POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Politecnico
di Torino

Master's Degree Thesis

Malware detection using hardware performance counters on RISC-V based cloud servers

Supervisors

Prof. STEFANO DI CARLO

Prof. ALESSANDRO SAVINO

Dr. CRISTIANO P. CHANET

Candidate

DAVIDE BRUNO

October,2024

Summary

Nowadays, cloud computing is widely utilised in the Information Technology (IT) industry to support critical infrastructure and services. Cloud services play a crucial role across private, public, and commercial sectors, where many are expected to operate continuously and support critical infrastructure. As cloud services become more essential, they face increasing security threats, both from known vulnerabilities and emerging challenges. To remain resilient, cloud infrastructures must be protected not only against familiar threats but also against unknown, zero-day malicious software. As a result, ensuring robust security has become increasingly crucial.
In this context, the Reduced Instruction Set Computer Fifth Version(RISC-V) architecture has seen growing adoption for its flexibility and scalability in cloud environments. However, the increased use of this open-source hardware also amplifies the risk of sophisticated malware attacks targeting cloud infrastructure.
This thesis focuses on the development of a hardware-based malware detection framework for RISC-V processors in cloud environments. By collecting and analysing Hardware Performance Counters (HPCs) data during the execution of applications and malware, a machine learning-based system is developed to differentiate between benign and malicious software.
The methodology involves using the Gem5 simulator to gather detailed HPC metrics from RISC-V processors running various applications and malware, followed by the use of multiple machine learning algorithms to detect benign and malicious behaviour. The thesis evaluates the effectiveness of different machine learning models. Preliminary results demonstrate the potential of combining HPC data with machine learning techniques to achieve over 80% accuracy in malware detection. This approach could provide a lightweight and scalable solution for enhancing the security of cloud services using RISC-V architecture.

Acknowledgements

"To my lovely family"

"Nothing behind me, everything ahead of me, as is ever so on the road."
Jack Kerouac, On the Road

I am deeply grateful to my supervisors, Professor Stefano Di Carlo and Professor Alessandro Savino for their unwavering guidance, feedback, and support during my research. Their expertise played a crucial role in completing this thesis.

I am also extremely thankful to my co-supervisor, Dr Cristiano Pegoraro Chenet , for his constructive feedback and essential suggestions that enhanced the quality of my work.

My deep appreciation goes to my family for their endless love, patience, and belief in me, which made this achievement possible. A special thanks to my mother and father for believing in me more than anyone else and for giving me the strength to keep pushing forward, time and time again. I couldn't have asked for anything better than their unwavering support.

To my lifelong friends, thank you for standing by me through every stage of life. Your unwavering support and the countless memories we have shared have been a constant source of strength. I couldn't have asked for better companions throughout this journey.

To my newer friends, who joined me during this chapter of my life, your encouragement and understanding have been invaluable. Whether through study sessions or our long long long breaks in the study room, you have helped make this experience far more enjoyable.

Lastly, my heartfelt thanks go to everyone who supported me during this tough journey.

And of course, I can't forget my beloved dogs. Although they can't read (at least for now), I'm sure they understand my love for them :)

Table of Contents

List of Tables

List of Figures

List Of Listings

Acronyms

IT
Information Technology

AI
Artificial intelligence

DL
Deep learning

ML
Machine learning

IOT
Internet of Things

HMD
Hardware-based Malware Detection

HPM
Hardware Performance Monitoring tool

RISC-V
Reduced Instruction Set Computer, Fifth Version

HPC
Hardware Performance Counter

CSR
Control Status Register

ISA

Instruction Set Architecture

CPU

Central Processing Unit

RAM

Random-Access Memory

IDS

Intrusion Detection System

DDOS

Distributed Denial of Service

OS

Operating System

FS

Feature Selection

PCA

Principal Component Analysis

OC-SVM

One-Class Support Vector Machine

LOF

Local Outlier Factor

IF

Isolation Fores

EE

Elliptic Envelope

Chapter 1

General Introduction

1.1 Problem Statement

Nowadays, **Cloud Computing** has become a foundational pillar of modern IT, playing a crucial role in transforming how businesses operate and deliver services. Organizations worldwide have rapidly adopted cloud solutions as the core of their operations, drawn by the cloud's unique scalability, flexibility, and cost-effectiveness. The on-demand nature of cloud computing enables businesses to expand their infrastructure and enhance performance while minimizing costs. Consequently, cloud adoption has grown exponentially across diverse industries—from finance and healthcare to retail—as companies leverage cloud platforms to manage data, run critical applications, and deliver seamless services to their customers.

According to the International Data Corporation (IDC), the global public cloud market was valued at approximately $669 billion in 2023, reflecting a 19.9% increase compared to 2022[1]. It is projected to reach $1.20 trillion by 2028, with a compound annual growth rate (CAGR) of 15% during this period. Industry leaders such as Amazon Web Services (AWS), Microsoft Azure, and Google Cloud are at

the forefront, collectively capturing a significant portion of the market. For instance, AWS reported cloud-related revenue of $90.8 billion in 2023, while Microsoft's Intelligent Cloud segment, which includes Azure, exceeded $68 billion during the same period[2].

New trends such as **Artificial Intelligence (AI)**, **Machine Learning (ML)**, and the widespread adoption of **Internet of Things devices(IOT)** are driving unprecedented demand for cloud computing resources. These technologies often require significant processing power, storage capabilities, and scalability to train complex models and analyze large datasets. Cloud platforms provide the necessary infrastructure to support these demanding workloads, making them essential for organizations leveraging cutting-edge technologies. As a result, the cloud market is experiencing rapid growth, with industry leaders like AWS, Microsoft Azure, and Google Cloud investing heavily in expanding their capabilities and offerings. This ongoing evolution is expected to continue, further solidifying the role of cloud computing as a foundational technology for businesses across various industries.

As the cloud market continues to grow, organizations need to adopt robust security measures to protect their data and applications from emerging threats. Despite continuous advancements in cloud technologies, the cloud still presents several unique security challenges. One particularly critical issue is the effective identification and detection of malware, as it often serves as the first line of defence against more significant security threats, such as Distributed Denial of Service (DDoS) attacks or data breaches[3].

Malware, short for malicious software, is designed to disrupt, damage, or gain unauthorised access to systems, often stealing sensitive information or corrupting data[4][5].

Malware targeting cloud environments has become a serious concern, often leading to data breaches, service disruptions, and substantial financial losses. Common types of malware targeting cloud environments include ransomware, trojans, cryptojacking, and botnets. To address the escalating threat of malware in cloud environments, organizations are increasingly turning to advanced security solutions specifically designed to safeguard cloud infrastructures. In this thesis, the approach used to address this challenge is **Hardware-based Malware Detection (HMD)**. It involves the dynamic analysis of micro-architectural events within a processor, utilising ML algorithms to differentiate between benign and malicious applications.

1.2 Objectives

In today's digital landscape, many software-based malware detection solutions, such as traditional antivirus programs, are no longer sufficient for protecting cloud-computing environments, given the constantly evolving and dynamic nature of these systems. These solutions primarily rely on signature-based detection methods, which can easily be fooled by the clever techniques used by advanced malware. This type of malware often employs obfuscation and other misleading strategies to evade detection, resulting in a growing number of successful attacks within cloud environments. Additionally, software-based solutions often are not fast enough for real-time malware detection, especially in safety-critical systems where immediate threat response is essential.

