files, classifying the threat level of the VM's executable files using neural network classification algorithms, thus eliminating any malicious VM and protecting the shared virtual environment with acceptable performance.

We have identified resources to provide an overview of the thesis's main topic and to support our study based on three aspects. We have determined these resources based on cache side-channel attacks, how they are implemented, and the most appropriate conditions and environment for their implementation. The second aspect has been identifying systems vulnerable to these attacks and studying their characteristics. The third and final aspect has been to recognise the limitations and shortcomings of state-of-the-art countermeasures against cache attacks.

Table 2.3: Summary of Current Solutions and Mitigations in Cloud

Paper	Idea	Target Attacks	Limitation(s)
[28, 29]	The idea is eliminating the time contrasts between cache hits and cache misses by injecting noise.	Cache Side-channel Attacks	These solutions were not feasible for applications that require fine-grained timing information. Implementing these solutions was not attractive because it required more modifications to the hypervisor.
[94–96]	Using machine learning to analyse HPCs and access activities of the DRAM.	Microarchitectural attacks(Rowhammer, Spectre)	They need to expand their scope to include more than one specific attack.

[30]	This solution relied on signature-based detection, comparing monitored applications with preidentified attack signatures.	Cache Attacks	Side-channel	It could detect attacks with known patterns but could not detect attacks with minor changes in their patterns. It created significant CPU overhead due to continuous comparing processes.
[23, 24, 27, 87, 88]	These mechanisms relied on obtaining data from hardware performance counters and then analysing the data using supervised machine learning to classify behaviours as benign or malicious.	Cache Attacks	Side-channel	These mechanisms provide limited protection, as there are several side-channel attacks that these mechanisms could not detect, such as Flush+Flush. Also, some of these mechanisms required significant modifications and intolerable latency for parsing attack readings. Furthermore, it was vulnerable when an attacker had root access mode.
[90, 92]	Reducing the probability of sharing the same resources.	Cache Attacks	Side-channel	It limited the number of VMs that used shared cache on the same host, and it requires significant changes in the current cloud model to be adopted effectively [91].

[25]	This mechanism relied on obtaining data from hardware performance counters to profile the expected behaviours of benign applications using unsupervised deep learning and then detecting abnormal behaviours if there were differences between the predicted counter value and the actual counter value.	Cache Side-channel Attacks	The detection mechanism requires much complicated computation and much training data for unsupervised deep learning models; making it unattractive to be adopted.[88]
------	--	----------------------------	---

[26, 89]	Using static analysis of applications, then detect malicious applications.	Microarchitectural Attacks	It had a high false positive rate and created significant overhead in the system. Moreover, it can be used only in the app store to scan a specific application; it was not designed to access virtual machines to extract files to scan against microarchitectural attacks.
----------	--	----------------------------	--

2.7 Hardware Performance Counter (HPC)

Modern microprocessors include a series of special registers called HPCs. These counters are utilised to keep track of the number of hardware-related events in a computer system for conducting low-level performance analysis. HPCs provide users with row readings of CPU

events to be analysed and associated with other counters to indicate a specific performance status [97]. We can access performance counters using the *Perf* Tool on Linux systems, which is manageable by the command-line interface to record statistical readings related to hardware to monitor various operations.

2.8 Machine Learning Classification Algorithms

In this section, we review some of the machine learning algorithms that were used in our projects to analyse the data effectively. The machine learning algorithms are characterised by their ability to train on the data of different systems and interact with them, which increases their development. The following supervised machine learning algorithms were relied on in our projects to support accurate decision making.

2.8.1 Logistic Regression

Logistic regression is a type of supervised machine learning approach utilised for binary classification to classify data into a set of specified categories, for example, classify an email as spam or not. There are more classification problems logistic regression can handle. It is a predictive analysis algorithm based on *Sigmoid* function $S(z) = \frac{1}{1+e^{-z}}$ to limit its predicted values between 0 and 1, that is the main difference between logistic and linear regression, as shown in Figure 2.12. In contrast, Linear regression can have greater than one and less than zero values. The sigmoid function produces probability values p between 0 and 1. In order to assign these probability values p to a separate class such as *True/False* classes, we set a threshold value at 0.5 such that $p \geq 0.5 = \text{Class 1}$ and $p < 0.5 = \text{Class 0}$. For example; If the prediction function returns a value of 0.6, the value is classed as 1, but if the prediction function returns a value of 0.2, the value is classified as 0. We made use of the logistic Regression concept to make classifications of two classes, and it was applied using the sigmoid function to divide between different benign and malicious activities, as discussed in Chapter 3.

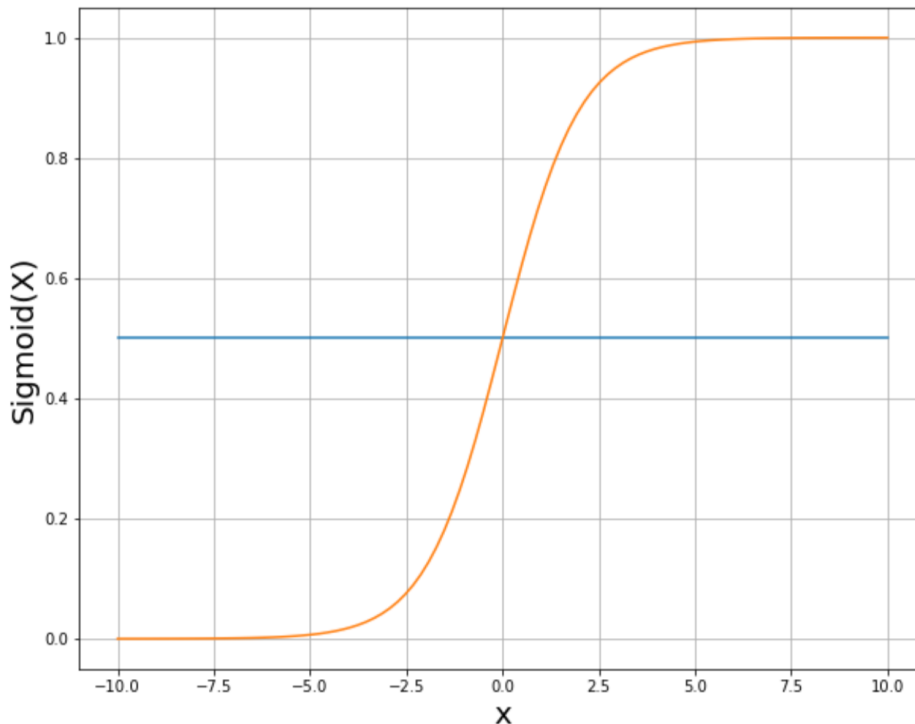


Figure 2.12: The sigmoid function takes two values, either zero or one, and it takes a curve that looks like the letter "S", and the blue line represents the threshold that separates two groups in the classification cases.

2.8.2 Softmax Regression

Softmax Regression or Multi-class Logistic Regression is a generalization of a logistic regression classifier algorithm that can handle multi-class classification problems such that $y^i \in \{1, 2, 3, \dots, C\}$ where C is the number of classes [98]. In contrast, the logistic regression model can classify between two kinds of discrete classes such that $y^i \in \{0, 1\}$.

In softmax regression, we replace the sigmoid function with the softmax function that is $S(z_i) = \frac{e^{z_i}}{\sum_{j=1}^N e^{z_j}}$. The softmax function receives the input vector, which is the output of the last layer of the neural network models, then normalizes it into a probability distribution for classes of the model. The sum of these values of the output equals 1 such that $\sum_{i=1}^N S(z_i) = 1$. We made use of the Softmax Regression concept to make classifications of more than two groups, and it was applied using neural networks to divide between different threat levels, as discussed in Chapter 4.

2.9 Summary

This chapter discussed a set of essential topics for comprehending the research project as required, such as the concept of multi-tenancy for cloud computing, the concept of memory deduplication, and their importance and disadvantages as factors assisting in implementing side-channel attacks.

the chapter also highlighted the side-channel attacks and microarchitectural attacks that threaten cloud computing. Furthermore, it reviewed the implicit characteristics of these types of attacks. It also discussed several previous studies on these attacks.

We also reviewed several possible current solutions against these attacks and identified their advantages and limitations for these proposed solutions to mitigate side-channel threats. We also discussed the machine learning algorithms that were used for analysis and classification in our project.

MITIGATION THROUGH MEMORY DEDUPLICATION

This chapter describes a method for detecting cache side-channel attacks using the memory deduplication feature. The VM uses the method to monitor cache locations of the sensitive shared executable file and shared cryptographic library and then classify suspicious behaviours using the logistic regression model. Our proposed method in this chapter can also work in conjunction with the other methods of the subsequent chapters to provide more robust and more reliable protection.

3.1 Introduction

Sharing applications and libraries is an optimisation technique introduced to improve memory utilisation of the shared virtualised system and thus improve overall system performance. The effect of this technique appears when a considerable number of virtual machines are running on the same physical machine and share the same physical memory. However, this technique raises the level of threats to reveal confidential information of victims' VMs when some of these sensitive programs, such as cryptographic libraries, are shared with malicious VMs [20]. Malicious users exploit the difference in timing information between retrieving data from cache (cache hit) and main memory (cache miss), as shown in Figure 3.1 that shows the scenario of retrieving data from the cache as well as from the main memory. Also, Figure 3.2 shows the normal distribution of CPU cycles for cache Hit and cache Miss. Attackers can exploit this information to break the isolation between VMs, uncover the victim's actions, obtain information about cryptographic operations, and break the encryption key.

In this chapter, we focus on a specific type of side-channel attack, which is enabled by the use of a cloud-computing virtualisation feature called *memory deduplication*. Memory deduplication is a method of reducing memory usage by keeping in the memory of the server only one copy of the data and code used by multiple VMs. An attacker can use deduplication to access the physical addresses of the shared memory area, followed by cache flush operations on any addresses of interest. Subsequently, any memory locations used by the victim VM are brought back into the cache, and can be identified by the attacker since the time to retrieve them is much shorter than the retrieval time for memory locations that were not used by the

victim. Determining the memory locations accessed by the victim VM in this way can reveal sensitive information about the victim [4, 20, 22, 99].

Although several methods for mitigating this type of side-channel attack have been proposed [23–25, 27, 30, 87, 88], these methods have significant drawbacks. First, they require the execution of code on the host (i.e., the physical machine), which is not something that the victim (i.e., the user of the VM) can typically do. Additionally, they do not provide any preventative protection for the victim VM. In particular, any false-negatives allow the malicious VM to perform the attack undetected.

We address these limitations of current mitigation solutions by introducing a new method that protects against memory-deduplication side attacks from within the victim’s VM. Our method monitors sensitive data addresses, and renders attacks ineffective by providing the malicious VM with fake results during attack attempts, even in the (rare) instances when these attacks are not detected (due to false negatives). To this end, the method uses memory deduplication itself to get readings of the monitored functions of an executable program and then analyzes the readings using logistic regression. The proposed method can be used inside the VM to be protected from cache attacks, with no changes to the virtualization platforms. The main contributions of this chapter are:

1. We present a method for for protection against memory-deduplication side-channel attacks by using memory deduplication and logistic regression from within the victim VM.
2. We introduce the design and implementation of the method.
3. We evaluate the method in multiple scenarios, in terms of attack detection accuracy and performance characteristics.

The remainder of this chapter is organized as follows. In Section 3.2 describes the proposed protection method. Section 3.3 provides an overview of the experiments we carried out using the new method. In Section 3.4, we discuss the evaluation of the implemented method and

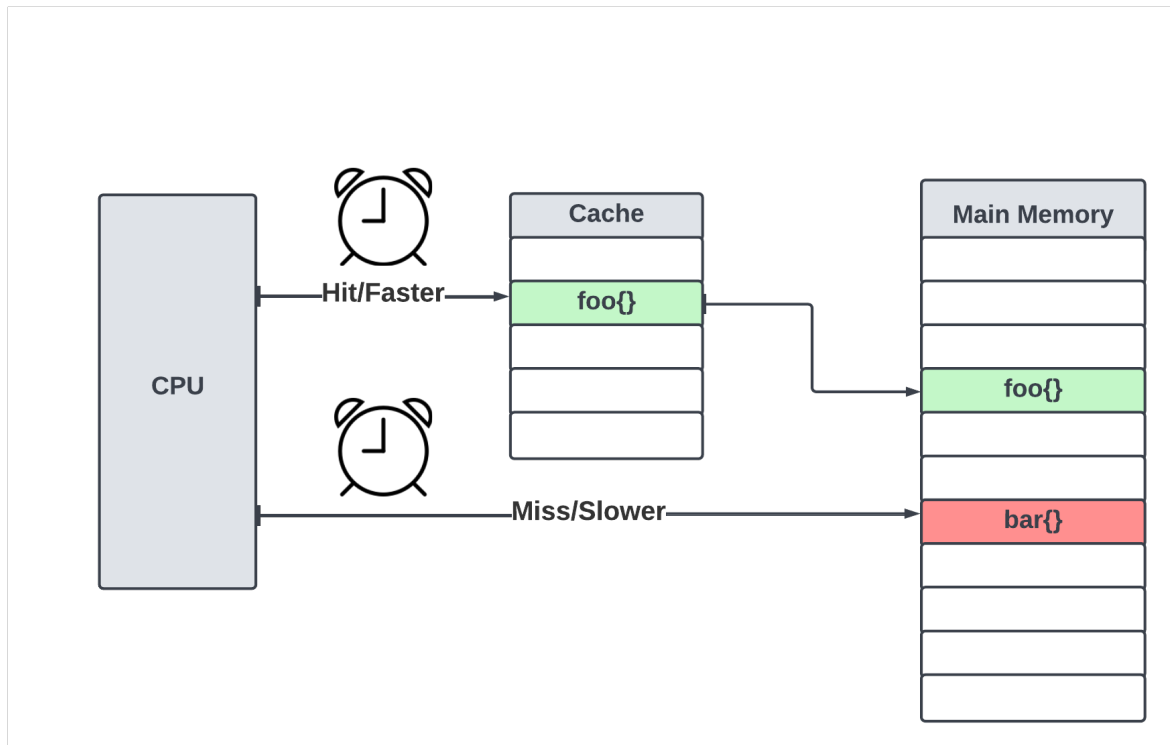


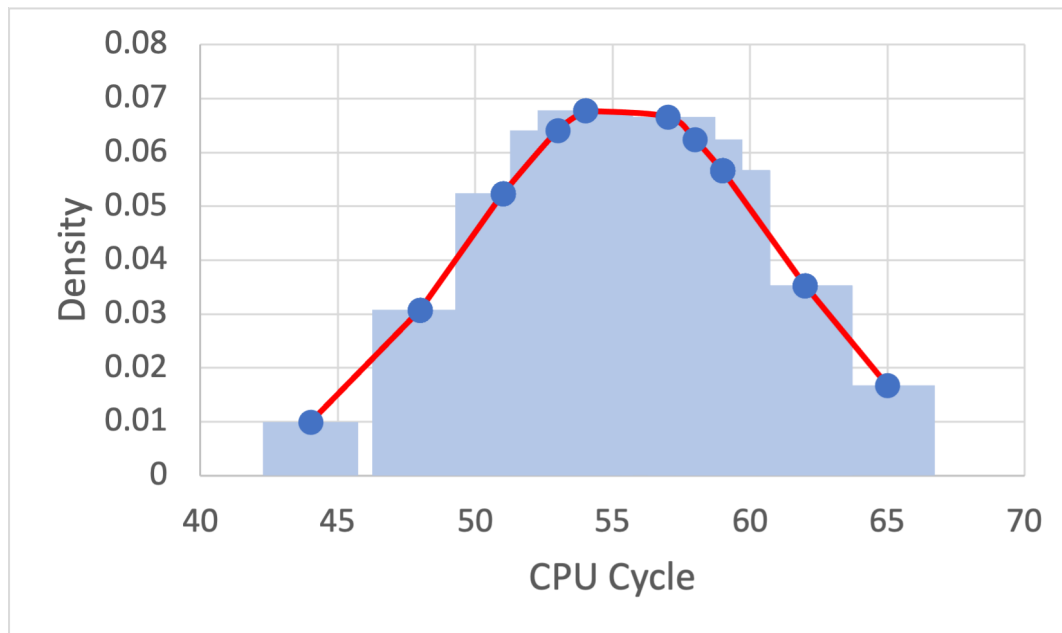
Figure 3.1: Cache Hit and Miss.

compares our method to related work. In section 3.5 provides an overview of limitations related to our method. Finally, Section 3.6 provides a brief conclusion.

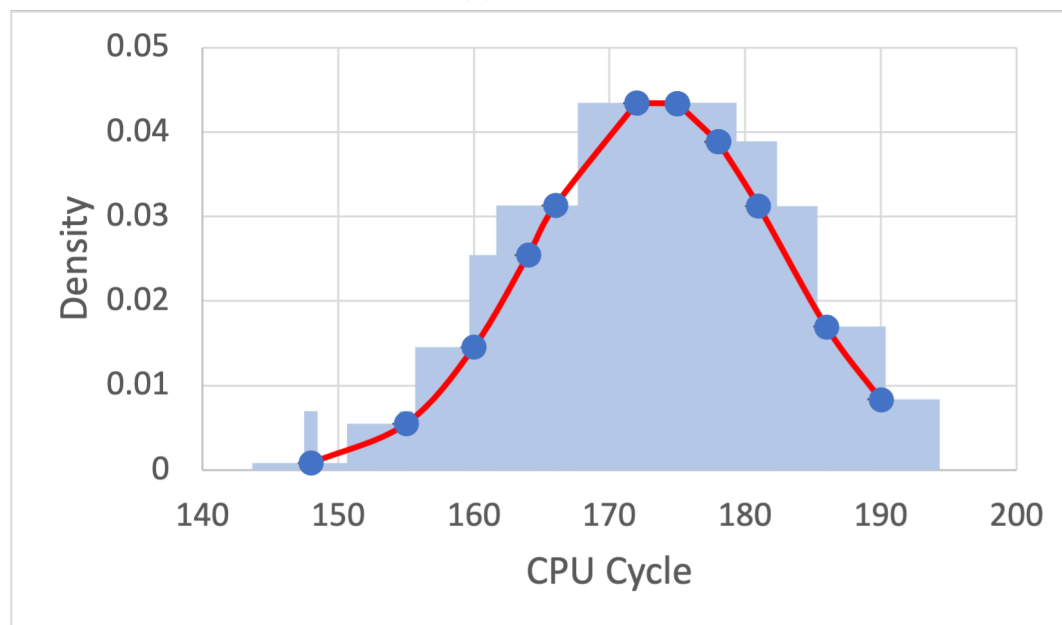
3.2 Protection Method

This section explains our proposed protection method. Consider two VMs in a virtualised environment that supports memory/page deduplication. One of the VMs is the attacker and the other is the attack target (victim). Both VMs are located on the same physical host and share the LLC and some files (e.g., libraries and executable files). The memory deduplication mechanism removes redundant copies of these files, so that only a single version of each shared file is retained.

As illustrated in Figure 3.3, users can access the shared memory addresses and are able to conduct flush-based side-channel attacks. Attacker's actions when executing the flush-based attack are as follows. The attacker identifies the desired memory pages that are to be



(a) Cache Hit



(b) Cache Miss

Figure 3.2: Normal Distribution of Cache Hit and Miss

monitored over a certain period of time and flushes them out of the cache, using the *clflush* instruction (flushing may need to be repeated multiple times to ensure the attack's success). The aim is that the flushed pages are recovered from the main memory when these pages are requested by the victim. Next, the attacker reloads the desired pages and measures the access

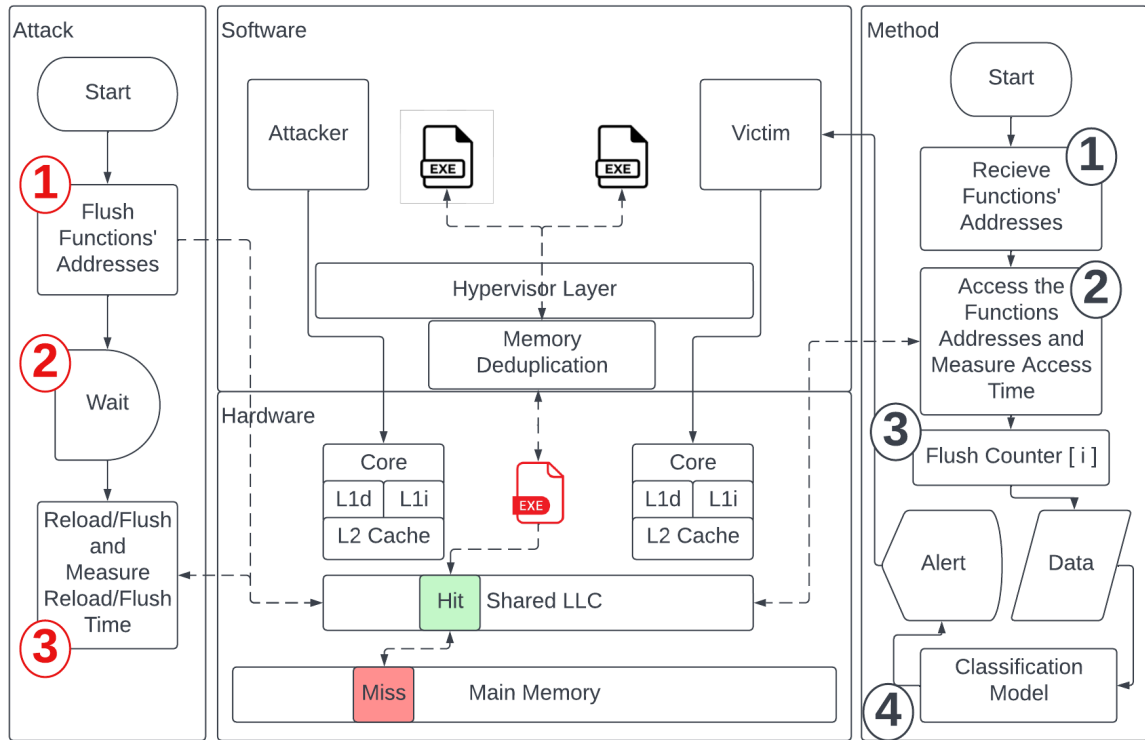


Figure 3.3: Flush-based Attacks Exploiting Shared LLC.

time (using the *rdtsc* instruction) to determine whether or not the victim has requested those pages.

The protection method involves the following steps:

1. Receives the addresses of the executable program's functions to be monitored and protected from the flush-based cache attacks.
2. Retrieves the monitored functions into the cache memory.
3. Measures the time taken to retrieve each function over each specified period of time. As a result, the functions will be pre-fetched and the flush instructions will be discovered. The measurement is carried out using the *rdtsc* instructions that provide a high-resolution time stamp counter [21]. It uses the *mfence* instructions as well.
4. Sets a sample for each function so that the retrieval time is measured frequently. It is

then measured against the threshold of the system to detect if flush instructions have been performed on the functions.

The aforementioned method can detect attacks based on a flush instructions, including the Flush+Flush attack . It requires no memory access and thus produces no cache misses. However, the cache hits are significantly decreased because of the continuous cache flushes, comparing to any other cache attack[21]. The protection mechanism runs parallel to the execution of sensitive operations in another terminal. It can run on a VM to monitor the sensitive program's physical addresses obtained by the debugging tool. Debugging process may be complicated for the user unless the sensitive program's physical addresses are identified and entered automatically. The attacker may notice that the protection mechanism is running because the attacker continuously records the same results.

Figure 3.3 shows that two virtual machines, one malicious and the other victim, are working in a shared virtual system that supports the memory deduplication feature. Both VMs share the last level of cache and also share the same cryptographic library due to the use of memory deduplication, so both virtual machines can access the same shared memory addresses of functions of the shared cryptographic library. Then The attacker takes the following steps:

1. The attacker flushes the target shared memory addresses out of the cache, using the *clflush* instruction (to ensure the success of the attack, the attacker might have to flush the same addresses repeatedly).
2. Then, the attacker waits to allow the victim some time to carry out encryption processes using the same shared memory addresses of the functions.
3. The attacker then reloads the addresses of the flushed functions and calculates the access time (using the *rdtsc* instruction) to determine whether the victim has requested and executed those functions.

The other part of the figure represents how to protect against these side-channel attacks using the proposed method that takes the following steps:

1. It receives the shared functions' addresses of the cryptographic libraries to be monitored and protected from these attacks.
2. Then, the protection mechanism retrieves the monitored functions' addresses into the cache memory while measuring each function's retrieval time. As a result, the functions will be reloaded, and the detection mechanism will discover the flush instructions.
3. It records the number of flushes for each monitoring function and saves them as a CSV file to be analysed later.
4. Then, the protection mechanism analyses the number of flushes for each function using a logistical regression model. and warns the user in case of attack.

3.3 Experiments

This section explains how to carry out experiments to prove the effectiveness of the proposed method. In this section, we describe the objectives of the experiments and how to create the threat model and the experimental environment. The results of the experiments are also recorded for the various systems used in the experiments.

Our approach's main idea involves the use of memory deduplication to periodically analyze the behaviors of the running VMs that are using the same executable program to recognize abnormal actions on the cache memory that could indicate and detect cache attacks. The proposed method's objective is to provide accurate information about the VM's activities conducted on the cache memory when two or more VMs share the same executable program. That information will be analyzed and classified using machine learning algorithms for detecting malicious activities.

The design and implementation objectives of the proposed method are as follows:

- It should provide cache-related information that indicates flush-based cache attacks among VMs through the shared LLC (last level of cache).
- The design should not require significant changes to the hypervisor to correctly and integrally apply the method on any hypervisor.
- It should produce accurate and reliable information with acceptable performance.
- It should be compatible with machine learning algorithms for providing a self-protection mechanism for the VM.

3.3.1 Threat Model

In this section, we describe our assumptions regarding the environment. We assume that we have two VMs (attacker and victim) share only the LLC; this is accomplished by assigning a different core for each; this means that the other cache levels are not shared between them. The proposed approach endeavours to provide information about the flush-based cache attacks carried over the shared LLC using data deduplication. Also, we assume both the attacker's VM and the victim's VM share the same executable program that is deduplicated using KSM (QEMU-KVM hypervisor), and each VMs is equipped with the GNU Project Debugger (GDB), which was the scenario used in previous studies [22, 99] We further assume that the attacker's VM and the victim's VM can use the x86 architecture set, such as *rdtsc* and *clflush*. The attacker's VM and the victim's VM do not have to be privileged users to use these instructions. In these circumstances, the attacker can access the last shared level of cache and perform operations such as flush operations.