This thesis aims to address these significant limitations by evaluating a **HMD Framework** on the **Reduced Instruction Set Computer Fifth Version (RISC-V)** architecture within a complete computer system running a Linux operating system and real-world applications. This represents a novel contribution, as such an approach has not been explored in previous research.

The evaluated framework is illustrated in Figure 1.1.

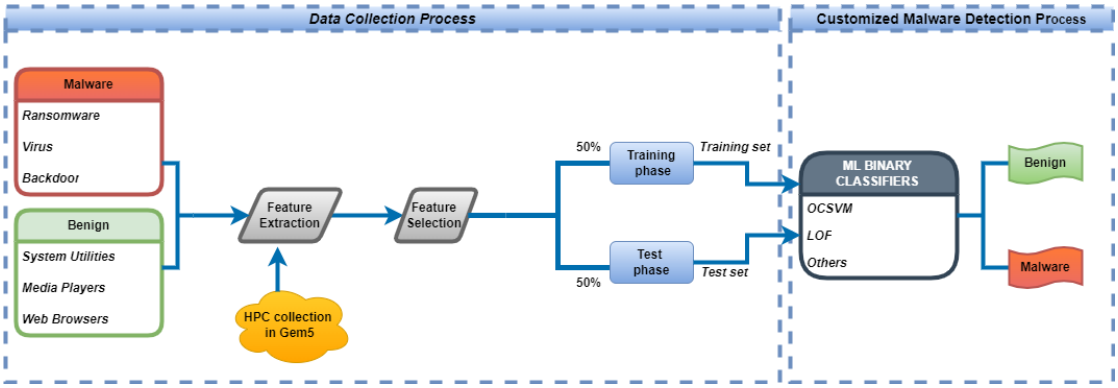

Figure 1.1: Evaluated hardware-based detection framework

RISC-V is an open-source **Instruction Set Architecture (ISA)** chosen for its transparency, modularity, and flexibility. This flexibility allows organizations and companies to tailor solutions according to their unique requirements.

The evaluated framework leverages **Hardware Performance Counters (HPCs)**, which are already integrated into modern processors, to detect malware more effectively than traditional software solutions. These counters provide valuable insights into the behaviour of programs, which often exhibit distinct phase behaviours. By monitoring program behaviour through HPCs, it is possible to identify time-behavioural patterns in micro-architectural events, thus allowing for the distinction between benign applications and malware. Examples of these micro-architectural events include cache accesses, instruction counts, and branch prediction outcomes.

All experiments are carried out using a specific simulator:**Gem5**[6]. This powerful tool is capable of simulating complete computer system architectures, running a Linux operating system and real applications, and extracting various hardware metrics. Additionally, it allows users to specify the micro-architectural events to

collect during simulations. The data collected from HPCs for both selected malware and benign applications will be used to create multiple **training** and **test sets**, which will serve as inputs for various **ML algorithms**. This approach significantly improves malware detection accuracy and reduces false positives, especially in cloud environments where the dynamic nature of the infrastructure can make traditional detection methods less effective.

In conclusion, the investigated framework has the potential to significantly reduce the number of successful malware attacks and mitigate the financial losses associated with data breaches, ultimately contributing to the establishment of more secure and resilient cloud computing environments.

1.3 Thesis Structure

To effectively address the challenges of malware detection in cloud computing environments, the thesis is organised into five remaining chapters, each dedicated to a critical aspect of the research.

Chapter 2: Malware Basics. This chapter provides a foundational understanding of malware and its detection. It covers malware classification, various malware detection methods, and the differences between software-based and hardware-based detection techniques. In addition, it discusses measurement metrics, the role of hardware events and performance counters in detecting malicious code, and the strengths and limitations of hardware-based malware detection.

Chapter 3: Simulation Environment. This chapter provides an in-depth overview of the simulation environment employed in the research. It introduces the gem5 simulator, detailing its architecture, configuration, and the specific parameters used to simulate RISC-V processors. The chapter also explains the use of configuration and simulation scripts in setting up experiments and describes the

benchmarks applied to evaluate both benign and malicious software.

Chapter 4: Data Analysis. This chapter outlines the methodologies used for analysing data generated from the simulations. It covers the selection of relevant hardware events, the extraction and selection of features from the HPC data, and the procedures for training and testing ML models.

Chapter 5: Experimental Results. This chapter presents experimental results that evaluate the performance and effectiveness of different ML algorithms in detecting malware. It presents a detailed comparison of these algorithms, highlighting their strengths and limitations across different target application datasets.

Chapter 6: Conclusion and Future Work. The final chapter summarises the findings of the thesis and suggests future research directions. It highlights the successful integration of HPC data with ML techniques while identifying areas for improvement and further exploration to enhance the security of cloud services using RISC-V processors.

Chapter 2

Malware basics

The proliferation of internet-connected devices and the growing sophistication of digital applications have made systems increasingly vulnerable to a wide range of malware-driven cyberattacks. Malicious software, commonly known as **malware**, is a piece of code designed to harm or subvert the functionalities of systems and their users by stealing sensitive information, corrupting files, or engaging in activities to annoy users[4][5].

It represents one of the most significant and persistent security threats to the modern Internet, continually evolving in complexity and scale to exploit new vulnerabilities and target critical systems.

Recent studies underscore the alarming rise of malware incidents and their consequences. According to Statista Research[7], in 2023, 6.06 billion malware attacks were detected worldwide, with the majority occurring in the Asia-Pacific region representing 31% of all reported incidents, with Europe and North America following at 28% and 25%, respectively[8]. Among the most frequently blocked types of malware were worms, viruses, ransomware, trojans, and backdoors. Remarkably, the two primary attack vectors are email and websites, which are used more commonly for phishing attacks[9].

The financial impact of malware is equally alarming. According to Cybersecurity Ventures, cybercrime, largely driven by malware, is projected to cost the global economy over $10.5 trillion annually by 2025[10]. For instance, the CodeRed virus outbreak infected over 359,000 hosts and led to financial losses estimated at $2.4 billion[11]. Beyond financial losses, the effects of malware extend to operational disruptions, reputational damage, and even legal consequences for organisations. These statistics highlight the critical importance of cybersecurity, with the World Economic Forum's 2023 Global Risks Report ranking cyber insecurity as the fifth greatest global threat[12].

Given the devastating impact that malware can inflict on both organisations and individuals, the identification and containment of malicious programs has become a critical priority in cybersecurity. The growing complexity of malware necessitates the development of more advanced detection techniques to ensure a secure cyberspace. However, malware detection is a challenging task, with threats constantly evolving and malware types becoming more diverse.

This chapter examines the current malware detection techniques, discussing their strengths and limitations. Additionally, it highlights the importance of hardware-based malware detection, which is central to this thesis for its ability to address the limitations of traditional software-based methods.