The attacker is able to identify the functions addresses of the shared executable program because of memory deduplication, so when conducting a flush+reload attack on these functions, the attacker is capable of knowing the behaviour of the victim who is sharing the same executable program and memory locations for these functions, so the attacker able to know which of the functions were executed by the victim just by analysing the results of the

flush+reload attack and distinguish between the functions that registered cache hits or cache misses.

The attacker's goal is to attack the functions of shared executable programs or the functions of cryptographic libraries shared by several virtual machines. When the attacker identifies the functions executed by the victim, he can identify the victim's behaviour in executing the shared program. The attacker can also identify the behaviour of the encryption process executed by the victim, thus breaking the encryption keys as explained in section 2.2.

3.3.2 Experimental Results

We performed experiments on the KVM hypervisor that runs KSM as a memory-saving deduplication feature. We created two VMs running a Linux operating system; one VM acting as a victim and the other acting as an attacker. We used QEMU-KVM hypervisor on a Debian 10 (buster) host with Intel Core i5-4200M CPU, 12 GiB memory, and Ubuntu's guest VMs 18.04.4 LTS. We also used QEMU-KVM hypervisor on a CentOS Linux 8 host with Intel Core i5-5300U CPU, 8 GiB memory, and Ubuntu guest VMs 18.04.4 LTS. We created the shared virtualized environment using the hypervisor's default settings. We mean by default settings that KSM is active on the QEMU-KVM hypervisor, which allows the hypervisor to save RAM (memory deduplication) by sharing executable files and applications over multiple VMs and processes of the same program with a reduced memory footprint, and makes it uncomplicated to find a shared physical address between two VMs to exploit. Furthermore, We installed no additional security software. We assumed that the victim's and the attacker's VMs were pinned to different processor cores, which means the LLC was shared between them. Thus, the other levels of cache were not shared among the VMs. The method intends to provide data about the shared executable program using a side-channel carried between two VMs within the shared LLC.

In both VMs, we installed and ran the same executable program, shared using KSM. We checked the number of pages that have been shared between VMs. We found the addresses of the application's functions using debugging tools such as GDB to be used as inputs in our proposed de-

tection program. After that, we created a covert channel between VMs to monitor the attacker behavior. We used a Mastik Framework [22, 100], which has several libraries to use and recreate many important functions such as *map_offset()*, *fr_monitor()*, and *fr_probe()*.

We developed a C program implementing Algorithm 1, which counts the number of flushes that happen to the shared executable program; flushes for each function are counted. This indicates the instability status of the cache memory due to the presence of a large number of flushes that occur in a short interval of time (e.g., 5 seconds). Figure 3.4 shows the results from a series of experiments that we carried out to evaluate the feasibility of our method.

Algorithm 2 Flush Counter in a Given Interval of Time

Input: App.elf, $F_Add_1, F_Add_2, \dots, F_Add_n$

Output: Number of flush operations in a given time interval

```

1: Read-Only Mmap App.elf into Memory Offset = 0;
2: for  $i = F\_Add_1$  to  $F\_Add_n$  do
3:   Initial Access Time Measurement, Prefetching to Cache
4:   FlushCounter[  $i$  ] = 0;
5: end for
6: while true do
7:   T1 = Start Clock ();
8:   Pause for 5 seconds
9:   for Iteration = 1 to Total No of Measurements do
10:    for  $i \leftarrow F\_Add_1$  to  $F\_Add_n$  do
11:     Time = Access Time Measurement( $i$ );
12:     if Time > Access Threshold then
13:      FlushCounter[  $i$  ] ++;
14:     end if
15:    end for
16:   end for
17:   T2 = Stop Clock();
18:   Time_Elapsed = T2 - T1 // time in microseconds
19: end while

```

The protection mechanism performs a series of functions. First, using the *map_offset()* function to load the executable program into memory as a Read-Only file means none of the users can modify the file to be shared between users. However, once the user modifies the file, the KSM will create a separate copy of the modified file only for the user who made the modification and will stop the deduplication of the file. After that, the user inputs the addresses of the functions that are needed to be monitored. It then initially measures the time it takes to retrieve data from these addresses, and thus we ensure that all addresses have

been placed in the cache memory. In addition, the approach will be ready to measure any flush instructions that occur to addresses that cause successive cache misses; their impact will be clear at the measurement time. We can then measure the actual time to retrieve data from addresses. Before that, we determine the system's threshold to differentiate between retrieving data from the cache memory comparing with retrieving data from the main memory. If the retrieving time is greater than the system's threshold, this means that the addresses have just been flushed out of the cache. After that, we measure the time that the CPU takes to measure the time of retrieving data, showing that the cache memory is unstable and exposed to flush operations and thus an attack on the cache memory. To be more precise, we create a sample loop so that we can measure the retrieving time for each address several times. The frequency can be changed according to the system.

We recorded the results for Debian 10 and CentOS 8 in different cases. First, we recorded the results in a normal case without any attack on the cache memory. Secondly, we recorded the results when the attack was on only one function in the shared executable program. Finally, we recorded the results in the case of an attack on all functions in the shared executable program. The relationship between the attack state (red line) and the normal state (green line) has been clarified using the plot charts shown in Figure 3.4. The plot charts show a big difference between an attack and no attack, thus a big difference in the stability of the cache memory in a short period.

In Figure 3.4, the results are recorded during all the scenarios mentioned above. In the first plot, the results are represented for all addresses that were monitored in the absence of an attack. It is also considered a stable case for a longer period compared to the results of other cases. As for the second chart, it represents the suspicious state of all the sensitive functions addresses, in cases of attacks on them all. It shows a high rise in flush commands on the addresses, indicating the attack status. As for the following charts, they show the cases of the attack on an address for one function only, as they show instances of instability for different periods due to the presence of the flush commands, indicating the attack on the specific function in each chart.

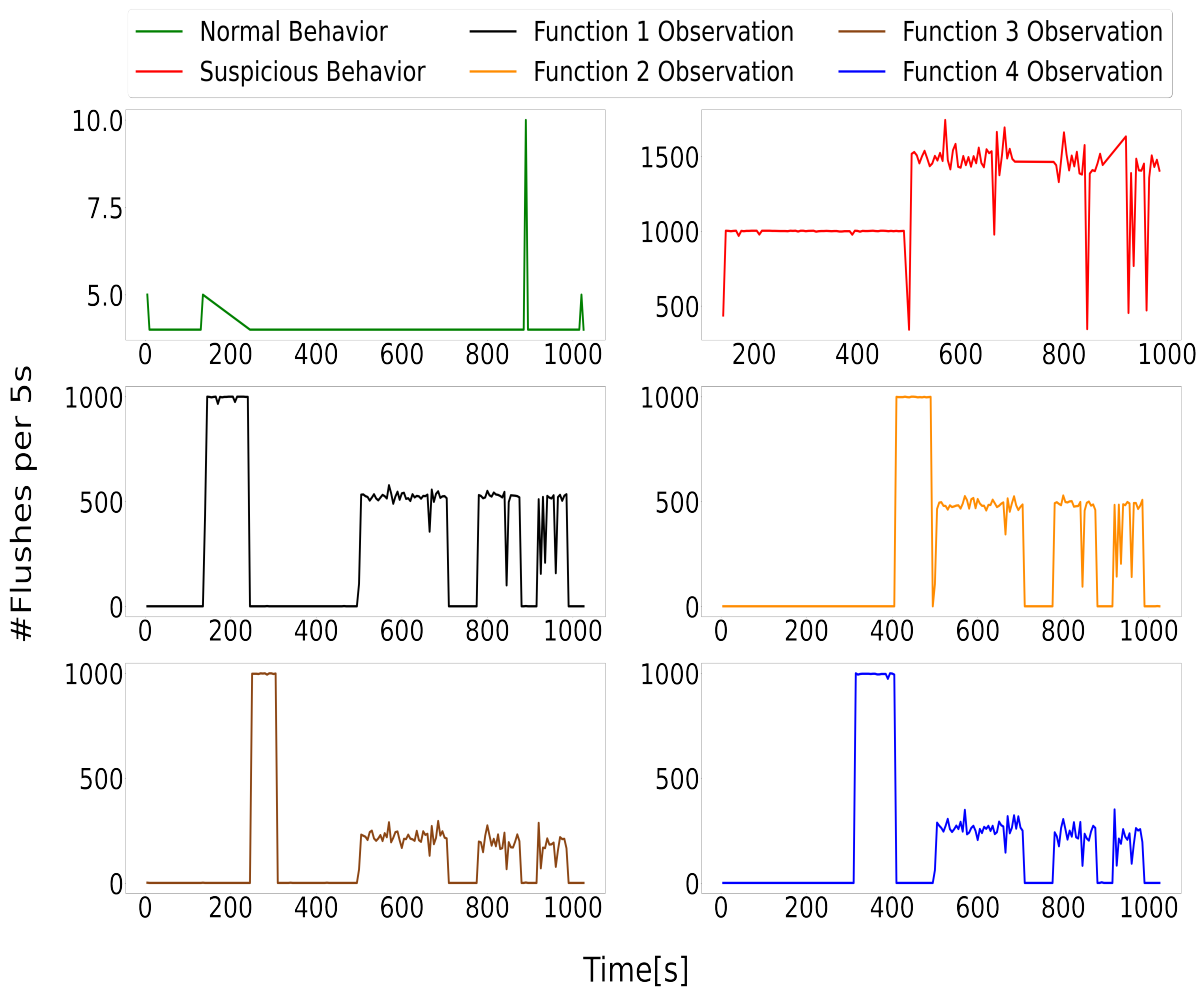


Figure 3.4: Number of flushes per 5s for different attack scenarios: no attack (top left), attack on all four functions of a software application (top right), attack on one of the functions of the application at a time (second and third row) .

Figure 3.5 We mean by "Normal Activity" that the virtual machine is running without any programs or applications running in the virtual machine background, and by "Noise", we mean the state of running programs and applications such as the browser in the virtual machine background. Therefore, we have clarified scenarios in this figure, which are monitoring the number of flushes while running programs and applications on the virtual machine (Normal Activities + Noise), and the other scenario is monitoring the number of flushes during performing the attack with applications and programs running in the background of the virtual machine (Attack + Noise). The figure shows a significant difference in the number of flushes in both scenarios.

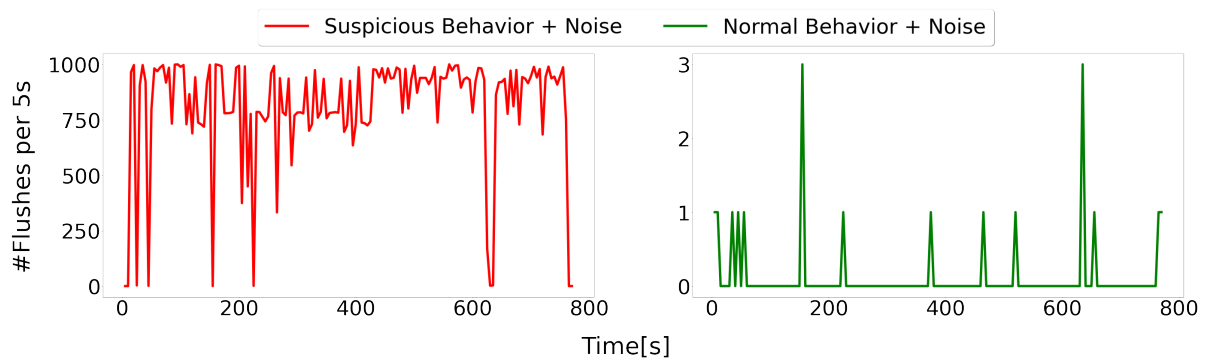


Figure 3.5: Number of flushes per 5s for attack and normal activities with background noise scenarios: attack (left), attack on a function of a software application with background applications noise (right), normal activities with background applications noise.

3.4 Evaluation and Comparison to Other Solutions

In this section, we evaluate and compare our protection solution to the other solutions. Previous works have been based on the performance counter provided by Linux and Intel, such as Intel PCM (Performance Counter Monitor), Intel CMT (Cache Monitoring Technology), and Linux perf. Most previous studies have used these tools inside the host machine to monitor counters affected by an attack. However, if they are used inside a VM then there will be limitations; the guest VM needs authorization from the host to use the hardware performance counters, and not all provided counters are supported for use in a VM[101–103], which makes it difficult to detect an attack using these tools inside a VM in order to provide it with self-protection. To validate our approach, we ran it on different operating systems (Debian 10 and CentOS 8) in different scenarios for the virtualized platform. We used the Mastik framework to recreate the attack. The attacker flushes the functions of the shared executable program and the sharing was achieved by the data deduplication feature. The scenarios were as follows.

- *No-Attack Scenario*: The attacker conducts no cache attack on the side channel, nor is there any sign of suspicious behavior. The results were almost consistent with only one flush for all functions and for both operating systems used in the experiment, indicating no attack on shared re-sources.

- *One-Function-Attack Scenario*: The attacker executes the flush-based cache attack on a shared executable program to attack only one function. The attacker specifies the function address to execute repeated flush instructions on this address. We recorded a very high number of flush instructions. The results were similar for both operating systems used.
- *Multiple-Functions-Attack Scenario*: In this scenario, the attacker performs the side-channel attack on a shared executable program to attack multiple functions. In both Debian 10 and CentOS 8, we recorded a different number of flush instructions, and the results were clear enough to indicate an attack on all functions. We produced plot charts that better illustrate the results of the experiment's scenarios, as shown in Figure 3.4.

Based on the obtained results, we were able to identify suspicious behaviour that exploited the shared executable program, which indicates the attack status. Also, the VM was provided with our self-protection mechanism by detecting the attack's impact on shared resources and obfuscating the results of the attack, as shown in Figure 3.8. The attack's effect on the system was identified, which was the cache state's instability for a long time and a massive increase in the number of flushes, which meant a significant difference in the case of an attack and no attack.

Like in the experiments shown in Figure 3.4, we evaluated the proposed approach in several scenarios: no attack, attacks on only one function of an application, and attacks on all four functions from our application. In the no attack case, the results were almost constant at one flush per time period (i.e., per 5| seconds) for all functions within the application, making it proof of non-attack. In the case of attacking one function, the results show a high number of flush instructions. For the attack on multiple functions, the results (shown in Figure 3.6) differed between the operating systems we used in our experiments.

In Debian 10, we recorded very large numbers of flushes (Figures 3.6a), while for CentOS 8 (Figures 3.6b) we had to increase the number of iterations of the for loop from lines 9–16 in

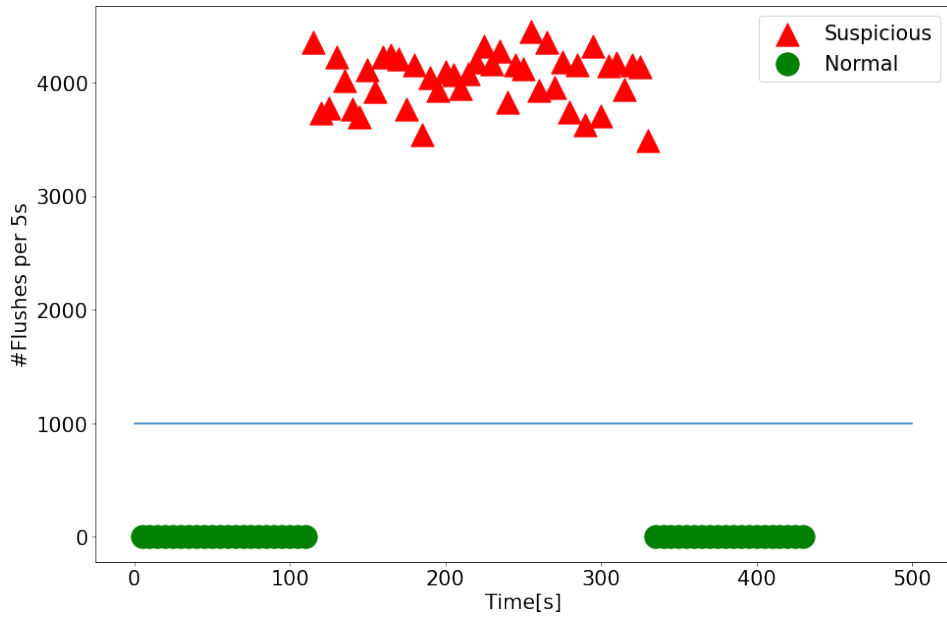
Algorithm 2 in order to observe the increased number of flushes caused by the attack. The explanation for the fewer flushes when several functions are under attack is that the attack is distributed across the multiple functions, with fewer flushes of individual memory locations occurring within a given period of time. With this calibration, we distinguished between the no attack and attack cases easily for both operating systems.

Although the results recorded in the case of attacking multiple functions in CentOS 8 showed fewer flush instructions than the results in Debian 10, the difference between it and the non-attack case was obvious and significant, and we could easily distinguish between them, as shown in Figure 3.6b. We could also specify a threshold to help increase the accuracy of the results or even increase the number of iterations. For example, if we set the threshold at 12 flushes, all the attack cases could be detected, as was proven in the experiment results. Suppose we set 12 flushes as a threshold to indicate suspicious behaviors. In such a case, the attack will always be detected because, for all experiments, the recorded results would be more than 12 flushes for all attack cases. That makes the proposed approach effective in helping to detect an attack.

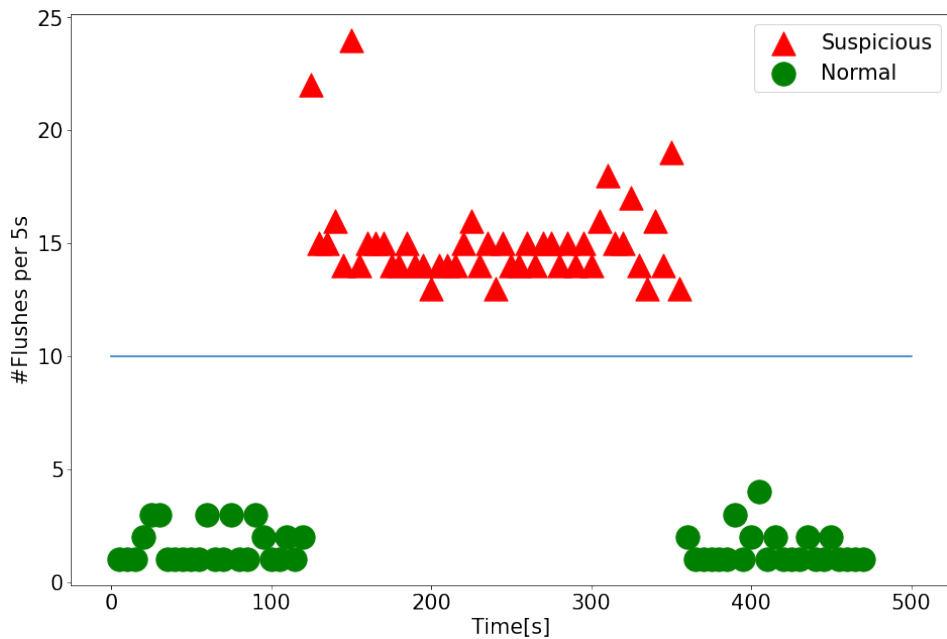
We tested the effectiveness of the detection mechanism when attacking a shared cryptographic library. We attacked the GnuPG implementation of RSA. Our detection mechanism was monitoring memory lines for the sensitive functions of the GnuPG executable, namely Square, Multiply and Reduce functions. We have implemented the Flush+Reload attacks and the Flush+Flush, so we found the readings extracted from the detection mechanism obvious to differentiate between the attack and non-attack cases with a difference in the threshold as shown in Figure 3.7.

The results of the readings for all scenarios were fairly evident. However, the difference of threshold was critical in all scenarios, so it was necessary to use an analysis mechanism for the extracted data, and to train the machine on all classes and scenarios, so we used machine learning, specifically used the logistic regression algorithm, to help us in this process effectively to increase the effectiveness of the detection mechanism.

3.4. EVALUATION AND COMPARISON TO OTHER SOLUTIONS

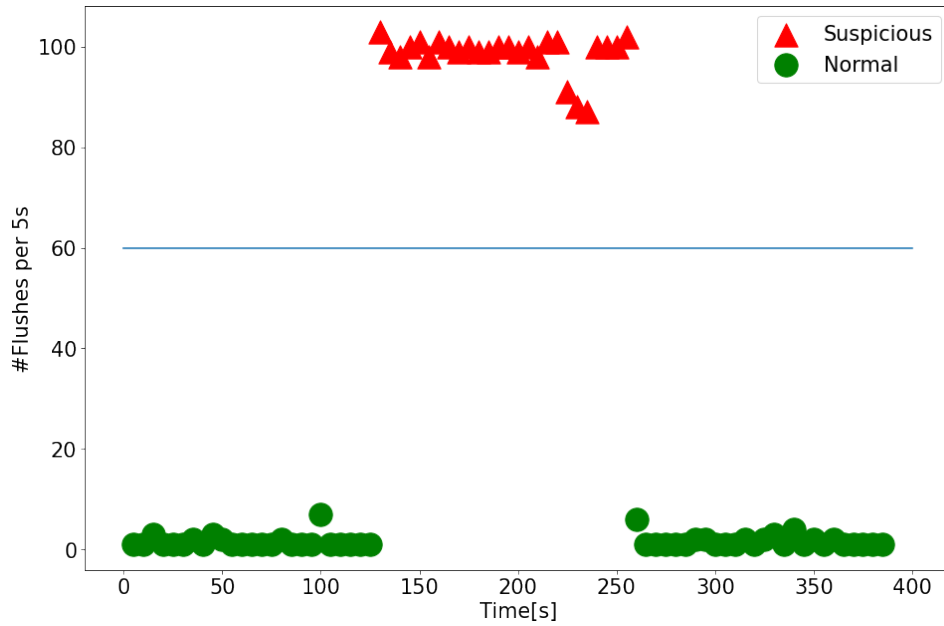


(a) Suspicious behavior in Debian 10.

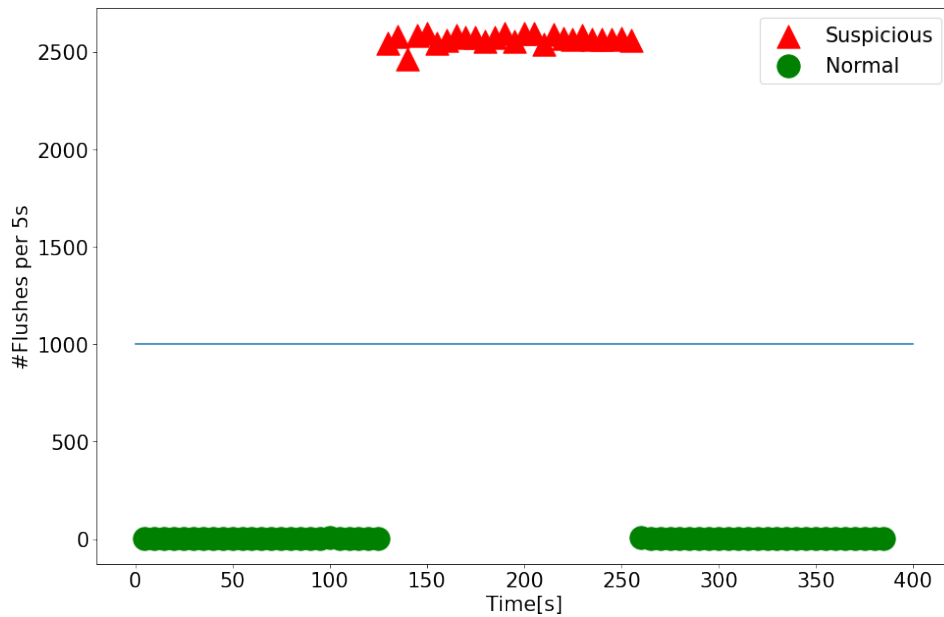


(b) Suspicious behavior in CentOS 8.

Figure 3.6: Number of flushes per 5s for attacks on multiple Application functions(the horizontal axis shows the time in seconds). Different thresholds (1000 flushes for Debian 10, and 10 for CentOS 8) were used to distinguish between normal behaviour (shown as green circles) and suspicious behaviour (shown as red triangles).

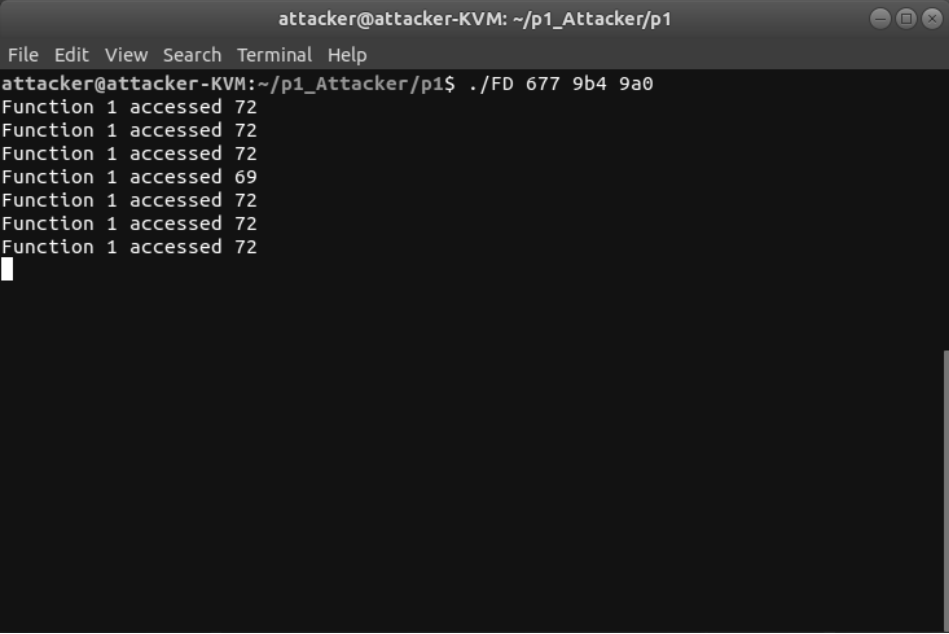


(a) Flush+Reload Attack the GnuPG implementation of RSA.



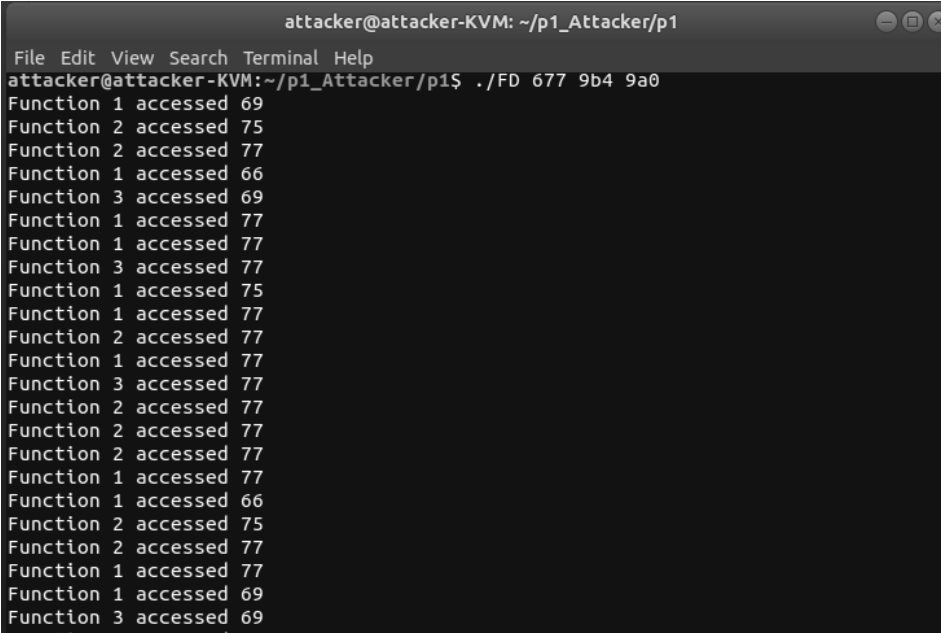
(b) Flush+Flush Attack the GnuPG implementation of RSA.

Figure 3.7: Number of flushes per 5s for attacks on the GnuPG implementation of RSA (the horizontal axis shows the time in seconds). Different thresholds (60 flushes for Flush+Reload Attack, and 1000 for Flush+Flush) were used to distinguish between normal behaviour (shown as green circles) and suspicious behaviour (shown as red triangles).



```
attacker@attacker-KVM: ~/p1_Attacker/p1
File Edit View Search Terminal Help
attacker@attacker-KVM:~/p1_Attacker/p1$ ./FD 677 9b4 9a0
Function 1 accessed 72
Function 1 accessed 72
Function 1 accessed 72
Function 1 accessed 69
Function 1 accessed 72
Function 1 accessed 72
Function 1 accessed 72
```

(a) The results of the attack while the detection mechanism is inactive.



```
attacker@attacker-KVM: ~/p1_Attacker/p1
File Edit View Search Terminal Help
attacker@attacker-KVM:~/p1_Attacker/p1$ ./FD 677 9b4 9a0
Function 1 accessed 69
Function 2 accessed 75
Function 2 accessed 77
Function 1 accessed 66
Function 3 accessed 69
Function 1 accessed 77
Function 1 accessed 77
Function 3 accessed 77
Function 1 accessed 75
Function 1 accessed 77
Function 2 accessed 77
Function 1 accessed 77
Function 3 accessed 77
Function 2 accessed 77
Function 2 accessed 77
Function 2 accessed 77
Function 1 accessed 77
Function 1 accessed 66
Function 2 accessed 75
Function 2 accessed 77
Function 1 accessed 77
Function 1 accessed 69
Function 3 accessed 69
```

(b) The results of the attack while the detection mechanism is active.

Figure 3.8: Obfuscation of attack results. If the detection mechanism is not activated, the results of the attack are precise (function 1 was recorded as used). In contrast, if the mechanism is activated, the attack's results are unclear which of the functions was used by the victim (all functions were recorded as used while just function 1 used).

The results were also recorded in CSV format to be used as a dataset in machine learning algorithms. We created a logistic regression model to support attack detection. We also analyzed the results and measured the accuracy of our detection mechanism. As shown in Table 3.1, our method achieved a mean accuracy of 99% in these experiments. Table 3.1 also shows a comparison of the proposed attack detection method to the methods introduced in previous studies, which are described in Section 2.6 from the thesis. While the CPU usage of our solution (2–8%) is above that of some of the existing approaches, this additional overhead comes with the major advantage that our solution can be deployed and run within the VM requiring protection. Furthermore, this overhead only occurs while our approach is activated during periods when the protected VM performs sensitive operations, which is typically infrequent. In contrast, the other approaches need to be active at all times, since finding out when the victim’s VM performs sensitive operations and requires protection is difficult to know.

Table 3.1 shows the F-score, which is a measure of a machine learning model’s accuracy in predictions. It also is defined as the harmonic mean of the model’s precision and recall. F-score is used to evaluate the model performance. F-score can be calculated as follows:

$$F\text{-score} = \frac{2}{\text{precision}^{-1} + \text{recall}^{-1}} = 2 \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} = \frac{TP}{TP + \frac{1}{2}(FP + FN)}$$

while $TP = TruePositives$, $TN = TrueNegatives$, $FP = FalsePositives$, and $FN = FalseNegatives$.

Table 3.1: Comparison to other approaches

Method	Tool	CPU Usage(%)	Detection Rate	
			Accuracy(%)	F-score
Zhang et al.[25, 30]	Linux perf	5%	-	0.85
Chiappetta et al.[24]	Linux perf	0.6%	-	0.93
Mohammad-Mahdi et al.[23]	Intel CMT	2%	-	0.67
Jonghyeon Cho et al.[88]	Intel PCM	0.9%	97%	-
Gulmezoglu et al.[25]	Intel PCM	18%	99%	0.99
Mushtaq et al.[27]	Intel CMT	4%	99%	-
Our Method	Memory Deduplication	2–8%	99%	0.99

3.5 Limitations

The proposed method provides many important characteristics in detecting and preventing side-channel attacks, but there are a number of limitations that must be taken into consideration to develop the method appropriately. One of the limitations is that it is necessary to use the digging tools to find out the shared memory addresses between the virtual machines and enter them into the proposed mechanism to monitor these addresses, and this process may be somewhat complicated for the ordinary user. The method also recorded a slight increase in the processor overhead, but the mechanism is not activated except when executing sensitive operations such as encryption operations. Additional experiments are required to further evaluate our method's effectiveness in a broader range of scenarios.

3.6 Summary

We introduce a method for protection against a side-channel attack made possible by the use of a cloud-computing feature called *memory deduplication*. Memory deduplication improves the efficiency with which physical memory is used by the virtual machines running on the same server by keeping in memory only one copy of the libraries and other software used by multiple VMs. However, this allows an attacker's VM to find out the memory locations (and thus the operations) used by a victim's VM, as these locations are cached and can be accessed faster than memory locations not used by the victim.

To perform the attack, the malicious VM needs to execute an abnormal sequence of cache flushes, and our new method detects this by monitoring memory locations associated with sensitive (e.g., encryption) operations and using logistic regression to identify the abnormal cached operations. Furthermore, our method disrupts the side channel, making it more difficult for the attacker to acquire useful information. The experiments we ran using the KVM hypervisor and Ubuntu 18.04 LTS VMs on both Debian 10 and CentOS physical servers show that our method can detect attacks with 99% accuracy, and can feed fake information to an attacker with between 2–8% CPU overheads.

We proposed a flush-based attack detection mechanism that works inside the protected VM without additional requirements to provide readings of a shared executable program's performance between VMs. The experimental results indicate that method works with a mean accuracy of 99% if a suitable threshold is set to determine the attack status. The method helps VMs to detect an attack by knowing the attack readings, thus providing self-protection for the VM. This differs from previous solutions, which need to run in the host machine.

The solution proposed in this chapter answers research questions 1 and 2 from Section 1.3 in the thesis, confirming that it is possible to exploit memory deduplication as a protective factor to detect and protect VM shared sensitive executable files by sensing cache instability during sensitive processes such as encryption processes. Also, continuous monitoring of the shared cache locations is enough to confuse the attack results due to the constant access to the shared cache locations to measure data retrieval time. Furthermore, the experiments carried out to validate this solution confirm hypotheses H1 and H2 from Section 1.4.

MITIGATION THROUGH DYNAMIC AND STATIC ANALYSIS

We design in this chapter a method for detecting and protecting against side-channel attacks. The method integrates dynamic and static analysis. The dynamic analysis uses *Linux Perf* to acquire runtime readings from 13 hardware counters associated with the shared cache. The logistic regression classification algorithm is then used to classify the VM behaviour as suspicious or benign based on these counters. Static analysis is then started to extract executable files from the suspicious VM's RAM and disk image. It then determines whether they contain side-channel attack operation codes. This information is used to determine the threat level of these files via the SoftMax classification method; there are a total of four threat levels. Following that, the VM that represents a security risk to the shared environment is excluded.

4.1 Introduction

This chapter focuses on a set of cache-targeting threats in the shared virtualized environment called cache side-channel attacks. Cache side-channel attacks require monitoring the cache hit and miss for a specific cache line while retrieving data, thus realizing whether the monitored cache lines have been recently used or not, which leads to cracking the encryption keys of the shared encryption libraries and exposing confidential data.

Although several mitigation approaches for this sort of cache attacks have been presented [26, 64, 88, 89, 104–107], a variety of flaws exist in these approaches. First, there is no a fundamental precautionary action when detecting malicious executable files, especially when performing static analysis because it takes longer. Some of these solutions do not have a signal to start as a reason for performing static analysis, which generally causes significant overload of the system. Additionally, most of the prior methods are required to enhance performance and detection results. Finally, none of the most widely used antivirus software detects any of the threats we examined (Cache side-channel attacks) for protecting shared virtualised environments [26, 64].

We introduce a method that combines dynamic and static analysis to detect and defend

against this type of attack. The dynamic analysis approach monitors the activities of VMs. It detects suspicious activities that indicate the presence of cache side-channel attacks by extracting readings from hardware performance counters using Linux *Perf* and classifying them using logistic regression to determine the status of the attack or not. If suspicious activity is detected for one of the VMs, this is considered the starting signal for operating the static analysis of the executable files of the suspicious VM detected. The virtual machine's executable files are accessed, disassembled, and analysed whether they contain implicit characteristics of the cache attacks or contain the operation codes of the attacks. The threat level of these files is then determined using a neural network classifier that uses the SoftMax algorithm for classification. The following are the primary contributions of this chapter:

1. We introduce an approach for detecting and protecting shared virtualized environments against cache side-channel attacks using dynamic and static analysis.
2. Mechanism design, implementation and experimentation.
3. We evaluate the approach in various attacks scenarios in terms of detection efficiency and performance attributes.

The remainder of the chapter is arranged in the following manner. Section 4.2 presents the required tools. Section 4.3 illustrates the proposed detection and protection method. Section 4.4 describes the experiments we conducted using the proposed method. In Section 4.5, we discuss the evaluation of the implemented method. In section 4.6 provides an overview of limitations related to our method . Finally, Section 4.7 provides the conclusion.

4.2 Required Tools

This section reviews the tools required to carry out the experiments and implement access to the RAM image of the VMs, and the analysis processes.

4.2.1 Libguestfs library

We have applied various tools to support and facilitate completing static analysis, and we must use software and tools to enable us to access the VMs and extract files for analysis. When we create a guest VM on a hypervisor such as Qemu-Kvm, we create a qcow2-type disk image as a virtual disk volume assigned to the guest VM. We could mount the disk image of a VM if we needed to examine or alter files on it. We would be able to change and inspect the disk image's content before unmounting it, so we utilized the Libguestfs library, a set of open-source tools to access the VM disk images to view the disk images or modify files inside the VM [108].

Libguestfs supports almost all sorts of disk images, including qcow2 disk images. Libguestfs toolset can be installed on Linux, Windows, and Mac. Moreover, It is compatible with other hypervisor systems like VMware, VirtualBox and Hyper-V. We can mount the VMs' disk image in a read-write or read-only mode according to our needs and purpose. When the libguestfs toolset is installed, we can access the qcow2 disk image of a VM using *guestmount* command-line. Also, we can define the image path, a mount point, and mounting mode [109].

In our project, we used QEMU/KVM as a hypervisor to form a shared virtualized environment, so it was the type of disk used qcow2. We also used Linux command lines to mount the required disk image, and these commands were executed using C. We created the mount point, identified the disk image path, and mounted the disk; then, we were able to extract executable files using *Find* command in Linux. After completing the static analysis, we unmounted the disk using *guestunmount* command line.

4.2.2 AVML Tools and Volatility Framework

It was essential to combine memory analysis and VMs' disk analysis in our method to provide comprehensive analysis for suspicious VMs. We utilized Acquire Volatile Memory Linux (AVML) and Volatility Framework to perform memory acquisition and memory dump analysis for the required VM in the shared virtualized environment. In this process, we know the executable

files that have been mapped into the RAM; hence, they have been used and operated recently.

AVML is an open-source volatile memory acquisition tool produced by Microsoft, and it was created with Rust programming language, and it supports almost all Linux distributions. We applied the AVML to regularly capture VM's memory image to analyze the memory image using the Volatility Framework to find out the executable files recently stored in the RAM [110].

Analyzing the memory images assists us in finding malware. In our project, we regularly create memory image files. These files contain information that can be useful in determining malicious VMs in the virtualized environments. We conducted a memory image analysis using the Volatility Framework.

The Volatility Framework is a set of open-source memory forensics tools implemented in Python. It is capable of analysing Windows, Linux, Mac, and Android operating systems. The Volatility Framework is beneficial in reconstructing the activities that the suspicious VM performed and identifying the running malicious binaries. Moreover, it provides a set of scanner plugins that improve the RAM dump analysis [111].

Using the analysis tools mentioned above, we can retrieve significant information that supports us in discovering the running and recently terminated processes, files mapped in the memory, command lines history, etc. We have benefited from these tools in extracting executable files, which have recently been activated, instead of analyzing a wide range of executable files as in the VMs' disk analysis.

4.3 Method

The proposed detection mechanism performs several successive operations and stages, as shown in Figure 4.1.

1. In the first stage, the proposed mechanism receives the process ID of the VM. It utilizes

Linux Perf to obtain the performance readings from the performance counters, such as instruction per cycle, L1, L2, and L3 cache miss counters. The appropriate threshold is measured and determined in advance during analyzing the side-channel attacks at the time of execution their impacts on hardware performance counters. This stage is the condition for starting static analysis.

2. In this stage, we create and train a logistic regression model that processes the data extracted from the performance counters and acts as a classifier between suspicious and normal behaviours.
3. In this stage, we access the virtual machine disk image on the KVM host by using the Libguestfs tools [108] that can also be used in popular hypervisors such as VMware and Hyper-V.
4. We then extract the executable files from the virtual machine disk and store them into a file to be checked in the stage number 6. In this stage we capture the VM RAM periodically using AVML Framework [110] then analyze them from within the host using the Volatility Framework [111] to extract the files that have been stored in the RAM, thus knowing which files that have been used recently. We then filter these files to reduce the number of files extracted from the disk to be processed later. This step is optional if you want to analyse the disk image of the VM only.
5. In this stage, we disassemble the extracted executable files using an open-source static binary analyser called the Radare2 and the Objdump command in Linux to facilitate executable file analysing.
6. Then we examine the extracted files against the implicit characteristics of the cache attacks codes. We utilise the R2pipe API tool [112] to automate reverse engineering of the executable files using Linux command lines. The implicit characteristics of the cache attacks codes are *clflush*, *rdtsc*, *mfence*, and *lfence*, as mentioned in [4, 26, 64]. The result of this stage will be a dataset for the next stage. The result is organized in columns as follows: File path, *clflush*, *rdtsc*, *mfence*, *lfence*, and *ThreatLevel*.

Table 4.1: Score-based threat classification

Input	Characteristics				Class
	<i>Clflush</i>	<i>Rdtsc</i>	<i>Mfence</i>	<i>Lfence</i>	
1	X	X	X	X	Green
2	X	X	X	✓	
3	X	X	✓	X	
4	X	X	✓	✓	
5	X	✓	X	X	Yellow
6	X	✓	X	✓	
7	X	✓	✓	X	
8	X	✓	✓	✓	
9	✓	X	X	X	Orange
10	✓	X	X	✓	
11	✓	X	✓	X	
12	✓	X	✓	✓	
13	✓	✓	X	X	Red
14	✓	✓	X	✓	
15	✓	✓	✓	X	
16	✓	✓	✓	✓	

The value of 1 indicates the existence of the implicit characteristic, otherwise it is 0 for each column. The threat level can take a value from 0 – 3 according to the variety of characteristics found. For example, we understand memory barrier (*mfence*, and *lfence*) are not a threat if they are not issued together with *rdtsc* timers or *clflush* eviction instructions. We have organised the threat classes, as shown in Table 4.1 which determines the threat level based on the combination of characteristics in the executable file, so that the (X) sign indicates that the corresponding characteristic is not existing, and the (✓) sign indicates its existence.

- Red: This level is considered the maximum threat and expresses the presence of all the implicit characteristics and dangerous instructions in malicious codes to launch side-channel attacks.
 - Orange: This level is considered a high threat and expresses the presence of two implicit characteristics (*clflush* and *rdtsc*) this may include having only one of the memory barrier instructions (*mfence*, and *lfence*). These dangerous instructions have enough ability to launch side-channel attacks.
 - Yellow: This level is considered a low threat and expresses the presence of only one of the implicit characteristics of (*clflush* or *rdtsc*); this also may include having only one of the memory barrier instructions (*mfence*, and *lfence*). Having just one of these dangerous commands is insufficient to launch cache side-channel attacks.
 - Green: This level represents the nonexistence of a threat and indicates the absence of any dangerous instructions. At the same time, it may include memory barrier instructions (*mfence*, and *lfence*) that do not represent a threat.
7. We create and train a neural network model using the SoftMax algorithm to classify executable files based on the previously mentioned threat levels 4.4.

4.4 Experimental Results

We executed experiments using the QEMU-KVM hypervisor. We used the QEMU-KVM hypervisor on various hosts; Ubuntu 18.04.5 LTS with Intel Core i5-4200M CPU, Debian 10 with Intel Core i5-4200U CPU, and CentOS 8 with Intel Core i5-5300U CPU. We then created two VMs running a Linux Ubuntu 18.04.5 LTS operating system, one VM running as a victim and the other running as an attacker VM. We installed the Libguestfs Tool [108], the Linux Objdump Disassembler, and Volatility Tools [111] inside the host. For the guests, we installed AVML (acquire volatile memory for Linux) [110] to capture the RAM regularly. We designed the shared virtualized environment utilizing the QEMU-KVM hypervisor's default settings, as

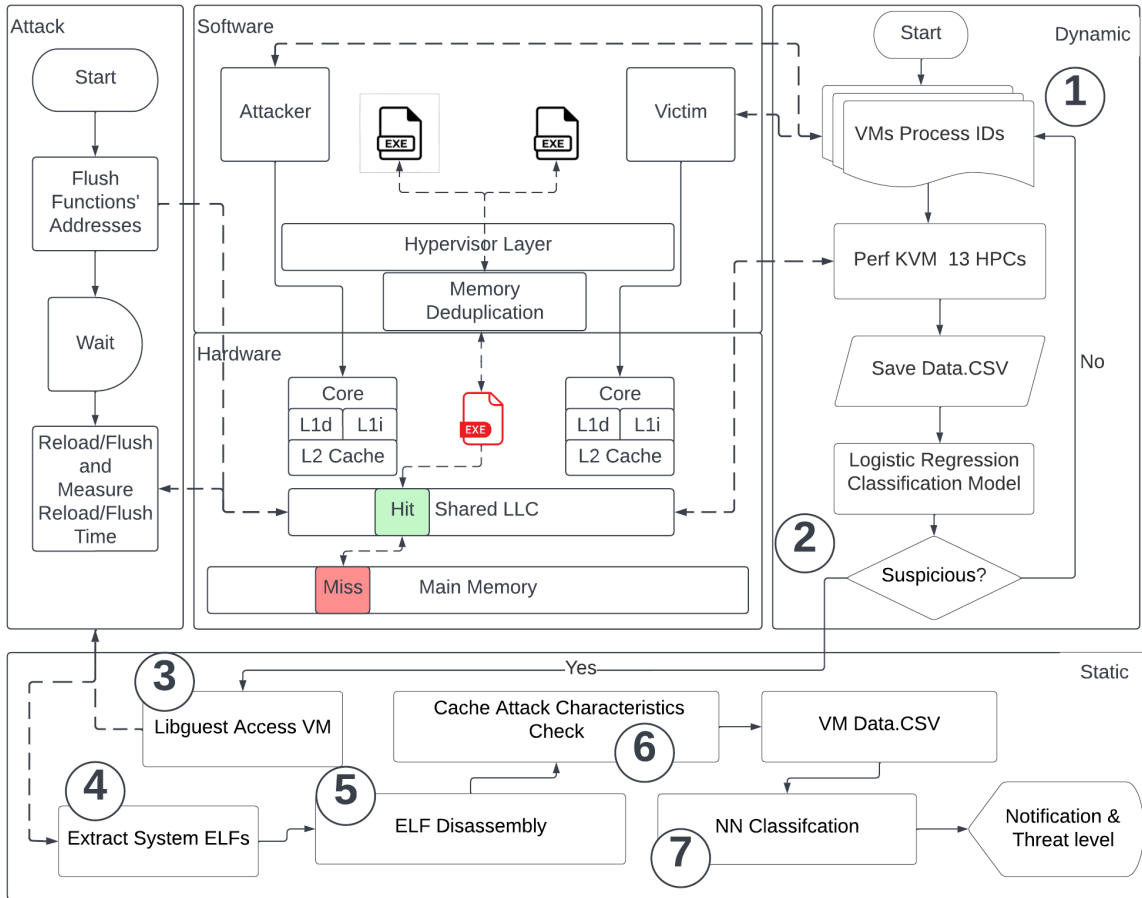


Figure 4.1: Detection Stages

mentioned in section 3.3.2 in the previous chapter we mean by default settings that KSM was enabled on the QEMU-KVM hypervisor, allowing the hypervisor to share executable files and applications across multiple VMs and processes and no additional security software was installed. We then assigned the victims and the attackers' VMs to different CPU cores, meaning the VMs shared the LLC cache. Thus, The VMs were not able to share the other levels of cache among the VMs.

The experience consisted of two main parts. At first, we conducted an experiment to monitor suspicious behaviours that lead and hint at an attack within the virtualized environment. The second part was to perform a static analysis of the suspicious VM to detect whether there was really a threat from the VM and to take the necessary action. These combined operations

reduce the fault rate in the detection process.

4.4.1 Monitoring Suspicious Behaviours

This section demonstrates the experiment to monitor effective performance counters related to a cache for detecting cache attacks in shared virtualized environments which are Flush+Reload, Flush+Flush, and Prime+Probe attacks. We monitored a set of performance counters and noticed that they constantly changed as we performed cache side-channel attacks. The Mastik framework was used to carry out the various cache attacks. Also, we used the Linux Perf tool to extract CPU performance counters readings. We executed the experiments with four scenarios. The first scenario is normal activities; We took the observations without any attacks or noise. In the second scenario, the observations were taken with the normal activities with noise of background applications; no attacks were performed in this scenario. To clarify, we mean by "Normal Activity", which is the state of the machine without any programs or applications running on the virtual machine, while by "Noise", we mean the state of the machine while programs and applications are running in the background of the virtual machine. In the third scenario, we fetched the readings from the performance counters during the execution of the cache attacks without any interference from background applications. Finally, we carried out the attacks with several applications; the readings were observed in this scenario as well.

Our experiment focused on 13 cache-related counters as shown in Table 4.2 to record observations in various scenarios every 15 seconds. The first scenario is obtaining the CPU's performance counters readings in the virtual machine's normal state without any activity; secondly, getting the readings while performing activities such as running applications on the virtual machine only without any attack. Third, taking observations during the execution of the side-channel attack only, without any noise from the background applications and finally, fetching the readings from the performance counters of the CPU during the execution of the side channel attacks with the presence of noise issued by the background applications. This set of performance counters was chosen because it changed significantly in the experimental

Algorithm 3 Pseudo-code of the Dynamic Analysis

```

Input: VM Process ID
Output: Malicious=1 or Benign=0
1: Receive vm process id
2: Model = logistic regression()
3: for Infinite Loop do
4:   Pause for 15 seconds
5:   Run perf kvm stat -o output.txt -e event counters
6:   Output = open and read output.txt
7:   while fgets(line,size,Output) != Null do
8:     Convert counters readings to integers
9:     Save counters readings to file.csv
10:  end while
11:  Data = open and read file.csv
12:  Result = Model ← Data
13:  if Result > 0 then
14:    Detect malicious behavior
15:    Stop kvm
16:    Start static analysis
17:  end if
18: end for

```

scenarios and to provided sufficient readings in the case of machine training. We implemented training stage by utilising the hardware performance counters gathered by the Linux Perf tool. We gathered the training data from various scenarios because the performance counters were affected by other programs operating on the VMs. The data was acquired while implementing cache attacks, both with and without running multiple background programs. About 3610 samples were collected. Also, a binary classification model was created using logistic regression to analyse the input data and categorize whether the system was under attack or not; 0 indicated a no attack state and 1 indicated an attack state.

We plotted the readings for the various aforementioned different scenarios. Figures 4.3 and 4.2 show the clear variance in cache misses counters and instructions counters readings in all scenarios. Also, all performance counters were analysed using machine learning to analyse the data more accurately and efficiently. We implemented the Logistic Regression classification model to analyse and classify the data, whether the extracted data indicated the presence of an attack or not, as this is the first step of the proposed detection process. The model was trained on 70% of the samples collected and tested on 30% of the samples. The model showed about 99% accuracy for the test case. The accuracy is an essential metric for evaluating the model performance of predictions, and it means how many times the model predicts the

Table 4.2: Intel Hardware Performance Events [2, 3]

Performance Counter	Counter Description
mem_load_uops_retired.l1_miss	This event counts retired load uops which data sources were misses in the nearest-level (L1) cache.
mem_load_uops_retired.l2_miss	This event counts retired load uops which data sources were misses in the mid-level (L2) cache.
mem_load_uops_retired.l3_miss	This event counts retired load uops which data sources were misses in the last-level (L3) cache.
br_inst_exec.all_branches	This event counts both taken and not taken speculative and retired branch instructions.
instructions	This event counts number of instructions.
L1-dcache-load-misses	Level 1 cache for data (L1d) read misses by a CPU core.
L1-icache-load-misses	Level 1 cache for instructions (L1i) read misses by a CPU core.
LLC-load-misses	Last level cache (LLC) read misses by a CPU core.
LLC-loads	Last level cache (LLC) read accesses by a CPU core.
cache-misses	This event counts number of memory access that could not be served by any of the cache.
cache-references	Cache accesses per CPU core.
iTLB-load-misses	Translation lookaside buffer for instructions (iTLB) read misses by a CPU core.
iTLB-loads	Translation lookaside buffer for instructions (iTLB) read accesses by a CPU core.

correct label, i.e. true positive. The accuracy can be calculated as follow:

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

while $TP = TruePositives$, $TN = TrueNegatives$, $FP = FalsePositives$, and $FN = FalseNegatives$. We used `confusion_matrix` from `sklearn.metrics` to assess prediction performance. We recorded the false positives and negatives as shown in the Figure 4.9.