2.1 Malware Classification

The classification of malware depends on several criteria, including the propagation mechanisms, intended objectives, or how it exploits or makes the system vulnerable. Three approaches are generally adopted in the literature: the classical approach, the approach based on the concealment strategy, and the approach based on the data structures manipulated by the malware.

In the classical approach, malware is categorised based on the propagation method.

The main categories are discussed below[13][14][15].

- **Virus**: It attaches to legitimate software, replicating and spreading by infecting files or systems. Viruses typically spread via shared files, email attachments, or removable media, making them a persistent threat in both personal and organizational environments.
- **Worm**: self-replicating malicious code that spreads autonomously across networks without the need for user interaction. By exploiting security vulnerabilities, worms propagate rapidly, often causing network congestion and significant slowdowns as they consume bandwidth and processing resources.
- **Trojan horse**: commonly referred to as Trojan, it is designed to appear as legitimate software. Unlike viruses and worms, Trojans do not replicate themselves. However, they can be equally destructive, often creating backdoors that grant attackers unauthorized access to the infected system.
- **Ransomware**: it encrypts the files on a victim's system and demands a ransom, typically in cryptocurrency for the decryption key, which makes it difficult to track the recipient of the transaction. The widespread use of ransomware has escalated to the point where entire networks or industries can be crippled by a single attack. It is commonly spread through email attachments.
- **Spyware**: it is designed to spy and collect sensitive information about a user's activities without their knowledge or consent. This information, which may include browsing habits, login credentials, and other sensitive data, is then transmitted back to the attacker. They can compromise user privacy, leading to identity theft, financial loss, and other security breaches. Spyware propagates by embedding itself in legitimate software, Trojan horses, or software vulnerabilities.

- **Adware**: it displays unwanted computer advertisements, often in the form of pop-ups or banners. Generally considered less destructive than other forms of malware. Some adware may be bundled with spyware, making it particularly dangerous as it can monitor user activity and steal sensitive information.
- **Back door**: malicious code that opens the system to external entities by-passing the local security policies to allow remote access and control over a network.
- **Key loggers**: malicious code designed to record which keys are pressed on a computer keyboard. Generally used to obtain passwords as well as encryption keys.
- **Rootkits**: set of malicious applications designed to conceal the presence of malicious software within a system, enabling attackers to maintain persistent, unauthorized access. Rootkits often operate at a low level, deep within the operating system, making them difficult to detect with conventional antivirus tools.
- **Botnet**: it is a network of compromised computers, known as bots or zombies, which are controlled by a central Command-and-Control (C&C) server, known as the master. These networks can be leveraged for a variety of malicious purposes, such as launching Distributed Denial-of-Service (DDOS) attacks, sending massive volumes of spam, or stealing data.
- **Fileless Malware**: it operates entirely within a system's memory. This type of malware leaves little to no trace on the hard drive, making it much harder to detect using traditional antivirus software.

The table 2.1 presents a selection of real-world malware examples across different types and years. These examples highlight various malicious software that has had significant impacts on systems, organizations, and individuals.

Year	Malware Name	Malware Type	Description
2007	Zeus	Trojan	It was primarily used to steal banking information by exploiting keyloggers. Over 74,000 FTP accounts on high-profile sites like NASA or Amazon, were compromised by June 2009.
2017	WannaCry	Ransomware	Affected over 300,000 machines across 150 countries.
2021	PhoneSpy	Spyware	With more than a thousand South Korean victims, the attackers have had access to all the data, communications, and services on their devices.
2001	Stuxnet	Worm	Mainly exploits undiscovered Windows zero-day vulnerabilities to infect Windows Systems. It was used to infect Iranian nuclear plants and a uranium enrichment plant.
1999	NTRootkit	Rootkit	First rootkit malware dedicated to Windows NT Operating System (OS)
2017	Fireball	Adware	Acts as a browser hacker, turning them into zombies to generate ad revenue and manipulate search results.
2016	Mirari	Botnet	One of the first malware to scan IoT vulnerable devices, mostly CCTV cameras, and use them to perform DDOS attacks on various sites.

Table 2.1: Real-World Malware examples by type

Modern malware frequently combines features from traditional categories to enhance its effectiveness and its ability to remain undetected. For example, Emotet is a sophisticated banking trojan that incorporates multiple functionalities, such as executing backdoor commands and spreading through phishing emails[16].

The second approach focuses on the concealment strategy employed by malware. Based on this, two main categories can be defined, as described below:

- **Non concealed Malware**: A type of malware that operates without employing any technique to hide itself. It is easy to design and can be easily detected.

- **Stealthy Malware**: A type of malware specifically designed to hide from users and detection mechanisms like static analysis and reverse engineering of the malicious code. By using concealment techniques, malware can remain undetected for a longer period in the system, stealing information or executing malicious activities without being noticed. Several concealment techniques can be employed[15][17]:

  - **Encryption/obfuscation**: The earliest and simplest technique used by malware developers. An encrypted malware consists of two main components: a decryptor and an encrypted main body. A decryptor is a small piece of code responsible for encrypting and decrypting the main body's code. The main body, generally a file, contains the malicious code, which remains encrypted until the decryptor decodes it. By encrypting its code, the malware makes the detection more challenging.
    When the infected file is executed, the decryptor retrieves the main body and executes the code. Encryption methods can vary significantly, extending from simple mechanisms to more complex techniques.

  - **Oligomorphism and polymorphism**: The main limitation of the Encryption method is that the decryptor remains constant across different exploitations, which facilitates detection through pattern recognition. Oligomorphism is an advanced form of encryption that contains a small collection of different decryptors. For each new infection, a random

decryptor is selected from the set and applied accordingly.
Polymorphism represents a sophisticated form of both oligomorphism and encryption. Polymorphic malware can generate a theoretically infinite number of distinct decryptors, continuously altering the code from one instance to another, and evading detection. They also employ obfuscation methods, like dead-code insertion or substituting instructions.

- **Metamorphism**: Metamorphic malware does not include any encrypted components, hence no decryptor is required. However, a mutation engine is employed, which produces a new malware version for each new infection. The mutation engine applies code transforming and obfuscation techniques to alter the malicious code.

The third approach categorises malicious code based on the exploited software vulnerabilities. Memory errors that enable memory corruption, are one of the most common and critical software vulnerabilities. Depending on the data structures manipulated during this memory corruption, two categories can be defined[15]:

- **Control-flow attacks**: they are common, easy to construct, and require minimal application-specific knowledge. They exploit vulnerabilities such as buffer overflows or injection flaws to modify the execution flow of a program, enabling arbitrary code execution. The attacks usually make system calls, e.g. starting a shell, with the privilege of the compromised victim process. Return-Oriented Programming (ROP) and Jump-Oriented Programming (JOP) are examples of techniques used in control-flow attacks.

- **Data-only attacks**: they are rare, and necessitate a more sophisticated knowledge of the program's semantics. However, they are more difficult to detect using traditional mitigation techniques. Data-only attacks target the program's data rather than its control flow, hence no additional code is injected

into the system. These attacks may corrupt applications for identification, configuration, and user input.

2.2 Malware Detection Overview

Malware detection is the process of identifying malicious code from benign code[13]. Malware detection can be seen as a "cat and mouse " game where researchers design new methods for detection while the goal of a malware writer (hacker) is to modify their malware to deceive the detectors. A malware detector is a component designed to apply the detection technique, like a virus scanner that uses signatures and other heuristics to identify malicious code. Malware detectors take two inputs: one input is its knowledge of the malicious behaviour, while the other is the program under inspection. Based on its knowledge, the detector can apply its detection techniques to determine whether the program is malicious or benign. Intrusion Detection Systems (IDS) and malware detectors are often used interchangeably. However, a malware detector typically serves as just one component of a comprehensive IDS[18]. Malware detection mechanisms are categorised into two main groups: Software-based mechanisms and Hardware-based mechanisms.

2.2.1 Software-based Malware detection

Software-based detection relies on specific software running in the system to detect potential malware through various approaches. A traditional example is Antivirus Software(AVS) using either signature or behaviour analysis. Despite being the most prevalent detection mechanisms, software-based solutions often incur significant computational overhead, making them unsuitable for resource-constrained systems such as IoT Edge devices, which operate under real-time and energy limitations.

Some of the most common approaches are[19]:

- **Signature-based**: One of the most popular commercial malware detection techniques used by antivirus. This approach extracts a specific sequence of bytes, the signature, from the malware executable. The signature is unique and is utilised to identify the specific malware. Once extracted, the signature is stored in a signature database. The database must be updated every time new signatures are generated, thus every time a new malware is detected. There are many different techniques to create a signature such as string scanning, top-and-tail scanning, entry-point scanning, and integrity checking. The detection process works in this way: the executable is scanned and the signature is generated based on structural properties or run-time properties; after that, the signature is compared with the signatures on the database; if a match is found, the program is marked as malicious otherwise it is considered benign. Although this approach is fast and efficient against known malware, it struggles to perform effectively against zero-day malware since the respective signature is not present in the database. In addition, malware from the same family can easily evade signature-based detection by employing obfuscation techniques[20][13].

- **Behaviour-based**: This approach observes the program behaviours using dynamic analysis, executed by monitoring tools, and determines whether the program is malware or benign. Dynamic characteristics might include processor and memory information, kernel usage (system calls), file system activities, and network communications. Behaviors are obtained by utilizing procedures like monitoring the system calls or monitoring the file changes, and they are stored in a dataset. Then, specific features from the dataset are obtained and classification is done by using ML algorithms. Although these techniques are largely immune to obfuscation, their applicability is limited by

their performance as dynamic analysis requires time, and determining malicious behaviours within the environment is an evolving challenge. Software behaviour methods can detect malware variants often missed by the signature-based approach[20][15].

- **Heuristic-based**: The idea behind heuristic-based detection is that there is no need to know the internal structure or the logic of a scanned program, but the aim is to reach the final decision with the best optimal path. To accomplish this, rules or ML techniques, like Support Vector Machine and Decision Tree, are employed. Although it has a high accuracy rate in detecting zero-day malware, it is ineffective at identifying more sophisticated malware.

- **Deep Learning**: A new approach based on Deep Learning algorithms to identify malware families. Although it is quite effective and reduces feature space drastically, it is not resistant to evasion attacks.

- **Other approaches**: **Cloud-based detection** mechanism employs several detection components hosted on cloud servers and offers security as a service. It works as follows: a user uploads any file to the cloud and receives a report indicating whether the file is malicious. However, this approach may lead to sensitive information leakages, like passwords, location, and banking details. Moreover, the detection is not performed in real-time, and an overhead is introduced by the communication between the user and the cloud infrastructure. **Mobile-based detection** focuses on malware developed for the Android platform. According to recent studies, new malicious apps for Android are introduced every 10s[20]. This mechanism uses ML algorithms on features like system calls and security-sensitive Application Programming Interfaces (APIs). **IoT-based detection** relies on log collectors to extract features from malware binaries and then utilises a lightweight Convolutional Neural Network

(CNN) for classifying their families[20].

Software-based malware techniques can also be classified according to their analysis technique. Malware analysis involves determining the malware's functionality and addressing key questions, such as how the malware operates, which machines and programs it affects, and which data is compromised or stolen. These techniques can be classified as follows[18]:

- **Static Analysis**: the malware is inspected without executing its code. Therefore, only the syntax and structural properties are considered.
- **Dynamic Analysis**: the malicious program is inspected while running its code or after its execution, leveraging run-time information.
- **Hybrid Analysis**: a combination of static and dynamic analysis.

2.2.2 Hardware-based Malware detection

Hardware-based Malware detection, or **HMD**, relies on ML classifiers based on real-time data collected from hardware components to detect malware applications[15]. The ML classifiers are trained using micro-architecture hardware events, which are monitored through **Hardware Performance Counters (HPCs)**[21]. These events are captured at runtime and represent the application behaviour. HPCs are a set of special-purpose registers embedded within the processing units, designed to improve various aspects of computing systems, such as performance and energy efficiency or for debugging[22]. In the context of malware detection, these HPCs are repurposed to enhance system security. The idea behind detecting malware using HPCs is based on the concept of phase behaviour in programs. Programs are executed in distinct phases, corresponding to patterns in architectural and microarchitectural events[23]. These phase patterns vary significantly across different programs, enabling malware identification by analysing the time-behavioural

patterns captured by selected performance counters[22].

2.3 Measurement Metrics

Since malware detection is a classification problem, the overall quality and reliability of the detectors rely on the standard classification metrics. They are essential for evaluating how well the detection system can distinguish between benign and malicious software. They can be grouped as performance metrics and efficiency metrics[15].

Performance evaluates how effectively a detection system fulfils its assigned tasks, specifically its ability to distinguish between malicious and benign software.

Efficiency measures how well a detection system performs its assigned tasks with the minimum consumption of resources.

The first tool to visualize and assess performance is the **confusion matrix**. A confusion matrix is a tabular representation of the system's predicted outcomes with the actual results.

For each prediction made by the detection system, there are four possible outcomes:

True Positives (TP): The system correctly identifies malware as malicious.

True Negatives (TN): The system correctly identifies benign software as non-malicious.

False Positives (FP): The system incorrectly classifies benign software as malware.

False Negatives (FN): The system incorrectly classifies malware as benign.