To carry out the experiment, we first extracted the process IDs of the virtual machines using the Linux command line. We then used the process IDs to monitor the performance counters of the virtual machines using the Linux Perf tools in different scenarios. Then we use the binary classification model. If there was suspicious behaviour from a VM, we proceeded to the next step, which was the static analysis of the virtual machine.

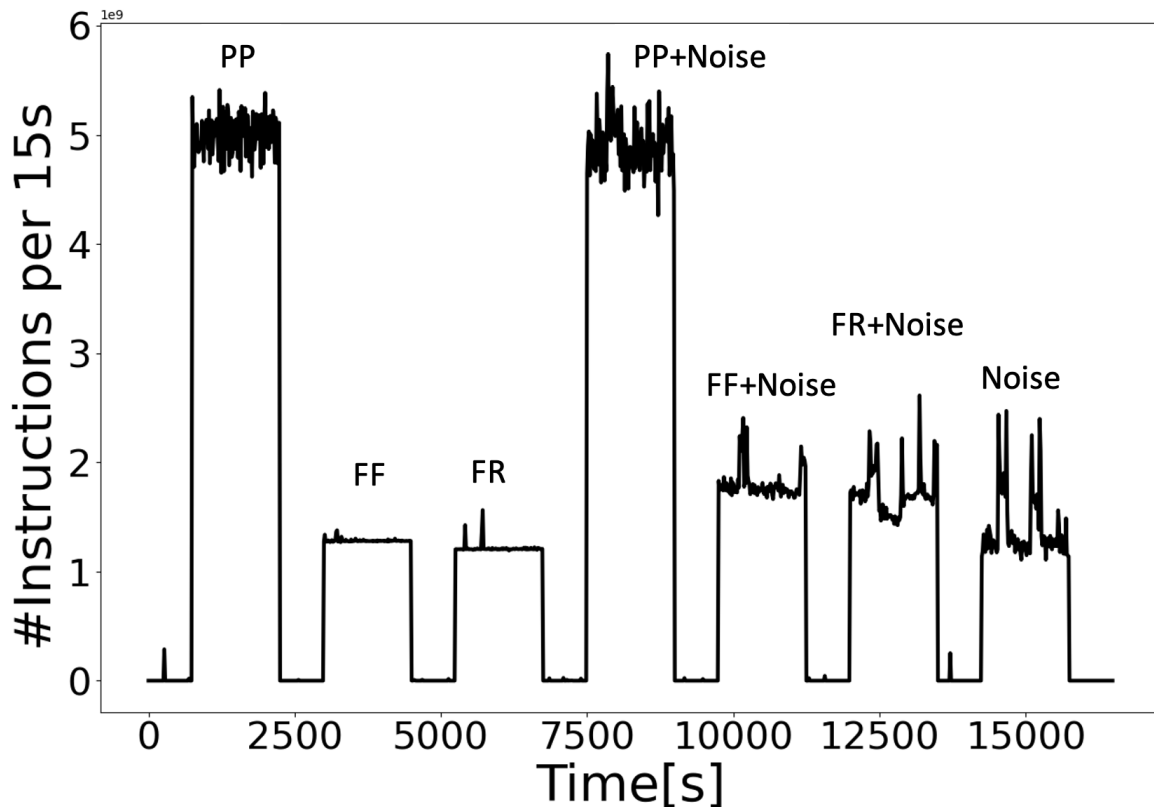


Figure 4.2: Experiment scenarios of all cache side-channel attacks in two cases without noise attack and with background noise. PP represents prime+probe, FF represents Flush+Flush, and FR represents Flush+Reload.

4.4.2 Static Analysis for VMs

This section discusses how the experiment was carried out for the static analysis of VMs that had suspicious behaviour within the shared virtualized environment. We downloaded and installed the aforementioned required programs for analysis on the host and the VMs.

Our analysis involved the Mastik tool designed by Yarom et al.[100], Xlate [113], Cache Template Attack source code represented by Gruss et al.[114], and the Flush + Flush attack tool [115], as well as other Github repositories inspired by "Cache Template Attacks"[57] and "FLUSH+RELOAD: a High Resolution, Low Noise, L3 Cache Side-Channel Attack" [4] such as [116], [117], [118], [119], and [120].

We implemented virtual machines static analysis experiments in two scenarios, as described

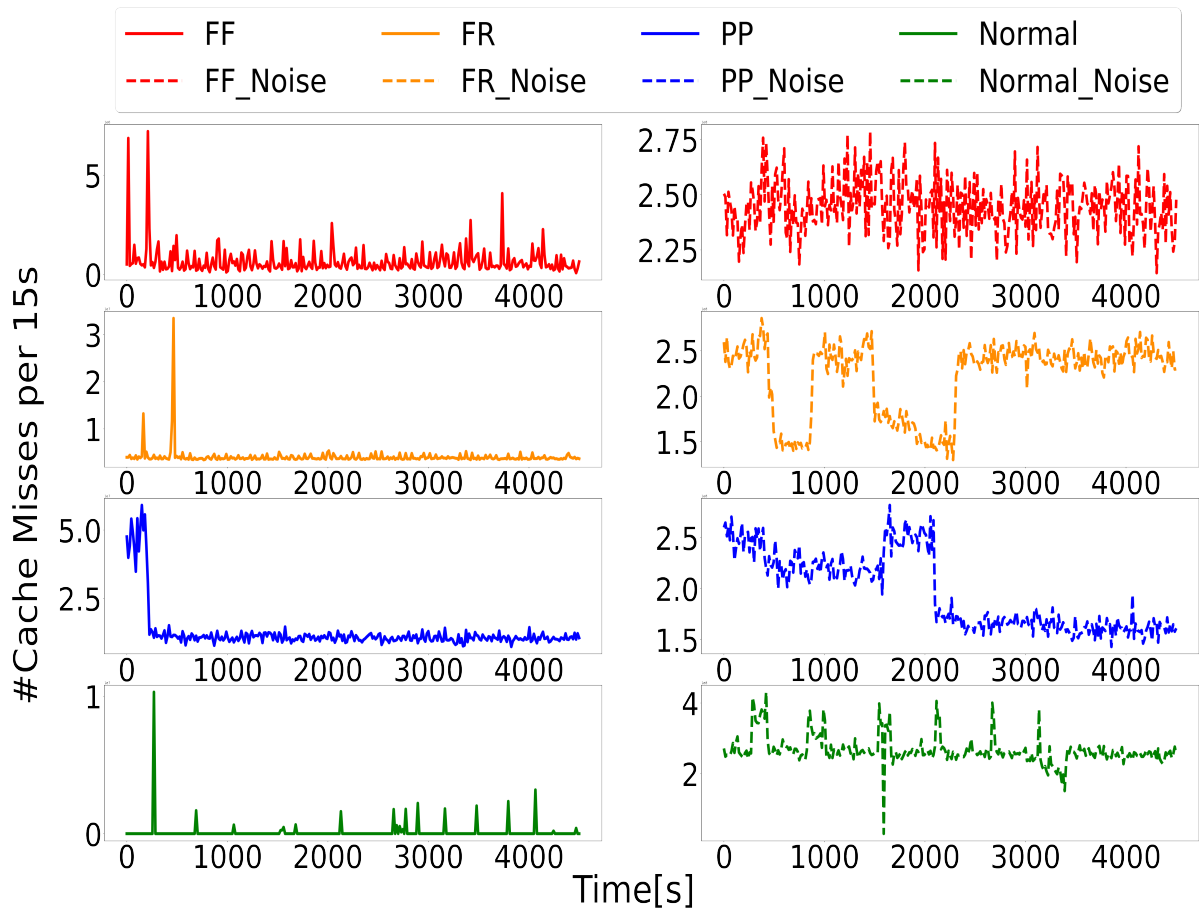


Figure 4.3: Number of cache misses per 15s for different attack scenarios.

below:

1. VM Disk Image Analysis: In this scenario, our proposed mechanism received the name of the VM. It created a mount point for the VM using libguestfs tools [108], which allowed access to the guest's filesystem from the host and extracted files through it so that the VM remained operating and we did not need to turn off the VM to access the VM's disk image. After that, we extracted the executable files in both user and kernel space files using the command lines that were implemented using the C language and stored the resulting files' paths in a *.txt* file. We then used the Linux Objdump tool to disassemble the executable files and display the opcodes of the executable files in the assembly language. Next, we made sure of the presence of the attack opcodes that had implicit cache side-channel attacks characteristics inside the executable files,

such as the flush commands *clflush* and the time stamp counter commands *rdtsc*, which provide the attacker with accurate measurements about execution times and knowledge of the time the the contrast between the cache hits and the cache misses, as they are commands that represent a threat to shared virtualized environments. Then we recorded the results of each extracted executable file, then we stored the results to be processed in the neural network model that depending on the SoftMax classification to determine the threat level of these files.

2. VM RAM Image Analysis: In this scenario, we analysed the VM's RAM image. We performed this experiment in several steps. First, we downloaded and installed the AVML (Acquire Volatile Memory for Linux)[110] to periodically capture the VM's RAM image. In addition, we have downloaded and installed the Volatility Framework [111] to support analysing RAM images. Using these tools, our method was able to capture images of the RAM periodically, making it possible to track the operations of the VM. It was constantly updated during the operation of the VM. From the host device we were able to process and analyse the RAM and recognize the files stored in the RAM, making our method more effective in detecting an attack, thus protecting the virtualized environment and getting rid of malicious VMs. We accessed the VM that had suspicious behaviour inside the shared virtualized environment using the libguestfs tools [108]. We then extracted the executable files the same as we performed in the first scenario, after which we accessed the RAM image produced from the AVML [110] periodical captures and extracted the executable files. We then filtered the files by comparing the files' paths elicited from the disk image and the files' paths elicited from the RAM image. We implemented files filtration in this approach to obtain sufficient information about the recently executed files and reduce the large number of files extracted from the disk image. We then analysed the filtered executable files using the Linux Objdump tool to convert the executable files into an assembly language, making it straightforward to investigate the presence of implicit attributes that comprise a threat on the system. We then stored the results to be classified later using the neural network model.

Algorithm 4 Pseudo-code of the Static Analysis

```
Input: VM Name and D=Disk or M=Memory
Output: Threat_level
1: Enter VM Name and d or m
2: Model = Neural Network Classifier()
3: if argv = "d" then
4:   guestmount vm disk image
5:   find ELF files path → ELFs.txt
6: else if argv = "m" then
7:   guestmount vm disk image
8:   find ELF files path → ELFs.txt
9:   find vm memory image → Image.raw
10:  volatility linux_elfs → ELFs.txt
11:  filter disk_elfs and memory_elfs → ELFs.txt
12: end if
13: while fgets(line,size,ELFs) != Null do
14:   objdump ELF file path (line) → objdump.txt
15:   while fgets(line,size,objdump) != Null do
16:     if strstr(line = clflush) then
17:       flush = 1
18:     else if strstr(line = rdtsc) then
19:       rdtsc = 1
20:     else if strstr(line = mfence) then
21:       mfence = 1
22:     else if strstr(line = lfence) then
23:       lfence = 1
24:     end if
25:     write result → result.csv
26:   end while
27:   Result = open and read result.csv
28:   Threat_level = Model ← Result
29: end while
```

The complete architecture of our neural network model for the attack detection is shown in Figure 4.4. The model contains 3 layers; the first layer, which is the input layer, contains 4 nodes $\{x_1 \sim x_4\}$ and receives either zero or one input for each node, depending on the presence of attack characteristics in the executable files. Then the hidden layer, which contains 16 nodes that were identified by a number of attempts to expand and deepen the hidden layer to get the best results in terms of accuracy. Finally, the output layer contains 4 nodes, which express the threat level, and the outputs are one of the following colours $\{Green, Yellow, Orange, Red\}$. To accomplish and build this neural network model, we used the Python language, and the TensorFlow library was used to adjust the settings of the model and create the layers. Then, we trained the model after collecting the training data from a different set of virtual machines in cases of the presence of the attack and in cases where there are no attack files inside these devices. A label was set for each sample to train the model to classify among the four mentioned categories. We collected the dataset to train and test the model by creating a

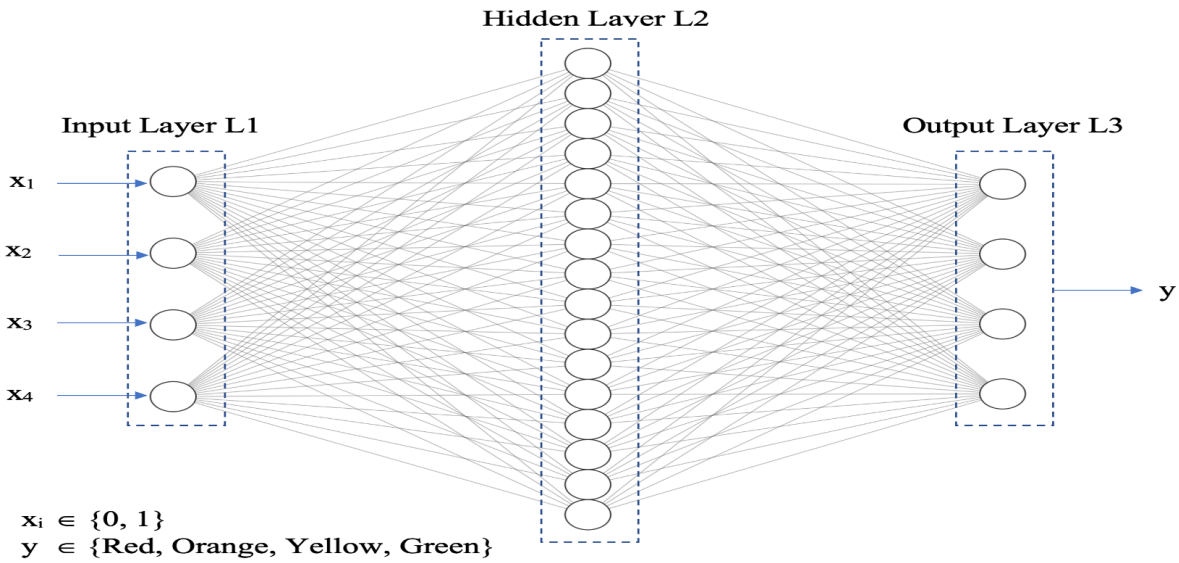


Figure 4.4: Neural Network Model

set of virtual machines, extracting executable files, and analysing them to record the implicit characteristics of these files. About 4,500 data samples were collected in two scenarios: the case of no attack files and the presence of a set of attack files. We used TensorFlow, a Google-provided open-source framework for machine learning methods, to develop the Softmax classification model. We configured the training settings with 200 epochs and 10 as the batch size. In addition, we applied the Adam optimization algorithm to adjust the weights and minimize the loss. Furthermore, we included early stopping in our training through a callback. The model was used as a classifier to classify the executable files according to the threat levels, and The model's accuracy ranged from 96% to 99%. We plotted the validation and training loss, as shown in Figure 4.10.

4.4.3 Expansion of the Static Analysis for VMs

Because several benign system files were classified as a threat because they carry some implicit characteristics of side-channel attacks as shown in Figure 4.6a, we proposed adding additional implicit characteristics up to eight characteristics to be analysed accurately, as the map function was added, loop analysis and analysing instructions inside the loop. Thus, The neural network classification model has been expanded to receive 8 inputs and 16 nodes in

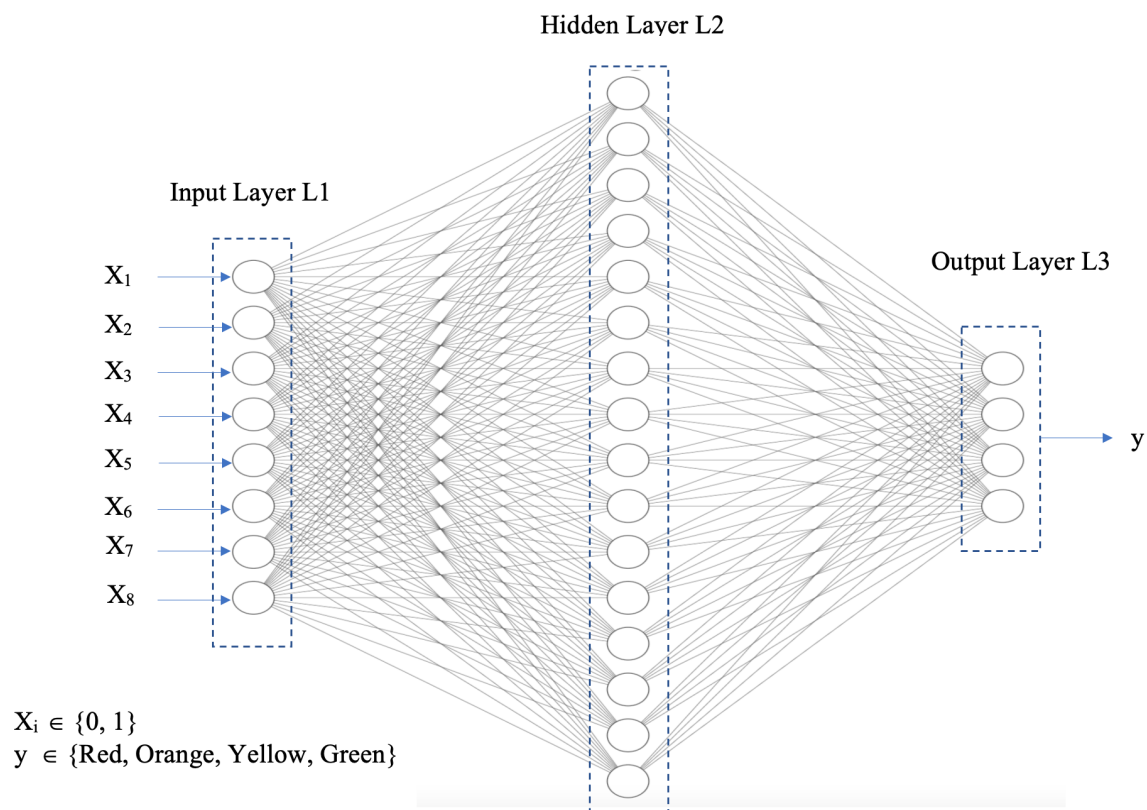


Figure 4.5: Expanded Neural Network Model

the hidden layer, then 4 possible results represent the threat level, as shown in Figure 4.5.

Figure 4.6 represents the results of classifying a benign virtual machine that does not contain any files that threaten the system. The results appear more accurate in the neural networks model that receives eight inputs from implicit characteristics compared with the four input neural networks model. Also, Figure 4.7 represents the results of classifying a virtual machine that contains high-threat files. The results of the eight input model shows more accurate output in detecting malicious files. In Figure 4.8 We plotted the validation loss, which is a metric used to evaluate the performance of a machine learning model on the validation set (a part of the dataset for validation), and the accuracy (number of correct predictions out of the total number of predictions) of the neural network classification model in this experiment.

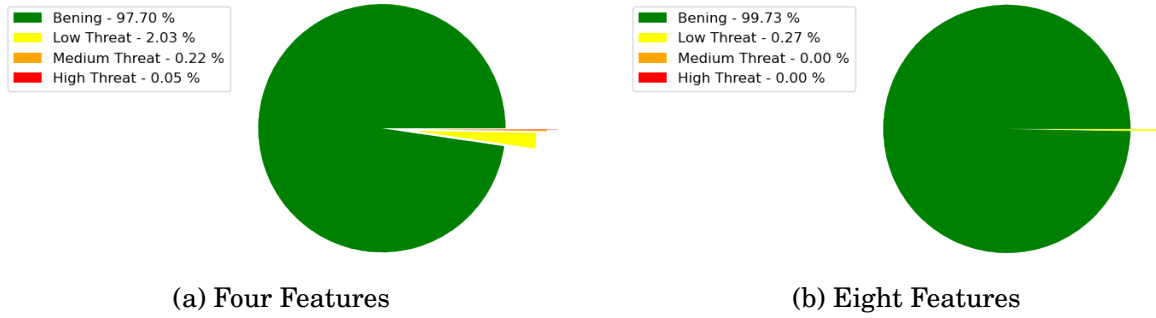


Figure 4.6: Neural Network Classification of Benign VM

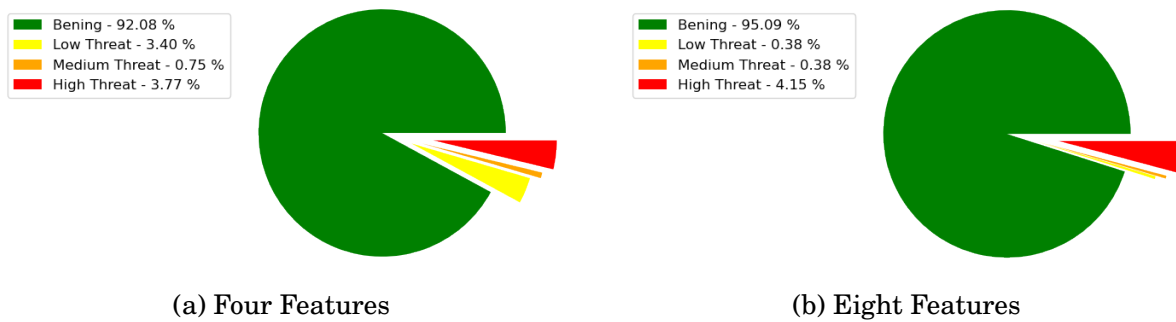


Figure 4.7: Neural Network Classification of Malicious VM

4.5 Evaluation

This section discusses a series of experiments conducted to assess and verify the efficacy of our method in detecting and protecting against cache attacks in virtualized environments. In particular, we evaluate the proposed method in terms of its ability to detect the attacks in different cases using a machine learning classifier and then determine the threat level of executable files for protection. To evaluate the performance of our method in detecting and protecting against the attacks, we established the shared virtualized environments explained in the previous section. The environments' hypervisor was QEMU-KVM, while the guests' operating systems were Linux Ubuntu 18.04.5 LTS. We performed experiments on three hosts, Ubuntu 18.04.5 LTS, Debian 10, and CentOS 8, with various processor models.

We evaluated the accuracy of detecting suspicious behaviours resulting from launching side channels and measured in different cases with a noise background from running several applications during the attacks. In the dynamic analysis, we produced the readings from the

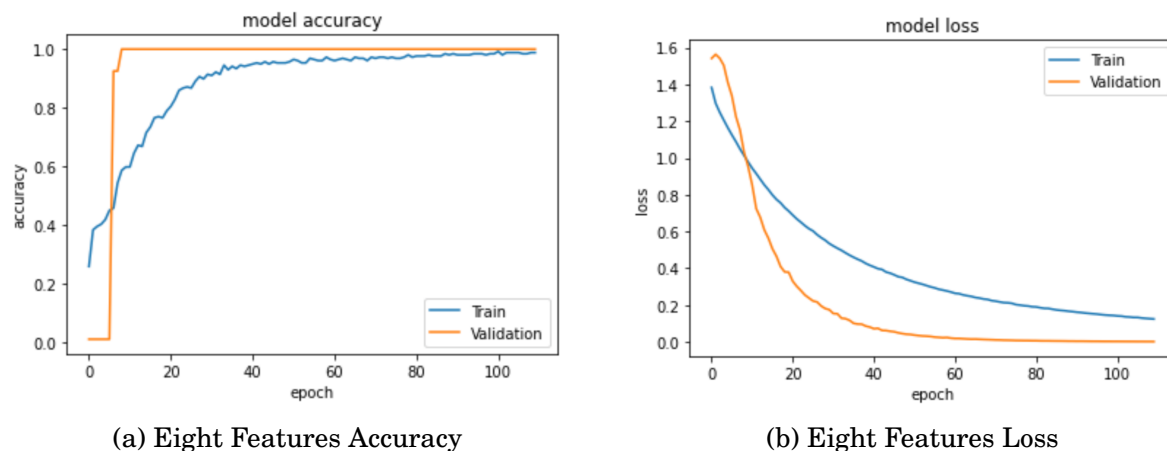


Figure 4.8: Validation Loss and Accuracy of the Neural Network Classification Model

system every 15 minutes to classify them based on whether they indicated the presence of a side-channel attack. We then measured the classification accuracy while using the Linux Top tool to check the CPU usage and the system load during operations.

We tested 200 samples for each host involving various side-channels attacks. The system counters readings were extracted by the Linux Perf to be classified. We then classified these samples using the logistic regression classification. There were simple differences in the accuracy of detecting suspicious behaviour ranging from 96% to 99%. The CPU usage was between 0.11, 0.44 and the load on the system ranged from 0.6 to 3 for all host devices in the dynamic analysis, as shown in the Table 4.3.

In cases of positive results, whether true or false, in dynamic analysis of suspicious behaviours, we moved to the static analysis that extracted and analysed executable files to validate the results. They performed their part in classifying a threat level for these files, thus the threat level of the VM. Suppose we received a false positive result in the dynamic analysis; we would move to the static analysis. The executable files would then be analysed and classified; if it was not a malicious VM then the VM files would be classified as green, meaning they do not pose any threat to the shared virtualized environments. In this process, the results are supported and the detection system's accuracy as a whole increases so that the final results are almost unaffected by faults that may stem from the dynamic analysis.

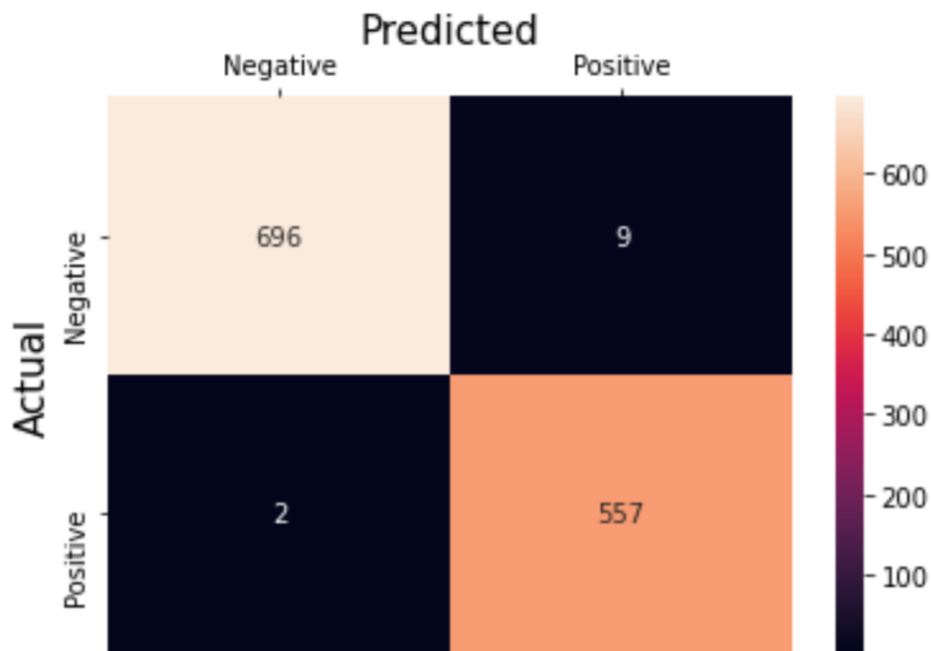


Figure 4.9: Logistic Regression False Positive and Negative

We then conducted several experiments to measure the accuracy of the static analysis and measured the CPU usage and the load on the system. Different host operating systems were used. In each of them, we created a set of virtual machines by downloading and installing a set of side-channel attacks tools and executable files as we described in section 5. Attack files included a collection of attacks framework tools such as the Mastik framework, Xlate framework, and other attack project source codes. After that, we performed the static detection in two scenarios, the desk analysis, and the memory analysis. As shown in the Table 4.3, the method detected attacks on executable files with 97-98% accuracy, with between 10–25% CPU usage and between 0.85 –1.46 system overhead.