The confusion matrix for malware detection can be summarized in the following table:

	Predicted Negative	Predicted Positive
Actual Negative	TNs	FPs
Actual Positive	FNs	TPs

Table 2.2: Confusion matrix for malware detection

Key Performance metrics include:

- **Accuracy(A)**: measures the proportion of correct predictions (both true positives and true negatives) among all predictions made by the detector.

$$Accuracy(A) = \frac{TP + TN}{TP + TN + FP + FN} \tag{1}$$

- **Precision(P)**: measures the proportion of true positive predictions among all positive predictions made by the detector.

$$Precision(P) = \frac{TP}{TP + FP} \tag{2}$$

- **True Positive Rate(TPR)**: measures the proportion of actual positives correctly identified by the system, also known as recall.

$$Recall(R) = \frac{TP}{TP + FN} \tag{3}$$

- **False Positive Rate(FPR)**: measures the proportion of actual negatives incorrectly classified as positive.

$$False\ Positive\ Rate\ (FPR) = \frac{FN}{FN + TP} \tag{4}$$

- **Specificity**: measures the proportion of actual negatives correctly identified by the system, also known as True Negative Rate(TRN).

$$Specificity(S) = \frac{TN}{TN + FP} \tag{5}$$

- **F1-Score**: the harmonic mean of precision and recall.

$$F1score(F1) = 2 \cdot \frac{P \cdot R}{P + R} \tag{6}$$

- **Receiver Operating Characteristic curve(ROC**: offers a visual representation of the performance. It plots the TPR against the FPR on a 2D graph.

$$ROC = 1 - S = \frac{\mathrm{FP}}{\mathrm{FP} + \mathrm{TN}} \tag{7}$$

- **Area Under the Curve(AUC)**: represents how effectively the classifier distinguishes between malware and benign applications.

$$AUC = \int_0^1 \mathrm{R}(\mathrm{FPR})\, d\mathrm{FPR} \tag{8}$$

Key Efficiency metrics include:

- **Latency**: refers to the time interval between the collection of all features analysed by the malware detector and the final detection result. Minimizing latency is essential for real-time detection of malware that operates within brief time intervals[15].

- **Power Consumption**: indicates the energy consumed by the detector over a given period. It is mainly influenced by two factors: the hardware technology that implements the classifier and the ML algorithm[15].

- **Hardware Cost**: refers to the cost needed to build the detection system. The main parameters in assessing hardware cost include the chip area (typically measured in square millimetres) and the process technology employed (e.g., 45 nm). Furthermore, the amount of memory, along with the costs associated with the operating system and the design of the system, can also affect the

overall cost assessment[15].

2.4 Hardware Events and Performance Counters

Hardware performance counters can access detailed information regarding the processor's functional units, caches, and memory. However, the types and meanings of these counters vary across different processors as a consequence of architectural differences[24]. These counters are collected through specialized hardware monitoring components known as **Performance Monitoring Units (PMUs)**, which can track a wide variety of hardware events generally categorised into five categories[25]:

- **program characterization events**, analyse the attributes of a program independent of the underlying processor architecture. Common examples of these events include the number and types of instructions completed by the program, such as loads, stores, floating-point operations, and branches.

- **memory accesses events**, often the largest category, analyses the performance of the processor's memory hierarchy, helping to evaluate memory latency, bandwidth usage, and cache performance.

- **pipeline stalls events**, reflect how effectively a program's instructions move through the processor's pipeline.

- **branch prediction events**, related to the performance of the processor's branch prediction hardware.

- **resource utilisation events**, enable to monitor how effectively the processor utilises various internal resources, such as the number of cycles spent using a floating-point divider.

Modern processors have hundreds of events that can be monitored. However, to minimise the cost and hardware complexity, they have only a limited number of HPCs (e.g., 2 to 8 in high-end processors). Each HPC can monitor only one hardware event at a time to maintain accuracy. As a result, collecting various performance events often requires running the application many times.
An HPC consists of 2 components: a performance event detector and an associated counter. The event detector can be configured to monitor any one of several performance events, typically using a bitmask to specify the event. The associated counter increments by one each time the corresponding event occurs, or when the event's value exceeds a predefined threshold. These updated values are continuously stored in an associated register, with the final count being available at the end of execution[15][25].

2.5 Limitations of Hardware-based malware detection

Compared to software-based detection methods, HMD offers several remarkable strengths[15][26][27]:

- **Run-time detection**: The analysis is performed based on hardware data collected in real-time, allowing for fast malware detection, often within milliseconds.
- **Low computational overhead**: hardware-based detection methods are implemented in microprocessor hardware with significantly low overhead as compared to the software-based methods. The detection is very fast (a few clock cycles).
- **Resilience to Obfuscation**: Traditional Software-based detection technique

relies on static analysis to detect malware by identifying suspicious patterns. However, malware authors can easily create numerous variants of the same malicious code or deploy stealthy malware to evade detection. In contrast, dynamic analysis via hardware-based approaches enables the detection of code variants and unknown malware more effectively.

- **Tamper Resistance**: HMD relies on secure hardware components rather than vulnerable software, reducing the risk of malware exploiting bugs or vulnerabilities to bypass detection mechanisms.

- **Cost efficiency**: Since HPCs are typically already integrated into most modern processors, HMD can leverage existing hardware, minimizing the need for additional hardware.

When HPCs are used for security purposes, several factors may cause discrepancies in event measurements, reducing the accuracy[27][15]:

- **External sources**: The runtime environment may differ across executions. Elements such as OS activity, multitasking scheduling, memory layout, and multi-processor interactions can vary between runs, resulting in differences. Likewise, shifts in the micro-architectural state can alter event counts.

- **Non-determinism**: Many system-level events are inherently unpredictable and largely influenced by OS behavior and the activity of other applications running on the system. Hardware interrupts and page faults are common examples of non-deterministic events.

- **Overcounting**: Certain processors, such as older Intel Pentium D models, may occasionally count multiple times specific hardware events, leading to inaccuracies in performance measurements.

- **Variations in tool implementations**: Many tools (such as Perf in Linux or Intel VTune) are employed for measuring HPC events, but they operate in different ways. Despite running identical programs under controlled conditions, different tools may yield differing results. This variability may be attributed to differences in the techniques used for reading the counters, the methods employed for acquiring the measurements, or the levels at which the measurements are taken.

The effectiveness of HMD is influenced by both the type of ML employed and the number and type of HPC events utilised. Moreover, malware detectors may be designed for specific devices with specific characteristics defined by the architecture and the manufacturer. For instance, processors may track different numbers of events at a time, and discrepancies in instruction counting methods are possible.

Chapter 3

Simulations Environment

This chapter will present the simulation workflow employed in this thesis, starting with an overview of the gem5 simulator, which was utilised to run the applications under investigation. The chapter details the configuration and simulation setup necessary to ensure accurate and effective use of the simulator's HPC. It also presents the set of applications used to collect the HPC data provided to the ML classifiers, explaining the rationale behind their selection.

3.1 Introduction to Gem5 Simulator and RISC-V

Gem5 provides a robust and flexible virtual environment for simulating RISC-V processors[6][28]. It is a highly customisable, modular, and widely used simulator for computer architecture research. Gem5 enables detailed and flexible simulations of complex hardware systems, making it particularly suited for HPC data gathering. It offers a comprehensive selection of CPU, RAM, and device models, along with support for various ISAs, including x86, ARM, and RISC-V.

This thesis focuses on HMD applied to systems based on a specific ISA: **RISC-V**. RISC-V, an open-source ISA, has gained significant attention in the research

community for its flexibility, extensibility, and growing ecosystem. Its open nature contrasts with proprietary alternatives, such as ARM or x86, offering researchers and developers the ability to customise it by adding or removing features. Its modular architecture, with variants in address space sizes, makes it suitable for a range of applications, from lightweight edge devices to high-performance servers, thereby applicable to diverse fields including embedded systems and cloud computing. The evaluation of application trustworthiness involves more than just basic performance metrics like execution time or clock cycles. It requires a comprehensive assessment of multiple factors, including pipeline stages, execution ports, associated latencies, reorder buffers, load/store queues, and cache organization. To gain insights into these aspects, capturing and analysing detailed performance metrics through HPC becomes essential.

3.2 Gem5 Architecture

Gem5 is characterized by several key features that enhance its versatility and flexibility[6]:

- **Pervasive object orientation**: Gem5 features an object-oriented design, where all major simulation components are implemented as SimObjects. A SimObject serves as an abstraction of real hardware components such as CPUs, caches, interconnects, and devices. Each SimObject is defined by a Python class for configuration and a C++ class for managing its state and performance-critical simulation behaviour.

- **Python integration**: Python in gem5 allows users to script and configure simulations. The common Python base class ensures consistent mechanisms for instantiation, naming, and parameter setting of SimObjects. Consequently, this allows for a highly dynamic and configurable simulated system.

- **Domain-specific languages**: To increase flexibility, Gem5 supports domain-specific languages for two key areas: ISA and cache coherence protocols. This allows full support of ISA details and facilitates the implementation of more complex components.

- **Standard interfaces**: Standard interfaces ensure that various components can communicate regardless of their specific implementations. This is vital for simulating diverse architectures and configurations.

When simulating a system with gem5, the key options to consider include:

- **CPU model**: Gem5 offers four distinct CPU models, each with different trade-offs between speed and accuracy. AtomicSimpleCPU is a minimal, single-Instruction Per Cycle (IPC) model that uses atomic memory accesses for fast functional simulation. TimingSimpleCPU is similar but incorporates timing memory accesses for more accurate performance simulation. InOrderCPU is a pipelined CPU that processes instructions in a strict, sequential order, and O3CPU is a pipelined, out-of-order model that allows for more advanced instruction reordering. The O3 and InOrder models use an "execute-in-execute" approach, meaning that the instructions are executed during the pipeline's execution stage. This contrasts with many simulators that execute instructions either at the start or the end of the pipeline, resulting in greater timing accuracy for gem5.

- **System Mode**: Each CPU model can function in one of two modes. In System-call Emulation (SE) mode, Gem5 avoids simulating devices and the OS by emulating key system-level services. In this mode, the simulator runs a single static application and the system calls are emulated or forwarded to the host OS. On the other hand, Full-System (FS) mode simulates a complete system, executing both user-level and kernel-level instructions while modelling

the OS and hardware devices. Figures 3.2 and 3.1 illustrate the different system modes.

- **Memory System**: Two memory system models are available: Classic and Ruby. The Classic model delivers a fast and easily configurable memory system for simpler simulations. In contrast, the Ruby model provides a more flexible infrastructure capable of accurately simulating a wide range of cache-coherent memory systems with greater detail. The memory system components are implemented using a MemObject and communicate with other components (SimObjects) via a master/slave interface. Typically, each master port is connected to an interconnect component, like a bus or bridge. Once connected, the system can exchange messages, such as a Request or a Packet.

An additional key feature of gem5 is its ability to create **checkpoints**. A checkpoint is a simulation snapshot, allowing users to resume the system from that exact state later. This is particularly advantageous in scenarios where simulations may take an extended period to boot the OS or execute applications. In this thesis the checkpoints are used to resume the system, switching the CPU model from a simpler CPU to the O3CPU, which is essential for gathering the HPC data.

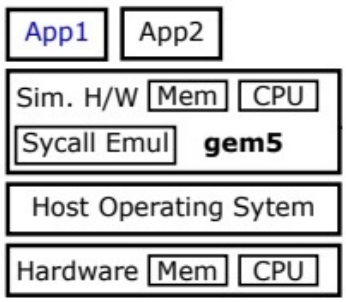

Figure 3.1: Gem5 System-Call mode. Figure from [29]

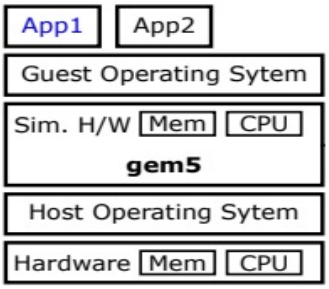

Figure 3.2: Gem5 Full System mode. Figure from [29]

3.2.1 HPC System Configuration

In a typical Linux environment, the PMU is managed by **Hardware Performance Monitoring tools (HPM)**, such as Perf, which provide detailed information by accessing HPCs directly. However, in the gem5 simulator, the full set of HPCs available on real hardware is not implemented. As a result, the Linux Perf tool is not applicable for simulations in gem5. Instead, HPCs are integrated within the gem5 simulation framework itself. These counters can only be accessed when using the DerivO3CPU, a specific implementation of the O3CPU model.
The HPCs, ranging from *hpmcounter3* to *hpmcounter31*, can be accessed via the thread context of the simulation process. At a lower level, each counter tracks specific events within the processor and is stored and accessible through a set of **Control Status Registers (CSRs)**[22]. Figure 3.3 presents the implementations of the functions that enable safe manipulation of HPC values within the simulation environment.

```
1  /* Function to read the HPC */
2  #define read_csr_safe(reg) ({
3          register long __tmp asm("a0");
4          asm volatile ("csrr %0, " #reg : "=r"(__tmp));
5          __tmp;
6          })
7
8  /* Function to write a value to an HPC event register */
9  #define write_csr_safe(reg, val)
10     do {
11          register long __val asm("a0") = (val);
12          asm volatile ("csrw " #reg ", %0" : : "r"(__val) : "memory")
       ;
13     } while(0)
```

Figure 3.3: Functions to read and write HPC values in gem5

The `write_csr_safe` function writes a specified event to the corresponding CSR

using the csrw instruction, enabling tracking of that event. Conversely, the `read_csr_safe` function reads the value from the specified CSR using the csrr instruction, allowing retrieval of the performance counter associated with the tracked event. Together, these functions enable the interaction with HPCs within the gem5 simulation framework.

3.3 Experimental Workflow

In this thesis, the Gem5 simulator is employed to conduct detailed simulations, enabling the collection of HPC data specific to RISC-V systems. The overall experimental workflow follows several key steps:

- **Configure the Gem5 Full System environment**: Set up the gem5 simulator in full-system mode to accurately model hardware and OS behaviours.
- **Select applications benchmark**: Choose the appropriate benchmark applications and malware.
- **Configure the Gem5 Simulation and start the simulation**: Define the necessary simulation parameters and run the simulations.
- **Data Collection**: Gather HPC data during the simulation which will be used as input for ML classifiers.
- **Data Analysis**: Run the ML classifiers using the collected HPC data to evaluate the malware detection accuracy.

3.4 Gem5 Configuration

In today's cloud environments, the Linux OS is widely adopted for its open-source nature and the flexibility it offers to developers. This thesis aims to replicate

similar environments in simulation, utilising Gem5 Full System mode for RISC-V to closely model cloud-based systems.

3.4.1 Gem5 Full System

Gem5's Full System mode models bare-metal hardware, meaning all system components must be defined from scratch. To emulate a RISC-V platform, it is necessary to specify both the hardware features and an OS that supports RISC-V. This process involves several key components, all of which must be compatible with the RV64GC variant of the RISC-V ISA:

Toolchain: The **riscv64-linux-gnu** toolchain enables cross-compiling software on a host machine (x86) to run on a RISC-V 64-bit target machine running Linux. It is crucial for cross-compiling both application and malware binaries for RISC-V.