The results extracted from the static analysis of the disk were a statement of a set of executable files that carried at least one of the implicit characteristics of side-channel attacks, after which the extent of the threat level to these files was classified using the SoftMax algorithm used in the neural network classifier, regardless of whether these executable files had been recently run on the systems or not. In the constant analysis of the memory, all files extracted as results had recently been run on the system, which is why they were mapped

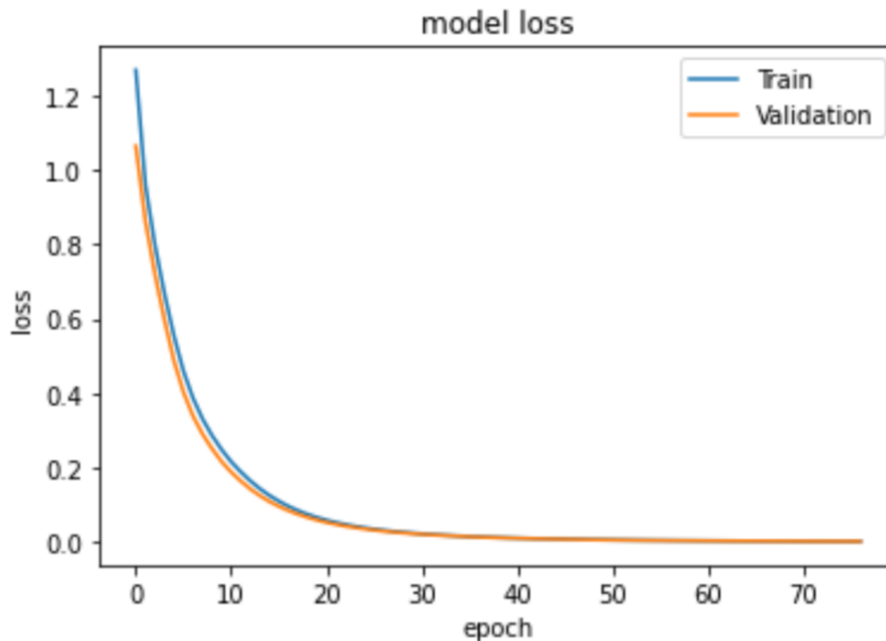


Figure 4.10: Validation and Training loss

into the memory. These executable files also carried at least one of the implicit characteristics of side-channel attacks.

Table 4.3 shows the measurements of the system load, the CPU usage, and the accuracy of detection. We mean by the system load, the average number of processes being executed or waiting to be executed by the CPU over the past 5 minutes. As for the CPU usage, it means the percentage of the CPU that was used out of the total CPU capacity during the execution time. Regarding the accuracy, it expresses the number of correct predictions by the model out of the total number of predictions. We have used Linux Top tools that facilitate monitoring the CPU and displaying the processes currently being executed. It also can be used to display information about the CPU Usage and the system load. The table also shows the time required to fetch readings from the hardware performance counters using the Linux Perf for each virtual machine. The time here is user-defined, so the time appears the same for each host machine.

We designed a mechanism that depends on analyzing the behavior of VMs at the beginning of the approach and then integrating it with static analysis to ensure good system performance,

check the results, and then decide whether to suspend the virtual machine or not. Based on the obtained results, we were able to identify suspicious VM behaviors, And scanned the virtual machine against executable attack files with high accuracy. Also, this approach is applicable in the shared virtualized environments to monitor the activities of the VMs and identify the threat level of the suspicious VM from within the host machine. Table 4.4 is a table that shows a comparison between our method and some of the previous methods through the tools used, the targeted attacks, the percentage of detection accuracy, and compatibility with the nature of cloud computing.

Table 4.3: Experiment results

Dynamic Analysis				
Os and CPU Type	%CPU Usage	System Load	Accuracy	Time
Ubuntu i5-5200U CPU	0.6 - 3	0.33 - 0.35	%96.00	15 Sec
Debian i5-4200U CPU	1.3 - 2	0.11 - 0.22	%99.00	15 Sec
CentOS i5-5300U CPU	0.6 - 2	0.25 - 0.44	%95.58	15 Sec
Static Analysis				
Os and CPU Type	%CPU Usage	System Load	Accuracy	Time
Ubuntu i5-5200U CPU	18 - 25	0.85 - 1.25	%97.91	8 - 12 Min
Debian i5-4200U CPU	10 - 24	1.09 - 1.46	%98.31	8 - 12 Min
CentOS i5-5300U CPU	12 -24	1.02 - 1.38	%97.27	8 - 12 Min

Although various approaches for malware detection have been introduced, these approaches have critical drawbacks and limitations. First, they perform static analyses frequently and does not require the start-up condition, thus increasing the load on the system. Moreover, they are also not effective in detecting and protecting the shared virtualised environments from side channel attacks. We address the limitations of the current detection methods by proposing a new approach that detects and protects the shared virtualised environments against cache side-channel attacks. Our approach monitors and detects any abnormal behavior of virtual machines periodically and then performs static analysis of the detected virtual machine's executable files and then classifying the threat level of virtual machine executable files using neural network classification algorithms, thus eliminating the malicious VM, and protecting the shared virtual environment with acceptable performance.

Table 4.4: Comparison to the Previous Works

Method	Type	Tools	Attacks	Accuracy	Cloud?
[26]	Scan Apps in App Store	IDA PRO Static Binary Analyzer	Microarchitectural Attacks	97%	No
[88]	Monitoring CPU Counters	Intel PCM(Performance Counter Monitor)/Neural Networks Classification Model	Cache Attacks	97%	Yes
[24]	Monitoring CPU Counters	Linux perf/Machine Learning models	Cache Attacks	93%	Yes
[27]	Monitoring CPU Counters	Intel CMT(Cache Monitoring Technology)/Machine Learning Models	Prime+Probe Attack	99%	Yes
Our Method	Monitoring CPU Counters/Scanning suspicious VMs' Disk/RAM	Linux Perf/Logistic Regression Model/Neural Networks Classification Model/Radare2-R2pipe tools/Linux Objdump	Cache Attacks	97-98%	Yes

4.6 Limitations

Despite the promising results of this method in reducing the overhead of the system by relying on the hybrid analysis between dynamic analysis and static analysis to detect suspicious behaviour and then detect attack programs inside malicious virtual machines, some limitations must be considered, such as obfuscation of the code or hiding attack programs to bypass static analysis. Since the proposed method depends on two types of analysis, as mentioned, the use of code obfuscation techniques will partially affect the proposed detection mechanism, as only static analysis is affected, but dynamic analysis of virtual machines' activities will not be affected by these techniques. Also, the proposed method requires further experiments and evaluations on host machines that can run many virtual machines because the number of virtual machines may significantly impact analysis and monitoring activities on shared

virtualised systems. The proposed detection mechanism also needs to improve the analysis processes to include microarchitecture attacks to protect the shared virtualised systems from these attacks.

4.7 Summary

We proposed a side-channel attacks detection and protection method which combined dynamic and static analysis. The dynamic analysis used Linux Perf to obtain readings from 13 hardware counters related to the shared cache at runtime. Based on these readings, the VM behaviour was then classified into suspicious or benign using the logistic regression classification. As a second step, the static analysis extracted the executable files from the disk image or the RAM image of the suspicious VM. It then checked whether they contained operating codes for side-channel attacks. Based on this, the threat level of these files was determined using the Softmax classification algorithm; we had four threat levels in total. After that, the VM that posed a threat to the shared environment was excluded. Our proposed mechanism combines the advantages of dynamic analysis and static analysis to reduce the load on a system.

As a hypervisor, we employed KVM (Kernel-based Virtual Machine), and as guest operating systems, we utilized Linux Ubuntu 18.04.5 LTS (64bits). We then conducted experiments on several host machines, namely Ubuntu 18.04.5 LTS, Debian 10, and CentOS 8, with various processor models. The accuracy of detecting suspicious behaviour and classifying the threat level was recorded as 96%–99% percent with between 0.6%–25% CPU overheads for dynamic and static analysis.

The method designed in this chapter answers research questions 3 and 4 from Section 1.3 in the thesis, confirming the effectiveness of integrating dynamic and static analysis to reduce the load on the system and support the results of attack detection and protection, and the possibility of designing a static analysis that matches the nature of the shared virtualised environments. Furthermore, the experiments conducted to validate this method verify hypotheses H3, H4 and H5 from Section 1.4.

**MITIGATION THROUGH PERIODICAL LONG-RANGE
INTERVALS SCAN**

This chapter proposes a detection and protection method against microarchitectural attacks . The method utilises the microarchitectural attacks' static analysis. The method periodically extracts executable files for the long-term scan, such as once a week, analyzes them to detect whether they have microarchitectural attack attributes, and stores the results to be classified using a logistic regression model. We have created a shared virtualised system to conduct experiments on different hosts and processors. The results of our experiments show that our method can detect attacks with 97.47% accuracy and between 24–26% CPU usage.

5.1 Introduction

This chapter focuses on threats to shared virtualized systems including microarchitectural attacks. The microarchitectural attacks exploit the shared systems to reveal or alter secret data resulting in decrypted encryption keys hence compromising systems. A number of approaches have been proposed to mitigate the threats of microarchitectural attacks such as [26, 89, 95, 96, 107, 121, 122]. However, some shortcomings exist in these approaches which makes them unattractive for adoption as sufficient solutions for shared virtualized systems. They do not provide comprehensive protection for a large number of malware and microarchitectural attacks that threaten the shared virtualized system. Also, some of these approaches need to improve the detection and protection results. Also, some solutions are not compatible with shared virtualization systems to provide effective protection against microarchitectural Attacks.

In this chapter, we introduce a new method based on static analysis of the implicit characteristics of the attack's executable files to address the shortcomings of the existing techniques mentioned above. Also, the method can detect a wide range of executable files of microarchitectural attacks within the shared virtualised system. Furthermore, the method contains two types of analysis regarding the duration of the analysis process to detect attacks: fast scan and full scan to scan executable files within the shared virtualised systems.

Hence, this chapter makes the following primary contributions:

1. We introduce an approach for detecting and protecting shared virtualized environments against microarchitectural attacks by analyzing implicit attributes of executable files.
2. We describe the method and results of the mechanism design, implementation, and experimentation.
3. We evaluate the approach in various attack scenarios in terms of detection efficiency and performance attributes.

The remainder of the chapter is arranged as follows. Section 5.2 illustrates the proposed detection and protection method. Section 5.3 describes the experimental investigations based on the proposed method. In Section 5.4, the evaluation of the implemented method is discussed. Finally, Section 5.5 provides the conclusion.

5.2 Method

The proposed method in this chapter is based on static analysis of the implicit characteristics of the attack's executable files. The proposed method consists of two types of executable file scanning, namely, fast scan and full scan combined with logistic regression model for classification. We propose combining the procedure with a logistic regression classifier model for analysing these specific attacks' executable files and recognising whether the executable files contain any malware of microarchitectural attacks. The method include several steps to complete the scan successfully. As shown in Figure 5.1 for Scanning Microarchitectural Attacks the proposed method is accomplished using the following set of steps and stages.

1. Record all disk image addresses so that only one address is randomly chosen every time period to be scanned against microarchitectural attacks.
2. Utilize the Libguestfs tools [108] to access the target virtual machine's disk image on the KVM host, which can also be utilized in other hypervisors as well.
3. Next, the choice is made between a fast scan and a full scan. In the fast scan option, the executable files are extracted from the userspace. In other words, the files are extracted

from the userspace home folder of the Linux virtual machine. In contrast, with the full scan of the virtual machine, all executable files are extracted inside the virtual machine. However, full scan mode takes a longer time to complete the examination process.

4. Next, the extracted executable files are disassembled utilizing the Objdump command in Linux and the extracted executable files are examined against the implicit characteristics of the microarchitectural attack codes as shown in Table 2.2

The result of this analysis produces a dataset. The value of 1 in the dataset indicates the occurrence of the implicit characteristic. Otherwise (if not found), the value is set to 0 for each implicit characteristic. The attack characteristics are also checked to identify if they are continuously executed in a loop that takes the value 2. For analyzing the loop, we use a python program [123] that places each line of the disassembled binary into a linked list and separates the memory address, opcode, and operands for each line, then analyzing node addresses for the control flow graph.

5. Then all values of the implicit characteristics of the executable file are recorded and saved in a CSV file to be read and entered into the machine learning classifier model.
6. We create and train a binary logistic regression model to classify executable files whether they are malicious or benign.
7. Then, if any file exists that is classified as malicious, the file that poses a threat to the shared virtual environment will be alerted.

5.3 Experimental Setup

The experiments were conducted on multiple hosts using the QEMU-KVM hypervisor. The Ubuntu 18.04.5 LTS was employed which utilized an I5-4200M CPU, the Debian 10 utilized an I5-4200U CPU, and the CentOS 8 utilized an I5-5300U CPU. Then we created two VMs for each host that was running Ubuntu 18.04.5 LTS OS, one working as a VM for the victim and the other as a VM for the attacker. After that, we installed the Libguestfs tools [108] so that

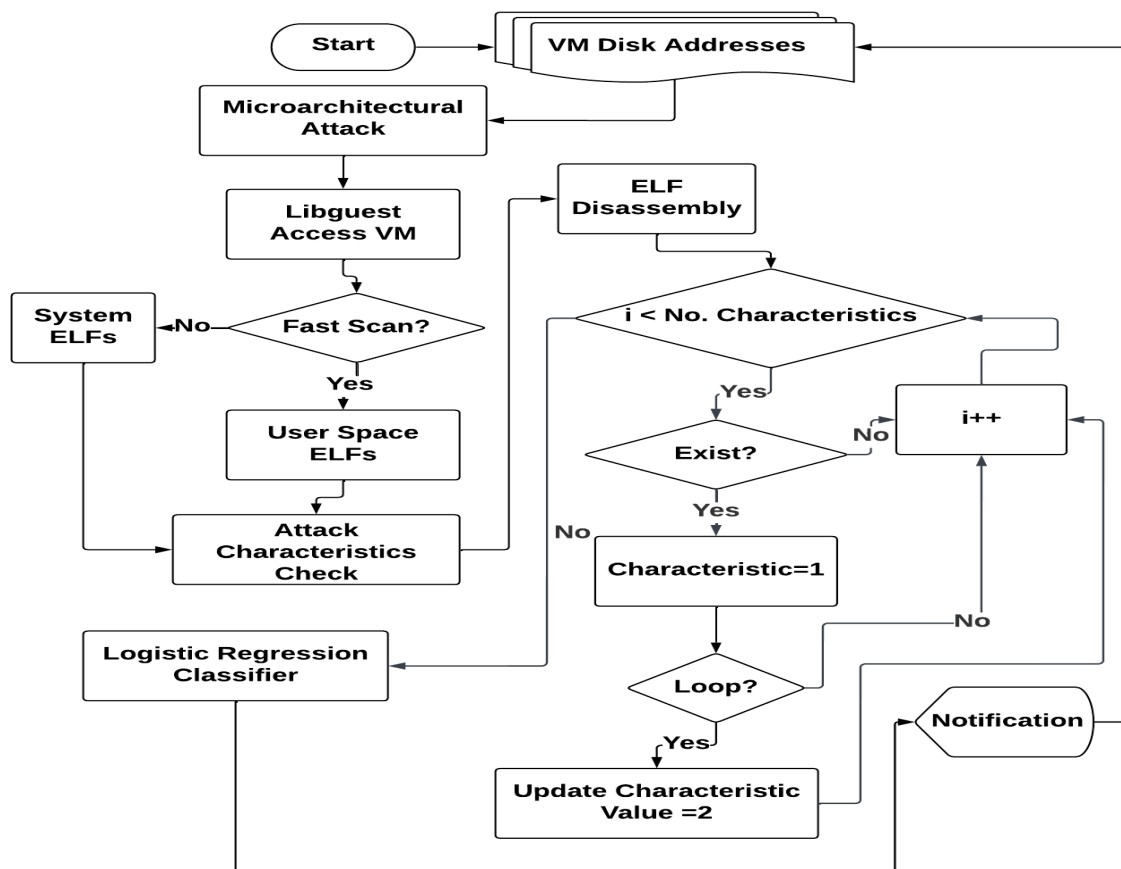


Figure 5.1: The Proposed Method

we can access, inspect, and modify the disk image of the VMs from the host. Linux Objdump tools was installed along with the Libguestfs tools to disassemble the executable files and check them afterward. We performed the static analysis for a set of microarchitectural attacks. Our analysis involved the Mastik tool designed by Yarom et al.[100], Xlate [113], Cache Template Attack source code presented by Gruss et al.[114], and the Flush + Flush attack tool [115], as well as other Github repositories inspired by "Cache Template Attacks"[57] and "FLUSH+RELOAD: a High Resolution, Low Noise, L3 Cache Side-Channel Attack" [4] such as [116], [117], [118], [119], and [120]. We have also analyzed and tested a set of the Specter attack programs [124–126], the Meltdown attack [127, 128], and Rowhammer [129, 130].

The proposed mechanism selected the disk image address of the target VM randomly every period from among many virtual VMs. After that, a mount point was created by the Libguestfs

tools [108] to extract executable files from the target VM (executing the static analysis from the host) while the VM was still running. We did not have to shut down the VM to access the disk image. After which, we created two options: extracting the executable files from the userspace or the system space. Extracting files from userspace is faster in the analysis process because the number of extracted files is much less. Next, we utilized the Linux Objdump tool to disassemble the executable files and display the opcodes or mnemonics for the machine instructions from the input file. Then we checked the presence of the attack's implicit attributes in the extracted executable files and also if they were in a loop. We recorded the appropriate values such that 0 indicates the absence of the implicit attribute and 1 indicates the presence of the implicit attribute but not inside the loop, and 2 indicates the presence of the attribute inside the loop. Then we saved the results into a CSV file to be processed by the logistic regression binary classifier model.

Creating a set of VMs, extracting executable files inside them, and analyzing extracted files to capture the implicit attributes included in these files, provided us with the dataset needed to train and test the classification model. We collected around 5,000 data samples in two cases, one for the case of no microarchitectural attacks files inside the VMs, and two the presence of a set of microarchitectural attacks files.

We then created the Logistic Regression classification model to analyze and classify the data based on whether the extracted data indicated the presence of an attack or not. We trained the model for 70% out of 5000 malicious and benign samples collected and tested on 30% of the samples. The model showed accuracy about 97.47% for the test case. We plotted the ROC curve that shows the accuracy, and also we recorded the false positives and negatives, as shown in Figures 5.2 and 5.3.

5.4 Experimental Results and Evaluation

In this part, we explain a set of experiments that were conducted to test and validate the efficacy of our method in terms of performance overhead, execution time, and accuracy of

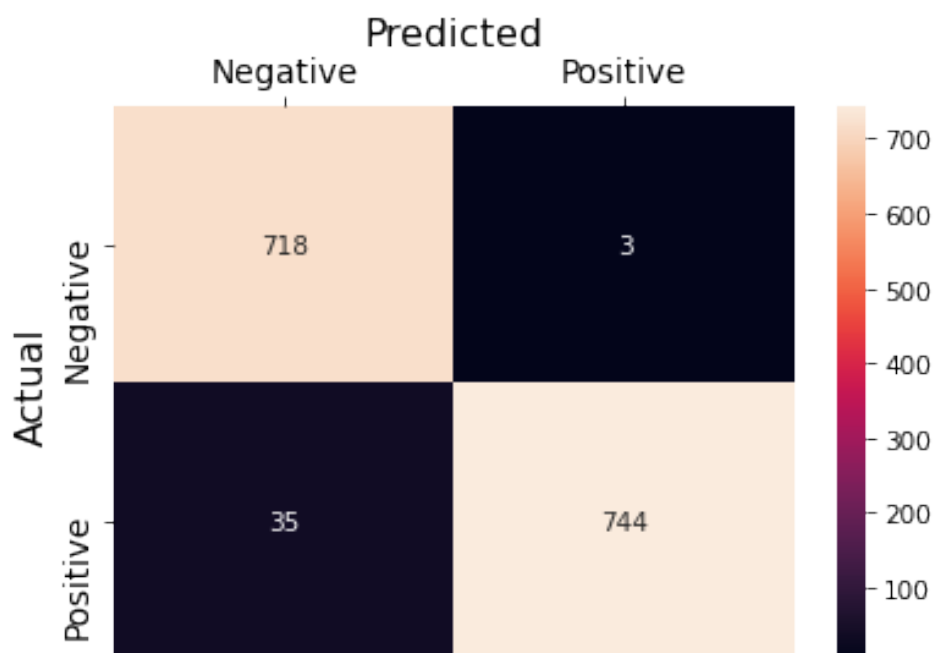


Figure 5.2: False Positive and Negative

detecting microarchitectural side-channel attacks using the logistic regression classifier. For evaluating our method efficiency in detecting attacks' executable files, we designed the shared virtualised environments described in the previous section.

We utilised QEMU-KVM hypervisor and ran test with various CPU models on three different hosts, including Ubuntu 18.04.5 LTS, Debian 10, and CentOS 8. After that, we recorded the results of the detection accuracy, performance overhead, and scan duration. In this section, these results are discussed.

We accomplished the evaluation in two parts, as shown in Tables 5.1 and 5.2. The first is a microarchitectural attacks scan, and the other is a viruses scan.

5.4.1 Microarchitectural Attacks Scan

We evaluate this part using a fast scan as well as a full scan for each host used as follows:

1. Detection Accuracy: To evaluate the accuracy of microarchitectural attacks detection

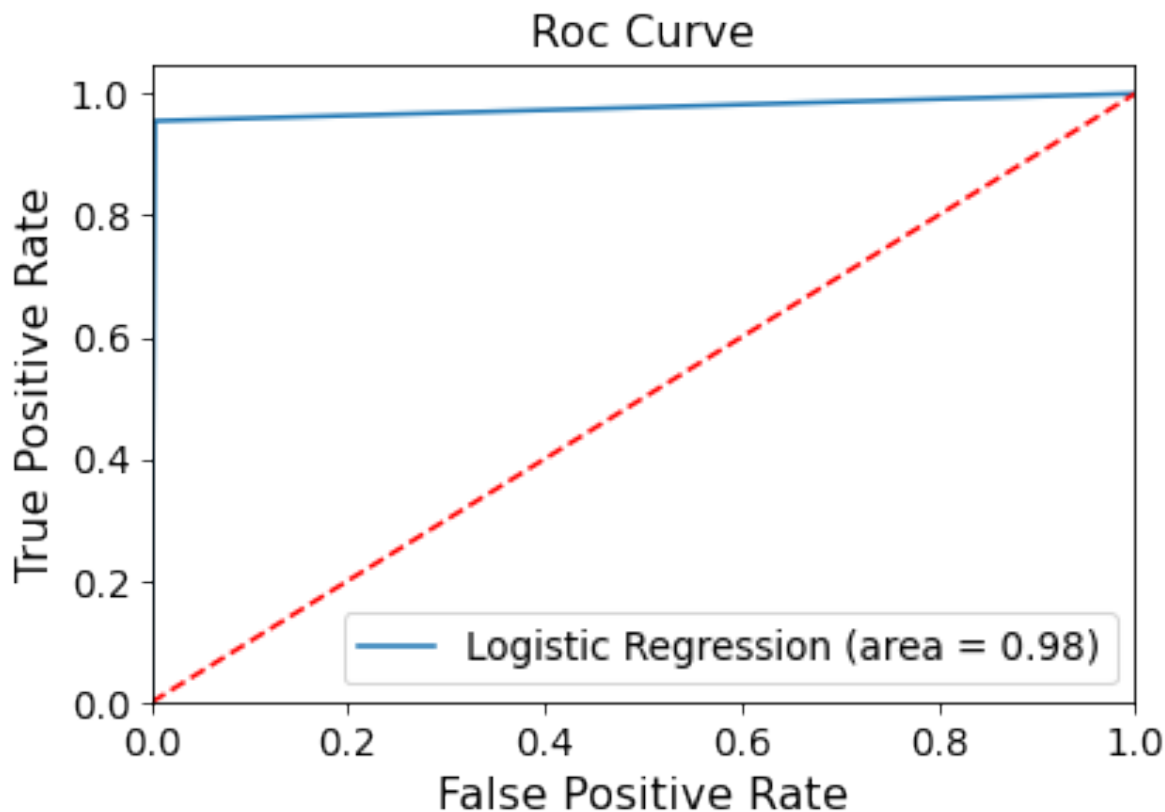


Figure 5.3: The Roc Curve

Table 5.1: Attacks Scan Experiment Results

Attacks Fast Scan						
Os and CPU Type	#Files	Duration	CPU Time	%CPU Usage	Overhead	%Accuracy
Ubuntu i5-5200U CPU	97	4 Mins	0.0103 Sec	25	1.23	97.47
Debian i5-4200U CPU	230	19 Mins	0.2001 Sec	24.5	1.4	
CentOS i5-5300U CPU	117	7 Mins	0.0152 Sec	26	1.06	
Attacks Full Scan						
Os and CPU Type	#Files	Duration	CPU Time	%CPU Usage	Overhead	%Accuracy
Ubuntu i5-5200U CPU	1922	38 Mins	0.1959 Sec	28.5	1.31	97.47
Debian i5-4200U CPU	1824	43 Mins	0.1472 Sec	34	2.38	
CentOS i5-5300U CPU	1956	37 Mins	0.2444 Sec	42	1.96	

based on the logistic regression model, we measure our detection model's true-positive and false-positive rates by analyzing the different types of microarchitectural attacks, as shown in Figure 5.2. We achieved an F1-score of 0.98, as shown in Figure 5.3. The accuracy of detection was recorded in the process of testing the logistic regression classification model.

Table 5.2: Viruses Scan Experiment Results

Viruses Fast Scan						
Os and CPU Type	#Files	Duration	CPU Time	%CPU Usage	Overhead	%Accuracy
Ubuntu i5-5200U CPU	97	13 Mins	0.0112 Sec	26	1.14	79
Debian i5-4200U CPU	230	50 Mins	0.0209 Sec	24.8	1.04	
CentOS i5-5300U CPU	117	36 Mins	0.0146 Sec	25.5	1.2	
Viruses Full Scan						
Os and CPU Type	#Files	Duration	CPU Time	%CPU Usage	Overhead	%Accuracy
Ubuntu i5-5200U CPU	1922	337 Mins	0.2099 Sec	32	1.44	79
Debian i5-4200U CPU	1824	449 Mins	0.1475 Sec	36.5	2.7	
CentOS i5-5300U CPU	1956	696 Mins	0.2231 Sec	47	2.17	

2. Performance Overhead: The overhead was recorded for the entire system while the detection mechanism was running to analyze just one VM. We found that the overhead was different for each host, but it was slightly close to each other. The Linux Top application was relied upon to record the overhead rate information every 5 minutes and the CPU usage rate, as shown in Table 5.1. The CPU usage was intensive. However, it depends on the computing power and model of the processor used.
3. Scan Duration: The duration of the detection mechanism was recorded as shown in Table 5.1. The execution time was also recorded using the CPU time. We noticed that the execution time depends on the number of files extracted for analysis, and the type of processor also has an impact on the execution time. There were two types of scanning, and there was a significant difference between a fast scan and a full scan for executable files, which indicates the importance of a fast scan, as shown in Figure 5.4 and 5.5 which show how many files have been scanned and how many minutes it has taken to scan, while we need to develop a full scan to be faster. Although we do not need to turn off the virtual machine for examination, detection speed is essential for the detection mechanism.

5.4.2 The Solution Works in Parallel with Antivirus applications

The other part of the experiment involved checking files using ClamAV antivirus. This application is capable of detecting numerous varieties of malware; moreover, its databases are updated continuously [131]. The ClamAV user can manage it and access all the functions

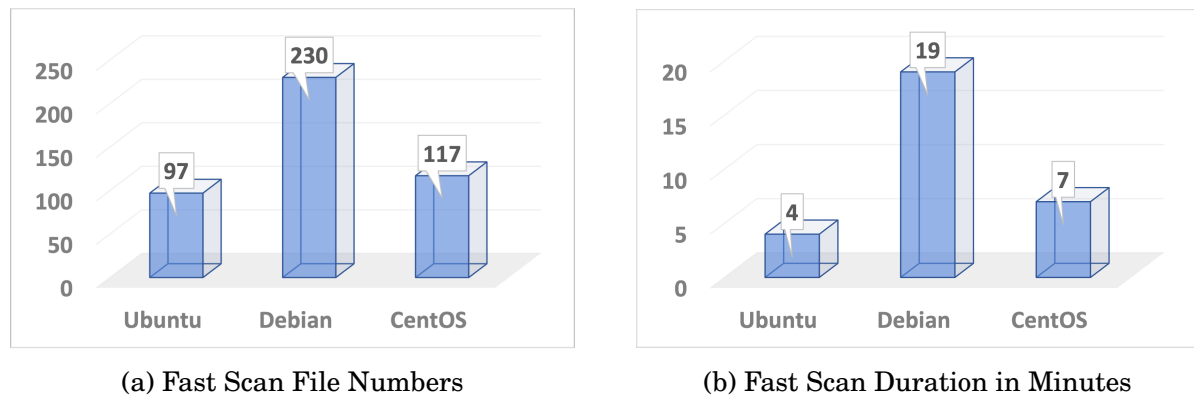


Figure 5.4: Fast Scan for Attacks

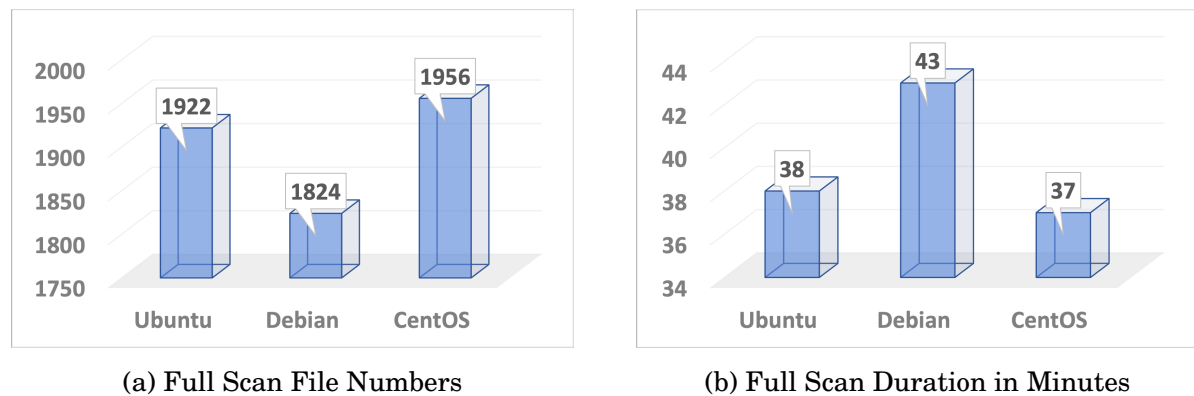


Figure 5.5: Full Scan for Attacks

through the command-line interface. There are numerous features for the ClamAV antivirus, such as scheduling scans and sending the scan report by e-mail. It also helps select files for the scan. We extracted the executable files from the VM and checked them directly. We can also reduce the execution time by choosing a fast scan option to extract files from the userspace. After that, if malicious files are detected, a notification of the scan results is sent. We have run the detection method in parallel with ClamAV as follows:

1. We keep track of all disk image addresses so that one address is randomly selected every time period to be inspected with Clam AV software.
2. We then make use of the Libguestfs utilities to gain access to the disk image of the target virtual machine running on the KVM host.

3. Then we choose between a fast scan or a full scan of the executable files of the target virtual machine.
4. After that, the extracted files from the disk image can be scanned using Clam AV without disassembling the executable files.
5. Then, if any file is identified as malicious, the file that poses as a threat to the shared virtual environment will be alerted.

Furthermore, we have evaluated this part using a fast scan as well as a full scan for each host using ClamAV as follows:

1. **Detection Accuracy:** ClamAV has been used because it is open source and supports the use of the command-line for managing its functions. This facilitates its adoption in the design of the detection mechanism. The detection rate was 79% that is a reasonable rate compared to other antivirus programs' detection rates, such as McAfee and Avast, according to the study presented in [132]. We can also use a different antivirus application or a group of antivirus applications to raise the detection rate. However, it may be a significant overhead on the system and may cause a longer execution time, affecting the entire system's performance.
2. **Performance Overhead:** Overhead was monitored every five minutes for the entire system when the mechanism was running using ClamAV to check only one VM per host, as shown in the table 5.2. CPU usage was also monitored, and there was no significant difference between all hosts. Values may increase when adding more VMs running on the same host.
3. **Scan Duration:** The execution time of the fast scan and the full scan using the ClamAV antivirus were recorded, as shown in Figure 5.6, as there was a significant difference between the duration of the two scans, and there was a massive difference in the number of files as well. ClamAV was slow in the scanning process, and it spent roughly eighteen seconds to scan only one file. However, the scanning mechanism does not

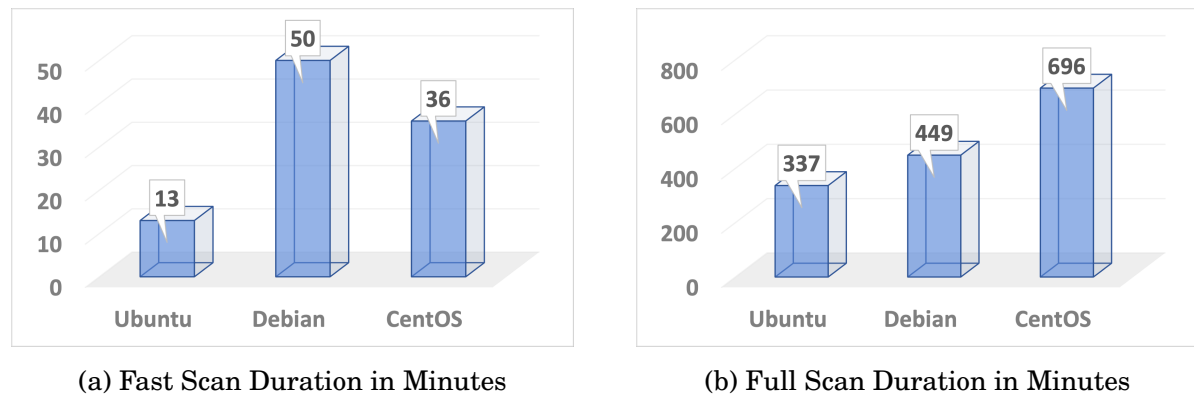


Figure 5.6: Scan Duration using ClamAV

require turning off the virtual machine, as it normally works, which may be acceptable to some extent.

5.5 Summary

This chapter introduced a detection and protection method against microarchitectural attacks using static analysis based on a machine learning algorithm. The process periodically extracts VMs' executable files and checks them against microarchitectural attacks' implicit attributes (opcodes). Depending on the scan duration, we identify two types of scans: (i) fast scan and (ii) full scan. In the case of the fast scan, the files are extracted from a VM's userspace. And in the case of the full scan, the files are extracted from the kernelspace of the entire VM system in a shared virtualised system.

A logistic regression model is used to classify the executable files in case of scanning against microarchitectural attacks. We employed QEMU-KVM (Kernel-based Virtual Machine) hypervisor, and as guest operating systems, we utilized Linux Ubuntu 18.04.5 LTS (64bits). We then experimented with various processor models on several host machines, namely Ubuntu 18.04.5 LTS, Debian 10, and CentOS 8. The accuracy of detecting executable files was recorded as 97.47% with between 24%–26% CPU usage for the static analysis.

The method proposed in this chapter answers research questions 5 from Section 1.3 in the

thesis, confirming the possibility and effectiveness of generalising static analysis to extend the scope of protection against other microarchitectural attacks. Also, it can work in parallel with antivirus applications such as ClamAV to perform long-term periodic checking against attacks and viruses as well. No antivirus can scan VM applications against microarchitecture attacks, so it is important to design our method comprehensively to accomplish this. Furthermore, the experiments conducted to validate this method confirm hypothesis H6 from Section 1.4.

CHAPTER



HOLISTIC PROTECTION SOLUTION

This chapter designs a comprehensive solution of the three methods we proposed in previous chapters to provide comprehensive protection for shared virtualised systems against microarchitectural attacks. This chapter presents experiments for comprising the three methods and evaluating them in the possible operational scenarios. It evaluates the overhead on the system and the CPU usage using different host systems for the evaluation processes.

6.1 Introduction

In the previous chapters we discussed several methods we have proposed to mitigate the threat of side-channel attacks and microarchitectural attacks. We evaluated each method individually to illustrate its advantages. In this chapter we combine these methods to design a new mechanism that combines the advantages of the proposed methods. The mechanism can monitor the activities that sensitive memory sites are exposed to, as it can monitor and analyse activities of VMs through CPU performance counters, and perform periodic inspections of VMs and suspend the VM that represents a threat to ensure the integrity of a shared virtualised system. The novelty of this project is the design of a diverse and comprehensive detection and protection system that protects against cache side-channel attacks and microarchitectural attacks. Our mechanism works at the level of VMs and host machines. Thus, the VM can provide self-protection by using the memory deduplication feature to monitor malicious activities that target shared cryptographic libraries and shared programs, obfuscating the attack results. Also, the host provides protection for the shared virtualised system by relying on hybrid analysis processes, namely dynamic analysis, to monitor suspicious activities. It also uses static analysis to extract executable files that pose a threat. Our method also relies on machine learning models to support data analyses and improve result accuracy. The proposed mechanism periodically scans VMs to ensure their integrity from the presence of executable files that contain implicit attributes of microarchitectural attacks. Our approach contains diverse lines of defence that are difficult for attackers to penetrate and bypass. It can also work in a shared virtualised system with acceptable performance and high accuracy. We have explained in detail in previous chapters the design of the detection

and protection systems and how to implement and evaluate it. This chapter aims to design a protection system based on a comprehensive solution that combines all the solutions in the thesis, implement the mechanism and evaluate its performance characteristics.

The remainder of the chapter is arranged as follows. Section 6.2 illustrates the comprehensive detection and protection methods. Section 6.3 describes the experimental setup. In Section 6.4, the evaluation of the results is discussed. Finally, Section 6.5 provides the conclusion.

6.2 Methodology

The methodology used consists of a combination of three methods, as explained in Chapter 3, Chapter 4, and Chapter 5, that operate in shared virtualised systems. We will discuss these separately below.

The first method monitors and protects sensitive shared program functions (cryptographic libraries and shared executable files) from within a VM. It also uses the memory deduplication feature to obtain attack readings and then analyses them using the logistic regression model. It can detect suspicious activities that sensitive shared programs are exposed to during the execution of sensitive operations in shared virtualised systems. Additionally, it can obfuscate the results of attacks obtained by the attacker. The method supports VMs to detect attacks by knowing the attacks' readings, thus providing self-protection for the VM.

Suppose two VMs run in a shared virtualised system that supports the memory deduplication feature. One of the VMs is malicious and the other is a target or victim VM. The two VMs share the last level of cache on the same host and also share the same cryptographic libraries and executable files as a result of using memory deduplication, which deletes all replicas of executable files and retains only one shared copy between them to save memory capacity. As shown in Figure 6.1, the VMs are able to access the shared memory and perform a flush-based cache attack in the system. The attacker performs the flush-based attack as follows (steps number 1 and 6 in Figure 6.1).

- The attacker defines the desired memory addresses related to the shared executable file's target functions and flushes them out of the cache, using the *clflush* instruction (the attacker may need to repeat flushing of the same addresses multiple times to ensure the attack's success). The intention is for the flushed functions' addresses to be retrieved from the main memory when the victim requests and executes these functions.
- After that, the attacker waits to give the victim some time to perform encryption operations or execute sensitive data related functions. Next, the attacker reloads the flushed functions' addresses and measures the access time (using the *rdtsc* instruction) to determine whether or not the victim has requested and executed those functions.

The proposed protection mechanism includes the steps numbered 2, 3, 4 and 5 in Figure 6.1).

- It gets the shared functions' addresses of executable files and cryptographic libraries to be monitored and shielded from the flush-based cache attacks.
- It recovers the monitored functions into the cache memory while measuring each function's recovery time over each specified period of time. As a result, the functions will be reloaded and the detection mechanism will discover the flush instructions. The measurement is conducted by utilizing the *rdtsc* instructions that provide a high-resolution time stamp counter. It sets an iteration sample for the monitored functions so that the recovery time is measured frequently. It is then measured against the threshold of the system to detect whether flush instructions have been conducted on the functions. Because the detection mechanism accesses the addresses specified continuously to be monitored, the attack results are obfuscated, so the attacker will record that cache hits for all the addresses monitored even if the victim does not use them.
- It records the number of flushes for each monitoring function and then analyses it using logistical regression.
- It then warns the user in case of attack.

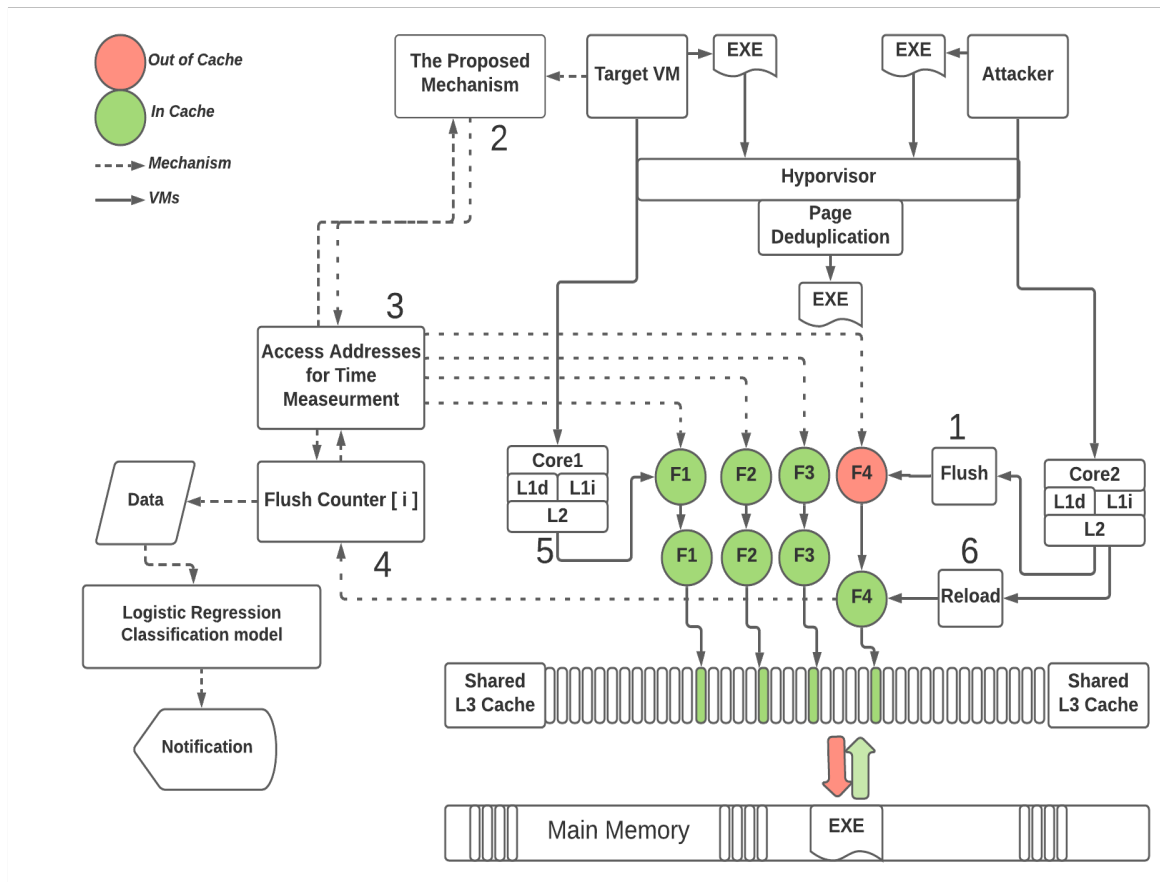


Figure 6.1: Flush-based Attacks Detection Method

The second method is a mechanism for detecting and protecting against cache side-channel attacks from inside the host. The technique is a hybrid of dynamic and static analysis, as shown in Figure 6.2. The dynamic analysis monitors VMs' activities in a virtualised system by obtaining readings from hardware performance counters relevant to the shared cache at runtime. Then the activities of the VMs are classified between benign or suspicious after analysing the readings using a logistic regression model. When any suspicious activity is detected, the static analysis runs. The static analysis accesses the suspicious VM and extracts executable files from a disk and RAM images. It then examines whether these files contain opcodes of cache side-channel attacks. Based on the results, the threat level of these files is determined using a neural network classification model.

The third method is based on a combination of static analysis with the ClamAV application.

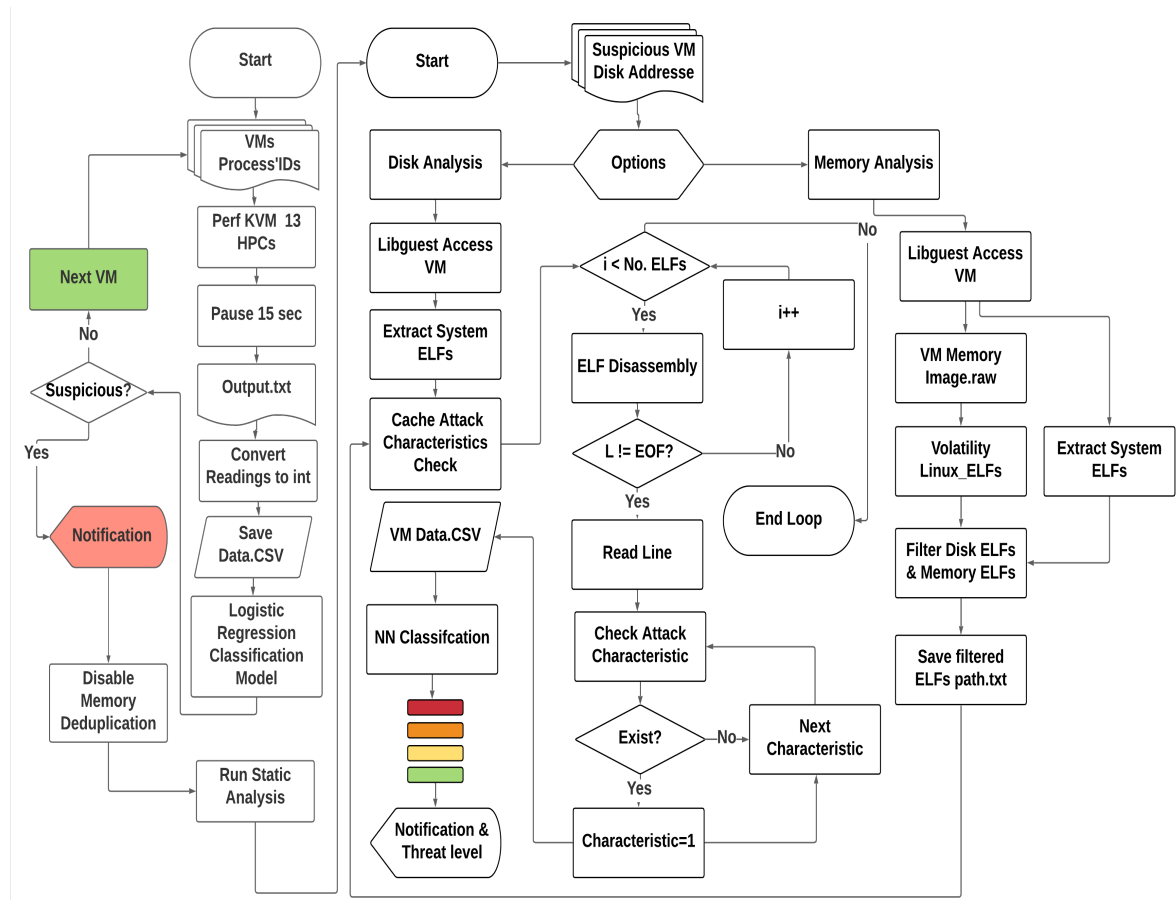


Figure 6.2: The Dynamic and Static Analysis Protection Method

The method runs periodically and the VMs are randomly selected for testing. There are two types of scans: a microarchitectural attacks scan and an antivirus scan, as shown in Figure 6.3. The mechanism accesses the VM, extracts executable files, checks if they contain the implicit characteristics of a microarchitectural attack, and then analyses the results using a logistic regression model to detect whether there are any malicious files and whether they contain viruses. The scan is divided into two parts: a fast scan and a full scan, in terms of scan duration. This method provides periodic scanning of a shared virtualised system to eliminate attack files and viruses, and to identify the malicious VM. The methods operate in tandem to provide adequate protection for a shared system.

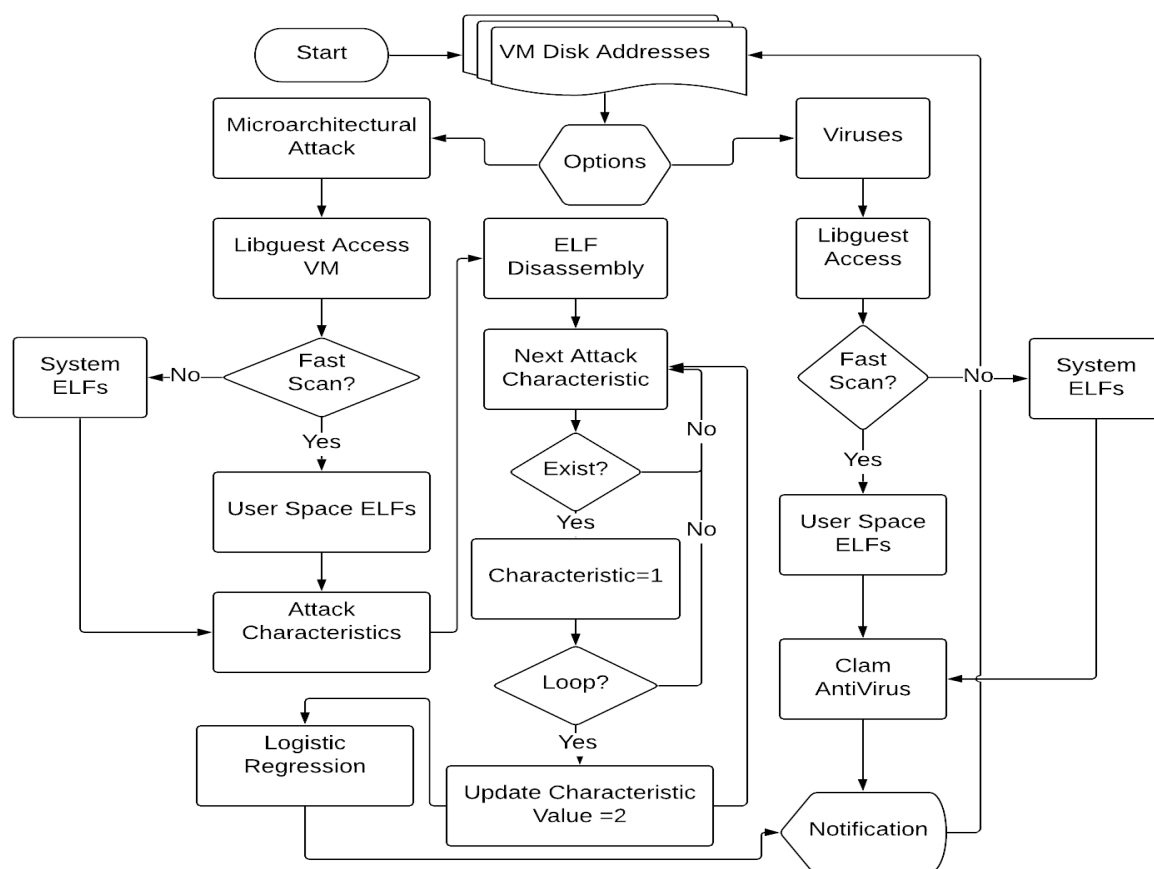


Figure 6.3: The Periodically Cleansing Method

6.3 Experimental Setup

The same computing environment was used as in previous chapters. We created shared virtualised systems and executed experiments with the QEMU-KVM hypervisor that runs KSM as a memory-saving deduplication feature. As hosts we used Ubuntu 18.04.5 LTS which uses an Intel Core i5-4200M CPU, and Debian 10 which uses an Intel Core i5-4200U CPU. We then created two VMs on the same host. The VMs were running Ubuntu 18.04.5 LTS OS, one VM as an attacker and the other as a victim. We installed several essential tools inside the VMs. For instance, the GDB (GNU Project debugger) tool to facilitate finding the shared executable file's functions' addresses, thus facilitating monitoring the functions. We also installed AVML (acquire volatile memory for Linux) to capture the RAM status regularly. Moreover, we installed the Linux Perf, the Libguestfs Tool, the Linux Objdump Disassembler,

radare2, Volatility Tools, and ClamAV inside the host. We conducted the attacks using the Mastik tool designed by Yarom et al. [100]. We conducted experiments using this environment to evaluate the system's overall performance and record the system load and CPU overhead. The mechanism combines four mechanisms, as shown in Algorithm 5, which may significantly impact a system. Thus, we decided to conduct experiments and record the effect.