Static Linux Kernel Binary: This binary is sourced directly from the official Linux repository and statically cross-compiled using the RISC-V toolchain.

Bootloader: The **RISCV Proxy Kernel (pk)** serves as an application execution environment. It incorporates the **Berkeley Bootloader (BBL)** source code, which takes the Linux kernel as a payload and generates a bootable binary, enabling the system to boot successfully within the gem5 simulation environment.

Disk Image: The **BusyBox image** provides a lightweight set of Unix utilities commonly used in embedded systems. BusyBox offers a collection of various standard Linux commands into a single executable, making it ideal for environments with limited resources, such as simulation setups. In this setup, the BusyBox image serves as the root filesystem in the gem5 simulation, providing basic utilities and a minimal environment to support the execution of applications and the kernel.

3.4.2 Gem5 Configuration Script

Once the resources intended for use in GEM5 Full System simulations have been defined, the next step is to configure the GEM5 Simulator. This configuration is carried out using a designated file known as *RunSystem*.

The Listing 3.1 provides a simplified version of the script used for the final simulations. A complete system setup, including all necessary device initialization, can be found in Appendix A.

```
class RiscvSystem(System):
    def __init__(self, bbl, disk, cpu_type, num_cpus, script):
        super(RiscvSystem, self).__init__()
        # Initialize clock and memory
        self.setupClock()
        self.setupMemory()
        # Create CPUs and set up platform
        self.createCPU(cpu_type, num_cpus)
        self.platform = HiFive()
        self.initDevices(disk)
        # Create cache hierarchy and memory controller
        self.createCacheHierarchy()
        self.createMemoryControllerDDR4()
        # Set the workload object file (bootloader)
        self.workload.object_file = bbl
        boot_options = [
            "console=ttyS0",
            "root=/dev/vda",
            "rw"
        ]
        self.workload.command_line = " ".join(boot_options)

    def createCPU(self, cpu_type, num_cpus):
        # Create CPU instances based on the specified type
```

```
25          if cpu_type == "Atomic":
26              self.cpu = [AtomicSimpleCPU(cpu_id=i) for i in range(
    num_cpus)]
27          elif cpu_type == "DerivO3":
28              self.cpu = [RiscvO3CPU(cpu_id=i) for i in range(num_cpus)
    ]
29          for cpu in self.cpu:
30              cpu.createThreads()
31
32      def initDevices(self, disk):
33          # Initialize devices and I/O components
34          self.initVirtIO(disk)
35          self.initBridge()
36          self.setupInterrupts()
37
38      def setupClock(self):
39          # Set up the clock domain and voltage domain
40          self.clk_domain = SrcClockDomain(clock='3GHz', voltage_domain
    =VoltageDomain())
41
42      def setupMemory(self):
43          # Create memory ranges and main memory bus
44          self.mem_ranges = [AddrRange(start=0x80000000, size='2GB')]
45          self.membus = SystemXBar()
46          self.membus.badaddr_responder = BadAddr()
```

Listing 3.1: RunSystem snippet

This script is used to create a RiscvSystem object, instantiating all the necessary components. Several key objects are instantiated:

- **CPU**: The system's CPUs can be configured as either Atomic or DerivO3, with the capability to instantiate multiple CPUs.

- **Memory Controller**: This component manages the connection between the processor and various types of memory.
- **Disk Image**: A Copy-On-Write (COW) image of the specified disk is created.
- **Bus**: This component enables data transfer among the CPU, memory, and peripheral devices.
- **Interrupt Controllers**: handle interrupt requests.
- **Workload**: This object configures the simulation environment for executing the RISC-V bbl and, consequently, the Linux kernel. It ensures that essential parameters, including the device tree, command-line options, and object files, are correctly configured.

Several components make up the overall architecture of the system. The organization of these components follows the schematic shown in Figure 3.4, where the interactions between the CPU, caches, buses, and memory controller are illustrated.

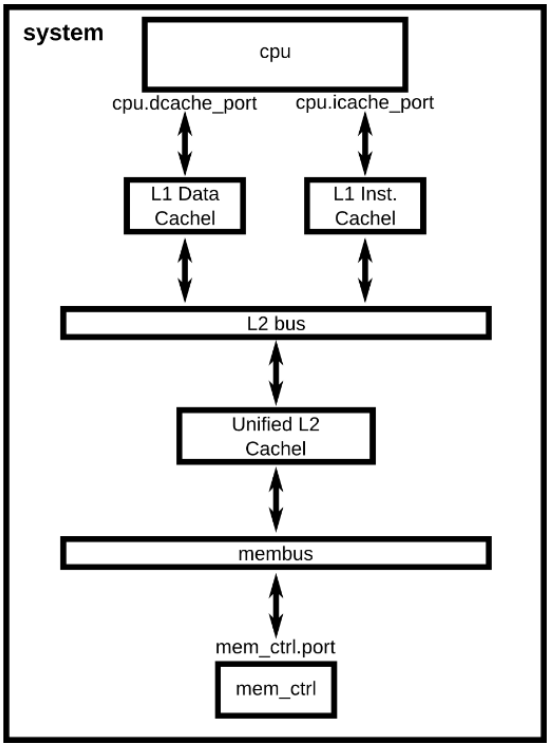

Figure 3.4: Architecture Hardware Configuration. Figure from [30]

3.5 Gem5 Simulation

The Configuration Script is responsible for creating the RiscvSystem, which defines all the low-level characteristics of the simulated system. On the other hand, the Simulation Script instantiates this object, allowing the system to be simulated. This simulation is executed using a designated file called *RunSimulation*.

3.5.1 Simulation Script

The Simulation Script specifies important parameters for the simulation, such as the bbl, the disk image, and the number and type of CPUs to be used. Additionally, it defines an **rcS** file, which acts as the primary means of passing input to the simulator. The rcS file can be accessed within the simulation and is used to

provide applications for execution along with their corresponding input parameters. This file, described in Listing 3.2, is essential for running specific workloads or benchmarks on the system.

```
default_kernel = './resources/bbl_5.18'
default_disk = './resources/riscv_disk.img'
default_cpu_type = 'atomic'
default_num_cpus = 2
checkpoint_dir = "./checkpoint"
script = "./configs/example/riscv/rcS"

def run():
    print("Running the simulation")
    cptdir = checkpoint_dir
    print("Checkpoint directory: %s" % cptdir)

    while True:
        event = m5.simulate()
        exit_msg = event.getCause()
        if exit_msg == "checkpoint":
            print("Dropping checkpoint at tick %d" % m5.curTick())
            cpt_dir = os.path.join(cptdir, "cpt.%d" % m5.curTick())
            m5.checkpoint(os.path.join(cpt_dir))
            print("Checkpoint done.")
        else:
            print(exit_msg, " @ ", m5.curTick())
            print("Simulation ended")
            break
    sys.exit(event.getCode())

if __name__ == "__m5_main__":
    # set up the root SimObject and start the simulation
    system = RiscvSystem(default_bbl,default_disk, default_cpu_type,
```

```
30      default_num_cpus, script)
31      root = Root(full_system=True, system=system)
32      globalStart = time.time()
33
34      # Check if restoration is needed
35      restore_checkpoint = None
36      if restore_checkpoint is not None:
37          print("Restoring from checkpoint")
38          cpt_dir = os.path.join(checkpoint_dir, "cpt.%d" %
    restore_checkpoint)
39          m5.instantiate(cpt_dir)
40      else:
41          print("No restoration, starting fresh")
42          m5.instantiate()
43      # Run the simulation
44      run()
```

Listing 3.2: RunSimulation snippet

This script is a simplified version of the one used for the final simulations. In the full simulation script, arguments can be passed to override the default values defined here. Running this script initiates the simulation, with checkpoints saved in a designated .cpt folder.