6.4 Experimental Results and Evaluation

The experiments were conducted to assess the load on the system and the overhead of the CPU. Therefore, the experiments were performed with different scenarios, as follows.

- Execution of the experiment on dynamic analysis mechanisms while we were conducting side-channel attacks: the mechanisms for monitoring the activities of VMs were only included with this experiment. The mechanism of Chapter 3 was combined with the mechanism of dynamic analysis in Chapter 4, and the overhead of the processor was measured.
- Execution of the experiment on the mechanisms of static analysis: the static analysis of side-channel attacks in Chapter 4 was combined with the static analysis of microarchitectural attacks in Chapter 5.
- Measuring the load on the system and the CPU's overhead while operating all mechanisms. The dynamic and static analyses were implemented together.

The load on the system and the overhead for the CPU were measured using the Linux *Top* tool, where the system load rate was recorded every 5 minutes, as shown in Table 6.1. Also, the detection accuracy of the proposed mechanisms were averaged based on the detection accuracy experiments in Chapters 3,4 and 5.

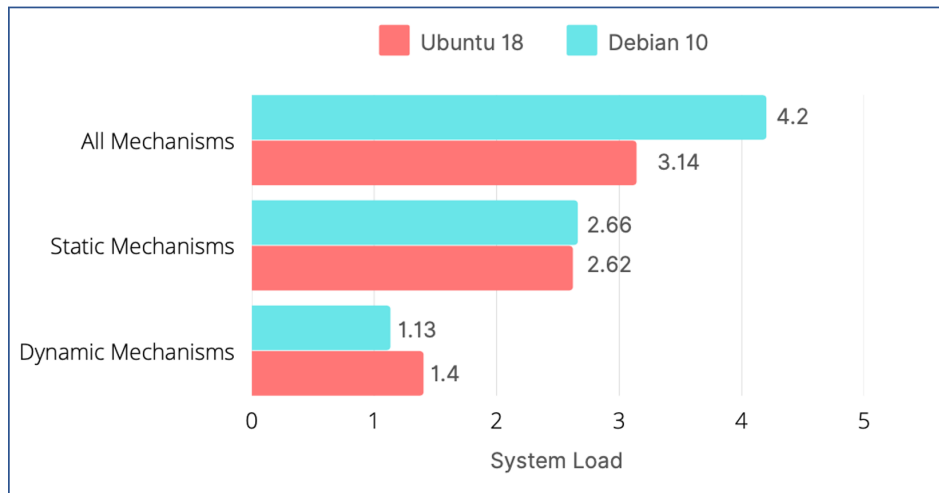
Based on the results shown in Figure 6.4, it was found that there is an increase in CPU usage and system load. However, the reason for this may be either the limited computing

Algorithm 5 holistic Protection Solutions**Input:** *Functions Addresses, VM process IDs, VM Disk Addresses***Output:** *Attack Detection Notification, Threat Level*

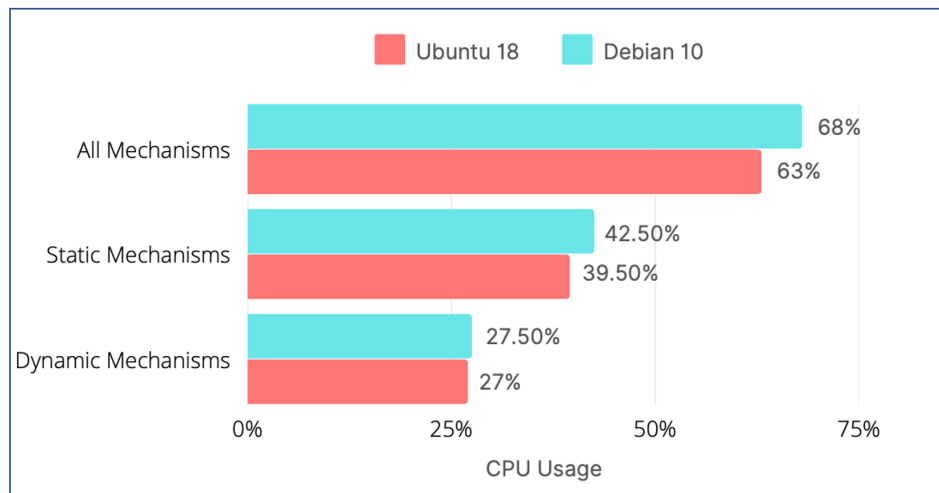
```

1: Model_P1 = Logistic Regression()
2: while true do
3:   for  $i = F\_Address_1$  to  $F\_Address_n$  do
4:     Run Monitoring Addresses access time measurements
5:     for Iteration do
6:       Record EvictCounter[i++] → Data_P1.csv
7:       Result_P1 = Model_P1 ← Data_P1.csv
8:       if Result_P1 = 1 then
9:         Detect Cache Attacks
10:      end if
11:    end for
12:  end for
13:  for VM process IDs do
14:    Model_P2.1 = Logistic Regression()
15:    Run Dynamic Analysis to monitor VM
16:    Record the readings of HPCs → Data_P2.1.csv
17:    Result_P2.1 = Model_P2.1 ← Data_P2.1.csv
18:    if Result_P2.1 = 1 then
19:      Disable Memory Deduplication
20:      Suspend the VM to Scan RAM
21:      Model_P2.2 = NNClassification()
22:      Run Static Analysis to access VM RAM Image
23:      Find and disassemble Executable files → files_P2.2.txt
24:      Search Cache Attacks opcodes in file_P2.2.txt → Data_P2.2.csv
25:      Threat Level = model_P2.2 ← Data_P2.2.csv
26:      Threat Level Notification
27:    end if
28:  end for
29:  Pause(Long Period)
30:  for VM Disk Addresses do
31:    Model_P3 = LogisticRegression()
32:    Find Executable files → files_P3.1.txt
33:    Result_P3.1 = ClamAV ← files_P3.1.txt
34:    if Result_P3.1 = 1 then
35:      ClamAV Notification
36:    end if
37:    Disassemble Executable files in files_P3.1.txt → files_P3.2.txt
38:    Search Microarchitectural Attacks opcodes in file_P3.2.txt → Data_P3.csv
39:    Result_P3.2 = Model_P3 ← Data_P3.csv
40:    if Result_P3.2 = 1 then
41:      Microarchitectural Attacks Notification
42:    end if
43:  end for
44: end while

```



(a) System Load



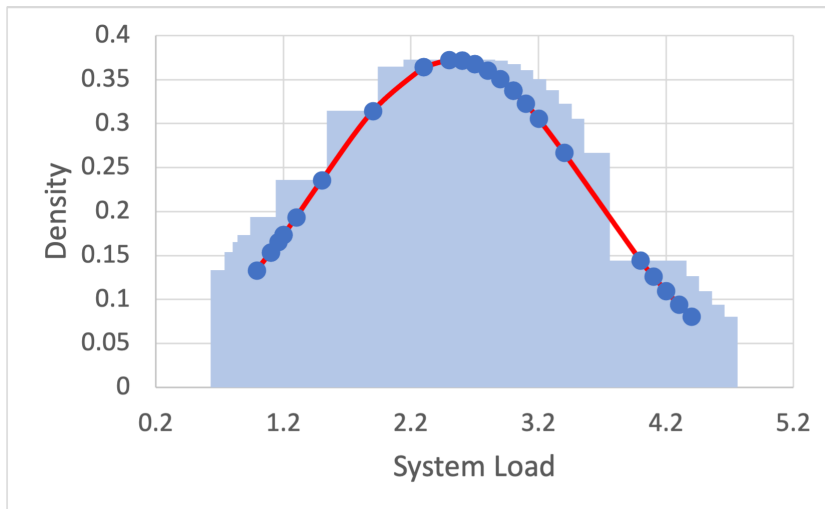
(b) CPU Usage

Figure 6.4: Systems Overhead

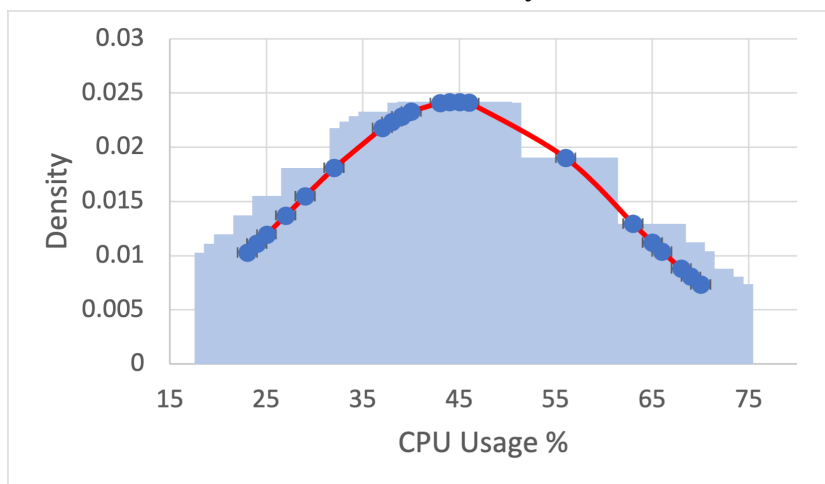
power of the systems used, as we utilised an Intel Core i5 processor for both hosts, or the complexity of the code. In both cases, the results will be better if a server with suitable computing capabilities is utilised. Moreover, the mechanism relies on dynamic analysis to determine whether there are any suspicious behaviours that require static analysis to ensure the presence of attack files inside a suspicious VM. This condition may reduce overhead and improve performance. Also, the static analysis that is used to detect files of microarchitectural attacks scans VMs for long-term periods, rationalising the reliance on static analysis, despite its importance in protecting a shared virtualised system.

Figure 6.5 represents the normal distribution of system load and processor usage. The figure also shows the probability density of load and processor usage. The normal distribution was calculated by calculating the mean and standard deviation, and then the normal distribution was calculated using Microsoft Excel. We can also calculate the normal distribution after calculating the mean μ and standard deviation σ using the following equation:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right)$$



(a) Normal Distribution of System Load



(b) Normal Distribution of CPU Usage

Figure 6.5: Normal Distribution of Systems Overhead

Table 6.1: System Overhead

Dynamic Mechanisms		
OS	System load	CPU Usage
Ubuntu	1.40	27%
Debian	1.13	27.5%

Static Mechanisms		
OS	System load	CPU Usage
Ubuntu	2.62	39.5%
Debian	2.66	42.5%

All Mechanisms		
OS	System load	CPU Usage
Ubuntu	3.14	63%
Debian	4.2	68%

Accuracy	
Ubuntu	97.25%
Debian	98.32%

6.5 Summary

Cloud computing relies on sharing resources between users of the same physical machine to reduce costs by optimizing and increasing utilization. However, sharing these resources may occur with malicious users, leading to confidentiality violations through co-residency attacks. These attacks may exploit the sharing of resources, such as cache memory, to reveal a legitimate user's recent activities. Multiple techniques and factors can be exploited to successfully perform side-channel attacks and other microarchitectural attacks. Therefore, despite all the benefits of sharing resources on the same physical machine, there is still a risk. If this security risk is not properly and adequately mitigated, it could be the main concern that obstructs cloud adoption. This chapter introduced the use of the three approaches together to provide the necessary protection for shared virtualised systems. These approaches provide self-protection for the VM on which they are used by monitoring activities within shared virtualised systems, determining the threat level of suspicious VMs, and providing periodic scanning of the virtualised system against microarchitectural attacks and viruses.

We have proposed the development of three methods to provide comprehensive and holistic protection for shared virtualised systems against microarchitectural attacks. The first method

detects cache attacks using memory deduplication and a logistic regression model. The second method detects and protects shared virtualised systems against cache side-channel attacks by integrating a dynamic and static analysis, and identifying the threat level of a particular VM by using machine learning algorithms. The final method periodically cleanses shared virtualised systems against microarchitectural attacks and viruses by analysing implicit attributes of executable files using a logistic regression algorithm comprised with ClamAV.

The method designed in this chapter answers research questions 6 from Section 1.3 in the thesis, confirming the possibility of use the proposed solutions together in parallel to develop a comprehensive protection system that operates on several levels. However, an increase in the CPU overhead was recorded according to the computing power used in the experiments. Furthermore, the experiments conducted to validate this method confirm hypothesis H7 from Section 1.4.

CHAPTER



CONCLUSION AND FUTURE WORK

Finally, in this chapter, we summarise the contributions and findings in order to determine how sufficiently the research questions have been addressed. In addition, this chapter discusses the project's limitations and offers some ideas for future work. The remainder of this chapter is organized as follows: Section 7.1 summarises the thesis research questions and objectives, Section 7.2 outlines the thesis contributions, and Section 7.3 discusses the limitations and opportunities for future work.

7.1 Conclusion

When the shared virtualised system or any of the VMs operating within the virtualised system are attacked using the side-channel attack techniques, previous studies relied on four possible solutions to address and defeat such attacks and mitigate their effects, which are: eliminating imbalance, isolating, avoiding co-location, and detecting malicious processes using the CPU counters [30, 61]. However, the previous studies did not explicitly study the causes of the problem in a manner that preserves the characteristics and features of the shared virtualised system and cloud computing. Furthermore, the previous studies have not focused precisely on what decisions and procedures are to be undertaken after detecting attacks in addition to the limitations related to them, which were mentioned and discussed in the literature reviewed in Chapter 2. Thus, no extensive investigations have been conducted on memory deduplication features in the cloud and how it can be used as a defence factor against these attacks, which may be more effective in defence against these attacks. Also, there were no widely conducted studies on static analysis in line with the nature of the shared virtualised system, and resorting to static analysis was not among the options for possible solutions to protect the shared virtualised system. Our literature review did not find studies and solutions based on hybrid analysis that combines dynamic analysis and static analysis to detect attacks, protect the shared virtualised system, and exclude virtual machines that represent a threat to the system.

This thesis has explored the possibility of designing and developing a comprehensive security control system to protect the shared virtualised system using different perspectives and analyses. Particularly, in this thesis, we did not lose sight of what will happen before and

during the attack, and what are the possible operations after confronting and detecting the attack. Hence we monitored the most likely attacking cache lines that carry sensitive data instead of monitoring the cache in general without any influence that could be done to the monitored cache line. Therefore, we were able to take precautions during the attack process, which may confuse the results of the attack if it is not detected in rare cases. We have also monitored suspicious activities on the shared virtualised environment and suspend the suspicious virtual machine until it is examined, the threat level is determined, and its integrity is verified. This step is very important to protect the virtualised environment in general. We have also not neglected to periodically check VMs at frequent intervals to detect microarchitectural attack programs and viruses as well.

From the results of this research, we are able to conclude that memory deduplication is not just a vulnerability of the shared virtualised system, but rather it is a very important factor in defending against side-channel attacks, in addition to obfuscating the results of the attack, which makes it difficult for the attacker to reveal sensitive information to the victim, and it also provides self-protection for VMs as illustrated in Chapter 3.

The thesis also concludes on the importance of relying on hybrid analysis combined with machine learning algorithms to defend against side-channel attacks to reduce system overheads, improve cache attack detection results, and decide to exclude malicious VMs discussed in Chapter 4. The thesis emphasises the importance of generalising the methods to other microarchitecture attacks and viruses to examine virtual machines and designing rapid and comprehensive examination methods for the virtual machine system, thus maintaining the integrity of the shared virtual system, the results have been discussed in Chapter 5. This thesis has also discussed the design of a system that combines the different methods, which makes it difficult for the attacker to carry out accurate architectural attacks, which effectively reduces the threats to the shared virtualised system (discussed in Chapter 6).

The aim of this thesis, as stated in Chapter 1 was to fulfill the following:

How to build a comprehensive security system based on the integration of different methods and machine learning algorithms to detect cache side channel attacks and remove any threats

to the shared virtual system with high accuracy and low overhead.

A variety of subjects have been discussed in the preceding chapters that associate with achieving the aim, answering the research questions, and accomplishing objectives specified in Chapter 1. The research contributions will be outlined below to show how the following research questions presented in chapter 1 have been addressed.

- *Question 1:* How can memory deduplication be used to design a security system to monitor the activities of VMs and obfuscate the outcome of the attack?
 - This question was answered in Chapter 3. Experiments proved the possibility of using memory deduplication as a protection factor to monitor sensitive cache locations, thus identifying abnormal activities that indicate cache side-channel attacks and the ability to confuse the attack results.
- *Question 2:* How can a VM detect suspicious activity and defend its apps from Flush-based attacks while hardware performance counters inside the VM are limited?
 - This question was also answered in Chapter 3, where the proposed mechanism can be used inside a VM to support it with protection and detection of suspicious activities and analysis of these activities using logistic regression. At the same time, there is a limitation in the use of hardware performance counters from inside the VM, as discussed in the chapter.
- *Question 3:* Is it possible to develop a protection system that relies on static and dynamic analysis of activities within the virtualised system to reduce system overheads and effectively support detection and protection results?
 - The proposed method in Chapter 4 answered this question. It demonstrated the effectiveness of combining dynamic analysis and static analysis in supporting results and reducing system overheads by decreasing the use of static analysis, except in cases where there are indications of a cache side-channel attack.

- *Question 4:* How can static analysis performance be improved to be compatible with shared virtualised systems, so that suspicious VMs can be scanned to protect systems against cache side-channel attacks?
 - This question is answered in Chapter 4, where a system was designed based on static analysis that utilised reverse engineering to scan VM's executable files against operating codes of the cache side-channel attacks. The conducted experiments also confirmed the effectiveness of the protection system and the accuracy of detecting malicious files.
- *Question 5:* Is it possible to extend the scope of static analysis to detect microarchitectural attacks more broadly and combine it with an antivirus application to detect malware, whether viruses or microarchitectural attack files, to design a more comprehensive system for protecting shared virtualised systems?
 - The proposed solution in Chapter 5 answered this question, confirming that it is possible to extend the scope of static analysis based protection to include other microarchitectural attacks, and it has been effectively integrated with ClamAV Antivirus to provide more comprehensive protection compatible with shared virtual systems.
- *Question 6:* How can the provided solutions in the previous chapters be combined to create an integrated system capable of protecting the shared virtualised system and evaluating the performance characteristics of the protection system?
 - Experiments were conducted in Chapter 6 confirming the possibility of using the solutions proposed in the previous chapters to design a protection system. Also, experiments were carried out to evaluate the overhead systems used in the experiments in various scenarios and the possibility of improving the protection system.

Unlike previous works where detection mechanisms were built using hardware performance counters and relying just on these counters for detection, a methodology has been proposed

to demonstrate the benefit and possibility of effectively building detection and protection mechanisms using memory deduplication and a logistic regression model for classification inside VMs, Thus answering our first and second research questions in Chapter 3.

Chapter 4 has discussed the establishment of a detection and protection mechanism based on a hybrid analysis between dynamic analysis to detect suspicious activities of virtual machines and static analysis to examine disk and RAM images of virtual machines and thus identify the virtual machine that constitutes a threat to be excluded. This mechanism relies on machine learning and deep learning algorithms to analyze results effectively and with high accuracy. The dynamic analysis was initially relied upon to reduce the load on the system. The reliance on static analysis repeatedly without any trigger for the beginning dramatically increases the system's overhead and the long execution time it might take for the static analysis to process the scan. Thus, this chapter answered the third and fourth research questions.

Chapter 5 answered the fifth research question and discussed the use of static analysis in line with the nature of the shared virtualised environment to analyse virtual machines based on logistic regression to detect microarchitectural attacks. One of the well-known antivirus applications was integrated with this mechanism. Two types of scan, Fast scan and Full scan, are designed to put an option to reduce the execution time that static analysis takes in the scan process. The performance characteristics of this mechanism were evaluated. The importance of this mechanism is to use it periodically in long-term intervals to ensure the safety of the shared virtualised system from viruses and microarchitectural attack programs.

The sixth research question has been addressed in Chapter 6. The section discussed the possibility and how to combine the systems that were proposed in the previous chapters to form a detection and protection system against microarchitectural attacks and malicious programs in general. This integrated system relies on multiple defence lines for the shared virtualised system to eliminate microarchitectural attacks and viruses before and during their implementation, neutralize them and eliminate the threats to the shared virtualised system.

7.2 Contribution to knowledge

Owing to the significant development in computer systems and information technology, researchers have given great attention to protection and privacy systems. However, cache side-channel attacks are a serious threat to cloud computing systems. There are many areas related to protection against side-channel attacks that have not been adequately covered and studied, and not all ideas leading to appropriate solutions have been consumed in line with the nature of shared virtual systems. Academic research outputs have an effective and direct impact on the development and growth of information and privacy protection systems and thus reduce the threats to cloud computing. The thesis proposed three methods for detecting and protecting against cache side-channel attacks; each method has its characteristics for mitigating the threats of side-channel attacks in shared virtualised systems. The main contribution of the work is highlighted below.

7.2.1 *Mitigation cache side-channel attacks through memory deduplication:*

The first aspect of the novelty of our research is proposing a method based on a logistic regression algorithm to mitigate side-channel attacks made possible through the use of memory deduplication so that malicious activities are identified by monitoring and analysing the activities that occur on sensitive memory locations belonging to shared cryptographic libraries and sensitive programs in particular. Previous studies depended on monitoring activities by fetching readings from the cache, in general, using hardware performance counters and then analysing them, which makes them unable to work inside the virtual machine unless there is an authorisation from the host to use them, and not all hardware performance counters are supported to work inside the virtual machine [101–103] as discussed in Chapter 3. However, monitoring of memory locations, in particular, is more accurate, as our method monitors the sensitive memory locations and determines whether they are stable or evicted in preparation for the attack and thus detect malicious activities, and at the same time, continuous access into the memory locations for monitoring leads to the obfuscation of the results of the attack

in rare cases of false negatives. The proposed method can be activated and used inside the VM to be self-protected from cache attacks, with no required changes to the virtualised platforms, thus reducing threats while preserving the characteristics of the shared virtualised systems.

7.2.2 Mitigation through Dynamic and Static Analysis:

One novelty aspect in this thesis is proposing a mitigation method that integrates dynamic analysis and static analysis based on machine learning and deep learning algorithms to detect and protect against cache attacks within shared virtualised systems as discussed in Chapter 4. The dynamic analysis uses Linux *Perf* To monitor VMs' activities and then analyze them to classify them into benign and suspicious using the logistic regression classification model. As a second step, the static analysis extracts the executable files from the disk image or the RAM image of the suspicious VM. It then checks whether these files contain opcodes that indicate side-channel attacks using automated reverse engineering. Based on this, the threat level of these files is determined using the Softmax classification algorithm. In contrast, the previous work used static analysis for a particular application analysis regardless of the nature of shared virtualized systems, such as analyzing the application before adding it to an app store. The reason behind this integration of these types of analysis is to reduce the overhead on the system and improve the performance of static analysis so that static analysis is accomplished only for the suspicious virtual machine, so the suspicion is determined by monitoring the activities of the VMs by dynamic analysis. In addition, the method makes it possible to effectively use static analysis within the shared virtualised systems to analyze the RAM and disk of the suspicious VM and thus make the exclusion decision for the VM that threatens the shared virtualised system.

7.2.3 Mitigation through Periodically Long-range Intervals Scan:

The next aspect of the novelty introduced by this thesis relies on proposing a method based on static analysis and machine learning to examine the executable files of VMs in accordance with the nature of the shared virtual environment. This method includes a number

of microarchitectural attacks, and the mechanism is integrated with the ClamAV antivirus. The reason behind this integration is that the shared virtualised systems needs for a mechanism that can detect microarchitectural attacks and viruses (malware) as well as provide adequate protection for shared virtualised systems from malicious programs that may threaten the integrity of the entire system, where none of the antiviruses can detect microarchitectural attacks [26, 64]. The importance of this mechanism lies in conducting periodic long-term inspections to ensure the safety of virtual machines within the system. Two types of scanning, the fast scan and the full scan, were also designed and evaluated as discussed in Chapter 5. In the case of the fast scan, the files are extracted from a VM's userspace, while in the full scan, they are extracted from the kernel space of the entire VM system in the shared virtualised system to be analysed and classified by a logistic regression model.

7.2.4 Integrated Protection System:

Another aspect of novelty introduced by this thesis rests on the integration of the proposed methods into a single mechanism that performs its functions in an integrated manner within the shared virtual systems, thus providing various lines of defence and protection, which reduces the threats of microarchitectural attacks and malicious programs in general within the systems as discussed in Chapter 6. Therefore, we designed compatible solutions and countermeasures to detect suspicious behaviour that could indicate these attacks and develop security controls that maintain the advantages of the multi-tenancy feature while reducing the security risks. Experiments were also carried out with different scenarios on different host devices to evaluate the performance characteristics of the protection system that integrates the proposed solutions in the previous chapters, confirming the possibility of utilising it as a comprehensive solution to mitigate the threats of microarchitecture attacks in general and from side-channel attacks in particular.

7.3 Limitations and Future Work

The mechanism proposed in the thesis relies on various approaches to meet the needs of shared virtual systems for adequate protection. However, some limitations may affect one method over the other. These limitations will be focused on in future work to mitigate their impact and on the general protection of the system. Our proposed mechanism is flexible enough to be compatible with new tools that can be added to address the limitations. This PhD research is restricted by time constraints, which did not allow the realization of all the promising directions, however, they are considered as opportunities to achieve them in future work. Below we list some of the opportunities for extending this research in future work.

Code Obfuscation:

It is one of the techniques an attacker can take to bypass static analysis of executable files. The attacker encrypts the attack program until its execution or transforms it into a unique structure, making it difficult for reverse engineering mechanisms to identify it [133]. Since our mechanism relied on a different set of methods and integrated lines of defence, the negative impact on our protection mechanism will be partial, as only the parts that depend on static analysis will be affected. However, many code deobfuscation techniques can be utilised in conjunction with our mechanisms to address this partial limitation in our mechanism.

Intel Software Guard Extensions (Intel SGX) to Hide Malware:

This approach may be more serious than the previously described technique because it misuses the Software Guard Extensions that allows the user to allocate a private area in the memory called enclave that is designed to protect applications from high-level privileged processes, making the Malware within this protected area hidden from the mechanisms of static and dynamic analysis, which depends on the hardware performance counters readings because these counters are not affected when the attack is executed within SGX enclave [134]. Since the first method in our mechanism does not depend on the hardware performance counters to monitor the activities inside the virtualised systems, it will not be negatively affected, as will the rest of the methods that make up our mechanism. Therefore, it will have a partial impact on the mechanism, whereas it is necessary to solve such an issue because it weakens

the performance of our protection mechanism.

Further Experiments and Evaluations:

Aspects that can be extended in the thesis are further experiments and evaluations. Although experiments have been performed on multiple scenarios and evaluated comprehensively, some parts can be improved, such as improving the duration of static analysis to accomplish the full scan of VM's executable files. More experiments and evaluations are necessary to continue developing our mechanism to keep pace with new threats and issues in shared virtual systems. It is also necessary to evaluate the proposed solutions on virtualised systems that have significant computing power in order to evaluate the system overhead and the percentage of processor usage for these large systems. However, the evaluations were conducted for the proposed solutions in different scenarios were sufficient to some extent to prove their effectiveness.

At the end of the doctoral journey, this thesis presents how to establish a security control system based on machine learning to detect and protect the shared virtualised systems against side-channel attacks and malware while preserving these systems' nature and features. When evaluating the objectives and research questions specified at the beginning of the thesis, we find that the results are generally promising, and the outputs of the hypotheses that have been tested are significant findings. However, this is considered a start of opportunities and participation in possible improvements and new explorations in future work in this field.

Availability of code and dataset. The code and datasets are uploaded on GitHub [Click Here](#).

The End...

BIBLIOGRAPHY

- [1] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, “A survey of microarchitectural timing attacks and countermeasures on contemporary hardware,” *Journal of Cryptographic Engineering*, vol. 8, no. 1, pp. 1–27, 2018.
- [2] “Intel PMU performance.” [Online]. Available: <https://wiki.opnfv.org/display/fastpath/PMU>
- [3] “Intel® 64 and IA32 architectures performance monitoring events.” [Online]. Available: <https://usermanual.wiki/Document/335279performancemonitoringeventsguide.2005880979/help>
- [4] 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.
- [5] P. Mell, T. Grance *et al.*, *The NIST definition of cloud computing*. Computer Security Division, Information Technology Laboratory, National, 2011.
- [6] H. Takabi, J. B. Joshi, and G.-J. Ahn, “Security and privacy challenges in cloud computing environments,” *IEEE Security & Privacy*, vol. 8, no. 6, pp. 24–31, 2010.
- [7] “Gartner identifies the top 10 strategic technology trends for 2020.” [Online]. Available: <https://www.gartner.com/en/newsroom/press-releases/2019-10-21-gartner-identifies-the-top-10-strategic-technology-trends-for-2020>

BIBLIOGRAPHY

- [8] K. Hashizume, D. G. Rosado, E. Fernández-Medina, and E. B. Fernandez, "An analysis of security issues for cloud computing," *Journal of internet services and applications*, vol. 4, no. 1, pp. 1–13, 2013.
- [9] M. Kazim and S. Y. Zhu, "A survey on top security threats in cloud computing." Science and Information (SAI) Organization Ltd., 2015.
- [10] G. Garrison, S. Kim, and R. L. Wakefield, "Success factors for deploying cloud computing," *Communications of the ACM*, vol. 55, no. 9, pp. 62–68, 2012.
- [11] H. Aljahdali, P. Townend, and J. Xu, "Enhancing multi-tenancy security in the cloud iaas model over public deployment," in *2013 IEEE Seventh International Symposium on Service-Oriented System Engineering*. IEEE, 2013, pp. 385–390.
- [12] S. Saxena, G. Sanyal, S. Srivastava, and R. Amin, "Preventing from cross-vm side-channel attack using new replacement method," *Wireless Personal Communications*, vol. 97, no. 3, pp. 4827–4854, 2017.
- [13] H. AlJahdali, A. Albatli, P. Garraghan, P. Townend, L. Lau, and J. Xu, "Multi-tenancy in cloud computing," in *2014 IEEE 8th International Symposium on Service Oriented System Engineering*. IEEE, 2014, pp. 344–351.
- [14] A. Donevski, S. Ristov, and M. Gusev, "Security assessment of virtual machines in open source clouds," in *2013 36th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE, 2013, pp. 1094–1099.
- [15] W. A. Jansen, T. Grance *et al.*, *Guidelines on security and privacy in public cloud computing*. Computer Security Division, Information Technology Laboratory, National, 2011.
- [16] S. K. Abd, R. T. Salih, S. Al-Haddad, F. Hashim, A. B. H. Abdullah, and S. Yussof, "Cloud computing security risks with authorization access for secure multi-tenancy

- based on aaas protocol,” in *TENCON 2015-2015 IEEE Region 10 Conference*. IEEE, 2015, pp. 1–5.
- [17] P. Saripalli and B. Walters, “Quirc: A quantitative impact and risk assessment framework for cloud security,” in *2010 IEEE 3rd international conference on cloud computing*. IEEE, 2010, pp. 280–288.
- [18] H. Dey, R. Islam, and H. Arif, “An integrated model to make cloud authentication and multi-tenancy more secure,” in *2019 International Conference on Robotics, Electrical and Signal Processing Techniques (ICREST)*. IEEE, 2019, pp. 502–506.
- [19] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, “Cross-vm side channels and their use to extract private keys,” in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 305–316.
- [20] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar, “Wait a minute! a fast, cross-vm attack on aes,” in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2014, pp. 299–319.
- [21] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, “Flush+ flush: a fast and stealthy cache attack,” in *Int. Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2016, pp. 279–299.
- [22] D. Philippe-Jankovic and T. A. Zia, “Breaking vm isolation—an in-depth look into the cross vm flush reload cache timing attack,” *Int. J. of Computer Science and Network Security (IJCSNS)*, vol. 17, no. 2, p. 181, 2017.
- [23] M.-M. Bazm, T. Sautereau, M. Lacoste, M. Sudholt, and J.-M. Menaud, “Cache-based side-channel attacks detection through intel cache monitoring technology and hardware performance counters,” in *3rd Int. Conf. on Fog and Mobile Edge Computing (FMEC)*, 2018, pp. 7–12.

BIBLIOGRAPHY

- [24] M. Chiappetta, E. Savas, and C. Yilmaz, "Real time detection of cache-based side-channel attacks using hardware performance counters," *Applied Soft Computing*, vol. 49, pp. 1162–1174, 2016.
- [25] B. Gulmezoglu, A. Moghimi, T. Eisenbarth, and B. Sunar, "Fortuneteller: Predicting microarchitectural attacks via unsupervised deep learning," *arXiv preprint arXiv:1907.03651*, 2019.
- [26] G. Irazoqui, T. Eisenbarth, and B. Sunar, "Mascot: preventing microarchitectural attacks before distribution," in *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, 2018, pp. 377–388.
- [27] M. Mushtaq, A. Akram, M. K. Bhatti, R. N. B. Rais, V. Lapotre, and G. Gogniat, "Run-time detection of prime+ probe side-channel attack on aes encryption algorithm," in *Global Information Infrastructure and Networking Symp. (GIIS)*, 2018, pp. 1–5.
- [28] Y. Zhang and M. K. Reiter, "Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 827–838.
- [29] B. C. Vattikonda, S. Das, and H. Shacham, "Eliminating fine grained timers in xen," in *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*. ACM, 2011, pp. 41–46.
- [30] T. Zhang, Y. Zhang, and R. B. Lee, "Cloudradar: A real-time side-channel attack detection system in clouds," in *Int. Symp. on Research in Attacks, Intrusions, and Defenses*, 2016, pp. 118–140.
- [31] R. R. Chowdhury, "Security in cloud computing," *International Journal of Computer Applications*, vol. 96, no. 15, 2014.
- [32] C. Barron, H. Yu, and J. Zhan, "Cloud computing security case studies and research," in *Proceedings of the world congress on engineering*, vol. 2, no. 2, 2013, pp. 1–6.

- [33] M. A. Nadeem, "Cloud computing: security issues and challenges," *Journal of Wireless Communications*, vol. 1, no. 1, pp. 10–15, 2016.
- [34] B. Alouffi, M. Hasnain, A. Alharbi, W. Alosaimi, H. Alyami, and M. Ayaz, "A systematic literature review on cloud computing security: threats and mitigation strategies," *IEEE Access*, vol. 9, pp. 57 792–57 807, 2021.
- [35] M. Irfan, M. Usman, Y. Zhuang, and S. Fong, "A critical review of security threats in cloud computing," in *2015 3rd International Symposium on Computational and Business Intelligence (ISCBI)*. IEEE, 2015, pp. 105–111.
- [36] S. Kuyoro, F. Ibikunle, and O. Awodele, "Cloud computing security issues and challenges," *International Journal of Computer Networks (IJCN)*, vol. 3, no. 5, pp. 247–255, 2011.
- [37] D. Chen and H. Zhao, "Data security and privacy protection issues in cloud computing," in *2012 International Conference on Computer Science and Electronics Engineering*, vol. 1. IEEE, 2012, pp. 647–651.
- [38] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 2009, pp. 199–212.
- [39] V. Varadarajan, Y. Zhang, T. Ristenpart, and M. Swift, "A placement vulnerability study in multi-tenant public clouds," in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, 2015, pp. 913–928.
- [40] A. Shahzad and A. Litchfield, "Virtualization technology: Cross-vm cache side channel attacks make it vulnerable," in *Proceedings of the Australasian conference on information systems*, 2015.
- [41] Y. Han, "Defending against co-resident attacks in cloud computing," Ph.D. dissertation, 2015.

BIBLIOGRAPHY

- [42] S. A. Hussain, M. Fatima, A. Saeed, I. Raza, and R. K. Shahzad, "Multilevel classification of security concerns in cloud computing," *Applied Computing and Informatics*, vol. 13, no. 1, pp. 57–65, 2017.
- [43] K. Popović and Ž. Hocenski, "Cloud computing security issues and challenges," in *The 33rd international convention mipro*. IEEE, 2010, pp. 344–349.
- [44] I. M. Khalil, A. Khreishah, and M. Azeem, "Cloud computing security: A survey," *Computers*, vol. 3, no. 1, pp. 1–35, 2014.
- [45] S. Anwar, Z. Inayat, M. F. Zolkipli, J. M. Zain, A. Gani, N. B. Anuar, M. K. Khan, and V. Chang, "Cross-vm cache-based side channel attacks and proposed prevention mechanisms: A survey," *Journal of Network and Computer Applications*, vol. 93, pp. 259–279, 2017.
- [46] R. Montasari, A. Hosseinian-Far, R. Hill, F. Montaseri, M. Sharma, and S. Shabbir, "Are timing-based side-channel attacks feasible in shared, modern computing hardware?" *International Journal of Organizational and Collective Intelligence (IJOICI)*, vol. 8, no. 2, pp. 32–59, 2018.
- [47] A. Facon, S. Guilley, M. Lec'Hvien, A. Schaub, and Y. Souissi, "Detecting cache-timing vulnerabilities in post-quantum cryptography algorithms," in *2018 IEEE 3rd International Verification and Security Workshop (IVSW)*. IEEE, 2018, pp. 7–12.
- [48] L. Yan, Y. Guo, X. Chen, and H. Mei, "A study on power side channels on mobile devices," in *Proceedings of the 7th Asia-Pacific Symposium on Internetware*. ACM, 2015, pp. 30–38.
- [49] K. Suzaki, K. Iijima, T. Yagi, and C. Artho, "Memory deduplication as a threat to the guest os," in *Proceedings of the Fourth European Workshop on System Security*. ACM, 2011, p. 1.
- [50] D. J. Bernstein, "Cache-timing attacks on aes." Citeseer, 2005.

- [51] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar, "Fine grain cross-vm attacks on xen and vmware," in *2014 IEEE Fourth International Conference on Big Data and Cloud Computing*. IEEE, 2014, pp. 737–744.
- [52] D. Jayasinghe, J. Fernando, R. Herath, and R. Ragel, "Remote cache timing attack on advanced encryption standard and countermeasures," in *2010 Fifth International Conference on Information and Automation for Sustainability*. IEEE, 2010, pp. 177–182.
- [53] A. C. Atici, C. Yilmaz, and E. Savaş, "Cache-timing attacks without a profiling phase," *Turkish Journal of Electrical Engineering & Computer Sciences*, vol. 26, no. 4, pp. 1953–1966, 2018.
- [54] Y. Yarom and N. Benger, "Recovering openssl ecdsa nonces using the flush+ reload cache side-channel attack." *IACR Cryptology ePrint Archive*, vol. 2014, p. 140, 2014.
- [55] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games—bringing access-based cache attacks on aes to practice," in *2011 IEEE Symposium on Security and Privacy*. IEEE, 2011, pp. 490–505.
- [56] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar, "Lucky 13 strikes back," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*. ACM, 2015, pp. 85–96.
- [57] 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.
- [58] C. Pereida García, B. B. Brumley, and Y. Yarom, "Make sure dsa signing exponentiations really are constant-time," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 1639–1650.

BIBLIOGRAPHY

- [59] 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.
- [60] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1–19.
- [61] C. Tang, Z. Liu, C. Ma, J. Ge, and C. Tu, "Secflush: A hardware/software collaborative design for real-time detection and defense against flush-based cache attacks," in *International Conference on Information and Communications Security*. Springer, 2019, pp. 251–268.
- [62] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin *et al.*, "Meltdown: Reading kernel memory from user space," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 973–990.
- [63] S. Bhattacharya and D. Mukhopadhyay, "Curious case of rowhammer: flipping secret exponent bits using timing analysis," in *International Conference on Cryptographic Hardware and Embedded Systems*. Springer, 2016, pp. 602–624.
- [64] G. Irazoqui, T. Eisenbarth, and B. Sunar, "Mascot: Stopping microarchitectural attacks before execution." *IACR Cryptol. ePrint Arch.*, vol. 2016, p. 1196, 2016.
- [65] "MFENCE - memory fence." [Online]. Available: <https://www.felixcloutier.com/x86/mfence>
- [66] "LFENCE — load fence." [Online]. Available: <https://www.felixcloutier.com/x86/lfence>
- [67] "CLFLUSH — flush cache line." [Online]. Available: <https://www.felixcloutier.com/x86/clflush>
- [68] "LOCK — assert lock signal prefix." [Online]. Available: <https://www.felixcloutier.com/x86/lock>

- [69] “MOVNTI — store doubleword using non-temporal hint.” [Online]. Available: <https://www.felixcloutier.com/x86/movnti>
- [70] “MOVNTDQ — store packed integers using non-temporal hint.” [Online]. Available: <https://www.felixcloutier.com/x86/movntdq>
- [71] C. Rebeiro, D. Mukhopadhyay, and S. Bhattacharya, *Timing channels in cryptography: a micro-architectural perspective*. Springer, 2014.
- [72] Y. Tsiounis and M. Yung, “On the security of elgamal based encryption,” in *International Workshop on Public Key Cryptography*. Springer, 1998, pp. 117–134.
- [73] W. Stallings, *Cryptography and network security, 4/E*. Pearson Education India, 2006.
- [74] N. Muyinda, “Elliptic curve cryptography,” *African Institute for Mathematical Sciences (AIMS)*, 2009.
- [75] K. Ren, C. Wang, and Q. Wang, “Security challenges for the public cloud,” *IEEE Internet Computing*, vol. 16, no. 1, pp. 69–73, 2012.
- [76] E. Keller, J. Szefer, J. Rexford, and R. B. Lee, “Nohype: virtualized cloud infrastructure without the virtualization,” in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3. ACM, 2010, pp. 350–361.
- [77] A. Jasti, P. Shah, R. Nagaraj, and R. Pendse, “Security in multi-tenancy cloud,” in *44th Annual 2010 IEEE International Carnahan Conference on Security Technology*. IEEE, 2010, pp. 35–41.
- [78] L. Malhotra, D. Agarwal, and A. Jaiswal, “Virtualization in cloud computing,” *J Inform Tech Softw Eng*, vol. 4, no. 2, p. 136, 2014.
- [79] W. Stallings, *Computer organization and architecture: designing for performance*. Pearson Education India, 2018.
- [80] S. P. Dandamudi, *Fundamentals of computer organization and design*. Springer, 2014.

BIBLIOGRAPHY

- [81] Q. Javaid, A. Zafar, M. Awais, and M. A. Shah, "Cache memory: An analysis on replacement algorithms and optimization techniques," *Mehran University Research Journal of Engineering & Technology*, vol. 36, no. 4, pp. 831–840, 2017.
- [82] Y. Cui and X. Cheng, "Abusing cache line dirty states to leak information in commercial processors," *arXiv preprint arXiv:2104.08559*, 2021.
- [83] J. E. Smith and J. R. Goodman, "A study of instruction cache organizations and replacement policies," *ACM SIGARCH Computer Architecture News*, vol. 11, no. 3, pp. 132–137, 1983.
- [84] W. Xia, H. Jiang, D. Feng, F. Douglass, P. Shilane, Y. Hua, M. Fu, Y. Zhang, and Y. Zhou, "A comprehensive study of the past, present, and future of data deduplication," *Proceedings of the IEEE*, vol. 104, no. 9, pp. 1681–1710, 2016.
- [85] J. Xiao, Z. Xu, H. Huang, and H. Wang, "Security implications of memory deduplication in a virtualized environment," in *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2013, pp. 1–12.
- [86] J. Lindemann and M. Fischer, "A memory-deduplication side-channel attack to detect applications in co-resident virtual machines," in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. ACM, 2018, pp. 183–192.
- [87] H. Wang, H. Sayadi, S. Rafatirad, A. Sasan, and H. Homayoun, "Scarf: Detecting side-channel attacks at real-time using low-level hardware features," in *IEEE 26th Int. Symp. on On-Line Testing and Robust System Design (IOLTS)*, 2020, pp. 1–6.
- [88] 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*, vol. 10, no. 3, p. 984, 2020.
- [89] X. Wang, J. Zhang, and A. Zhang, "Machine-learning-based malware detection for virtual machine by analyzing opcode sequence," in *International Conference on Brain Inspired Cognitive Systems*. Springer, 2018, pp. 717–726.

- [90] Y. Han, J. Chan, T. Alpcan, and C. Leckie, "Virtual machine allocation policies against co-resident attacks in cloud computing," in *2014 IEEE International Conference on Communications (ICC)*. IEEE, 2014, pp. 786–792.
- [91] —, "Using virtual machine allocation policies to defend against co-resident attacks in cloud computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 14, no. 1, pp. 95–108, 2015.
- [92] J. Shi, X. Song, H. Chen, and B. Zang, "Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring," in *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 2011, pp. 194–199.
- [93] Y. Yarom, Q. Ge, F. Liu, R. B. Lee, and G. Heiser, "Mapping the intel last-level cache." *IACR Cryptology ePrint Archive*, vol. 2015, p. 905, 2015.
- [94] J. Depoix and P. Altmeyer, "Detecting spectre attacks by identifying cache side-channel attacks using machine learning," *Advanced Microkernel Operating Systems*, vol. 75, 2018.
- [95] Z. B. Aweke, S. F. Yitbarek, R. Qiao, R. Das, M. Hicks, Y. Oren, and T. Austin, "Anvil: Software-based protection against next-generation rowhammer attacks," *ACM SIGPLAN Notices*, vol. 51, no. 4, pp. 743–755, 2016.
- [96] A. Chakraborty, M. Alam, and D. Mukhopadhyay, "Deep learning based diagnostics for rowhammer protection of dram chips," in *2019 IEEE 28th Asian Test Symposium (ATS)*. IEEE, 2019, pp. 86–865.
- [97] C. Malone, M. Zahran, and R. Karri, "Are hardware performance counters a cost effective way for integrity checking of programs," 10 2011.
- [98] J. Wolfe, X. Jin, T. Bahr, and N. Holzer, "Application of softmax regression and its validation for spectral-based land cover mapping," *The International Archives of*

BIBLIOGRAPHY

- Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. 42, p. 455, 2017.
- [99] T. Hornby, "Side-channel attacks on everyday applications: Distinguishing inputs with flush+reload," *BlackHat USA*, 2016.
- [100] Y. Yarom, "Mastik: A micro-architectural side-channel toolkit," <https://cs.adelaide.edu.au/~yval/Mastik/>.
- [101] Intel, "Virtual targets," <https://software.intel.com/content/www/us/en/develop/documentation/vtune-help/top/set-up-analysis-target/on-virtual-machine.html>.
- [102] J. Du, N. Sehwat, and W. Zwaenepoel, "Performance profiling of virtual machines," in *7th ACM SIGPLAN/SIGOPS Int. Conf. on Virtual Execution Environments*, 2011, pp. 3–14.
- [103] R. Hat, "2.2. Virtual performance monitoring unit (vPMU) red hat enterprise linux 7," https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/virtualization_tuning_and_optimization_guide/sect-virtualization_tuning_optimization_guide-monitoring_tools-vpmu.
- [104] A. Shabtai, R. Moskovitch, C. Feher, S. Dolev, and Y. Elovici, "Detecting unknown malicious code by applying classification techniques on opcode patterns," *Security Informatics*, vol. 1, no. 1, pp. 1–22, 2012.
- [105] R. Patil, H. Dudeja, and C. Modi, "Designing in-vm-assisted lightweight agent-based malware detection framework for securing virtual machines in cloud computing," *International Journal of Information Security*, vol. 19, no. 2, pp. 147–162, 2020.
- [106] X. Xie and W. Wang, "Lightweight examination of dll environments in virtual machines to detect malware," in *Proceedings of the 4th ACM International Workshop on Security in Cloud Computing*, 2016, pp. 10–16.

- [107] L. Joseph and R. Mukesh, "Detection of malware attacks on virtual machines for a self-heal approach in cloud computing using vm snapshots," *Journal of Communications Software and Systems*, vol. 14, no. 3, pp. 249–257, 2018.
- [108] "Libguestfs tools for accessing and modifying virtual machine disk images," 2020. [Online]. Available: <https://libguestfs.org/>
- [109] D. Nanni, "How to mount qcow2 disk image on Linux," 2020. [Online]. Available: <https://www.xmodulo.com/mount-qcow2-disk-image-linux.html>
- [110] Microsoft, "Microsoft/avml: Avml - acquire volatile memory for linux," 2021. [Online]. Available: <https://github.com/microsoft/avml>
- [111] Volatilityfoundation, "Volatilityfoundation/volatility: An advanced memory forensics framework," 2020. [Online]. Available: <https://github.com/volatilityfoundation/volatility>
- [112] "The r2pipe apis: radare2-r2pipe," 2021. [Online]. Available: <https://github.com/radareorg/radare2-r2pipe>
- [113] M. Chiappetta, E. Savas, and C. Yilmaz, "Xlate:," <https://www.vusec.net/projects/xlate/>, 2020.
- [114] D. Gruss, R. Spreitzer, and S. Mangard, "Cache Template Attacks," 2019. [Online]. Available: https://github.com/IAIK/cache_template_attacks
- [115] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush + Flush," 2019. [Online]. Available: https://github.com/IAIK/flush_flush
- [116] Nepoche, "Flush and Reload Cache Side Channel Attack," 2017. [Online]. Available: <https://github.com/nepoche/Flush-Reload>
- [117] K. Akash, "Flush-Reload-Attack," 2018. [Online]. Available: <https://github.com/AkashWorld/Flush-Reload-Attack>

BIBLIOGRAPHY

- [118] H. Pasic, "University project - Side Channel Attack (Cache attack)," 2019. [Online]. Available: <https://github.com/HarisPasic/SideChannelAttack>
- [119] J. Park, "CSCA (Crypto Side Channel Attack)," 2018. [Online]. Available: <https://github.com/jinb-park/crypto-side-channel-attack>
- [120] Nagnagnet, "Prime+Probe is a last-level cache side-channel attack." 2018. [Online]. Available: <https://github.com/nagnagnet/PrimeProbe>
- [121] M. Mushtaq, J. Bricq, M. K. Bhatti, A. Akram, V. Lapotre, G. Gogniat, and P. Benoit, "Whisper: A tool for run-time detection of side-channel attacks," *IEEE Access*, vol. 8, pp. 83 871–83 900, 2020.
- [122] T. K. Lengyel, J. Neumann, S. Maresca, B. D. Payne, and A. Kiayias, "Virtual machine introspection in a hybrid honeypot architecture." in *CSET*, 2012.
- [123] "Playground/Objdump_Cfg," 2015. [Online]. Available: https://github.com/zestrada/playground/tree/master/objdump_cfg
- [124] F. Wu, "Spectre Attack Demo (i5-3320M and Intel Xeon v3)," 2018. [Online]. Available: <https://github.com/flxwu/spectre-attack-demo>
- [125] K. Paimani, "Meltdown and Spectre attacks," 2019. [Online]. Available: <https://github.com/kianenigma/meltdown-spectre>
- [126] R. Crosby, "Proof of concept code for the Spectre CPU exploit," 2018. [Online]. Available: <https://github.com/crozone/SpectrePoC>
- [127] M. Eichlseder, "Meltdown Proof-of-Concept," 2019. [Online]. Available: <https://github.com/IAIK/meltdown>
- [128] S. Xing, "Meltdown attack demo," 2018. [Online]. Available: <https://github.com/SamuelXing/MeltdownDemo>
- [129] D. Gruss, "Program for testing for the DRAM "rowhammer" problem using eviction," 2017. [Online]. Available: <https://github.com/IAIK/rowhammerjs>

- [130] Zaweke, "CLFLUSH-free rowhammer," 2020. [Online]. Available: <https://github.com/zaweke/rowhammer>
- [131] "Clam Antivirus." [Online]. Available: <https://www.clamav.net>
- [132] Ö. Aslan, "Performance comparison of static malware analysis tools versus antivirus scanners to detect malware," in *International Multidisciplinary Studies Congress (IMSC)*, 2017.
- [133] S. Talukder and Z. Talukder, "A survey on malware detection and analysis tools," *International Journal of Network Security & Its Applications (IJNSA) Vol*, vol. 12, 2020.
- [134] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoechl, and Y. Yarom, "Another flip in the wall of rowhammer defenses," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 245–261.