3.6 Applications Benchmarks

After defining the simulation and configuration parameters, the next step is to initiate the simulations. In addition to this, the thesis also focuses on curating a balanced selection of benign and malicious applications to be used as input for the machine learning classifiers. A well-constructed dataset, composed of both benign and malicious applications, is crucial for training accurate and reliable ML models.

3.6.1 Benign Apps

For the benign application samples, a diverse set of programs is utilised, including Linux system utilities, text editors, benchmarking suites like MiBench, as well as tools for file compression, media processing, network scanning, and backup management. The table 3.1 provides a list of all eight applications used in this thesis.

Application	Description
SUSAN	An image recognition application designed for detecting corners and edges in Magnetic Resonance Images (MRI) of the brain. Typically used in real-world, vision-based quality assurance systems[31].
GSM	A voice encoding/decoding application that utilises Time- and Frequency-Division Multiple Access (TDMA/FDMA) for processing data streams[31].
FFMPEG	A media converter application capable of reading various input formats, including live devices, and transcoding them into multiple output formats. It allows users to specify input and output files via command-line options, supporting complex stream selections while emphasising the importance of file order and stream indexing[32].
RESTIC	A backup application that supports backing up files from Linux, BSD, macOS, and Windows to various storage options, including self-hosted and online services. It also ensures security through cryptography[33].

Application	Description
AG	A fast command-line tool for searching through code, designed to enhance search speed and efficiency compared to similar tools. It is particularly useful for developers looking to find text patterns within large codebases on Linux systems quickly[34].
UNRAR	A command-line utility used to extract files from RAR archives. It allows users to easily access and manage compressed content[35].
NMAP	An utility for network exploration and security auditing, widely used by system and network administrators as well as hackers. It utilises raw IP packets to identify available hosts on a network, determine the services they offer, detect OSssand versions, and assess firewall characteristics[36].
NPING	A tool for network packet generation, response analysis, and response time measurement. In addition to these features, it can also be used for DDOS attacks and route tracing[37].

Table 3.1: List of Applications and Their Descriptions

3.6.2 Malicious Apps

The selected collection of malware applications is sourced from various datasets, which are hosted on GitHub repositories. These datasets typically consist of numerous malware samples that are categorised by type, intended effects, and the systems they target. However, a significant portion of the available malware targets

Windows systems, making them unsuitable for the Linux-based system being developed in this thesis. As such, only Linux-based malware has been considered. Additionally, many datasets contain only precompiled binaries, which are not ideal as they often depend on system libraries or functionalities that are not present in the disk images used. As a result, only malware with accessible source code was selected, with a focus on C/C++ code, since other languages like Python are incompatible with the system being created. The table 3.2 lists the six malware applications used in this thesis.

Malware	**Description**
GONNACRY	A ransomware that targets Linux systems. It encrypts files using the AES-256 algorithm and appends the ".GonnaCry" extension to them. Notably, it operates without a Command-and-Control(C&C) server, displaying a ransom note directly on the infected system for payment instructions[38].
RANDOMWARE	A ransomware that encrypts a victim's files using XOR with a secret key and data and demands payment to restore access[39].

Malware	Description
LINUX-PARASITE	A Linux-based virus that opens a backdoor to provide unauthorized remote access. It operates as part of a (C&C)system, where a central server can control multiple infected clients. This type of malware typically establishes deep control over the OS, enabling attackers to perform various malicious actions, such as stealing data or executing commands remotely[40].
POP3 TROJAN	A Remote Access Trojan(RAT) specifically designed to exploit the POP3 email protocol. Upon execution, the malicious binary opens a backdoor on the victim's machine, enabling the attacker to issue specific commands remotely or execute commands embedded within the binary itself. This functionality can lead to the theft of sensitive information, such as email credentials, and may result in broader system compromise[40].
SATAN-BOMB	A botnet malware designed to execute DDOS attacks against the target system by utilising the fork function, which allows the malware to generate multiple processes to flood the victim with requests[39].

Malware	Description
TRIGEMINI	A sophisticated botnet malware capable of launching various types of DDOS attacks. It supports multiple attack vectors such as TCP (SYN, FIN, ACK), UDP, and ICMP, making it versatile in overwhelming network defences[41].

Table 3.2: List of Malware Applications and Their Descriptions

3.6.3 Application and Malware Execution

Once the applications and malware are selected, the Gem5 simulations can be run. The applications are automatically executed within the simulated environment using the *RunAppMalware Script* 3.3.
This script is crucial to the entire simulation process, as it manages the automated execution and monitoring of applications, serving as the core component in collecting the necessary data for analysis. It is executed once the system boot process is complete and operates in distinct parts:

- **Event Configuration**: In this initial phase, the script specifies which events to monitor by writing them to the relevant HPC counters.
- **Counter Initialization**: The second part involves reading the initial values of the HPC counters.
- **Application Execution**: The third part executes the application, which can be either a benign application or a combination of benign and malicious software.
- **Post-Execution Measurement**: After executing the application, the script

re-reads the HPC counter values and calculates the difference from the initial readings. This result represents the HPC measurement obtained during the application's execution.

- **Output File Generation**: Finally, the results are saved to a file, which can be exported from the simulation environment using the m5 write command.

```
/* Function Definitions */
void run_app() {
    /* Initialize HPM event tracking */
    write_counters();
    /* Capture initial counter values */
    read_counters(0);

    /* Execute the benign application from input */
    system(command);  // Example: "./restic init --repo restic-copy"
    /* Execute the malware application or remove it if only the
    application is executed */
    system("./pop3 -b -p 90");  // Example: POP3 malware

    /* Capture final counter values */
    read_counters(1);
    /* Calculate and save the difference in counter values */
    compute_results();
    /* Save the simulation result to the host system*/
    save_results();
}

int main(int argc, char **argv) {
    /* Create the checkpoint */
    system("m5 checkpoint");
    /* Read input file with application commands */
    system("m5 readfile > input.txt");
```

```
26      /* Execute the application (with or without malware) */
27      run_app();
28      m5_exit(0);
29      return 0;
30  }
```

Listing 3.3: RunAppMalware snippet

The snippet in the Listing 3.3 is a simplified version of the script used in the final simulations. Each simulation follows this structured procedure, generating a file that contains all relevant HPC data. These files will subsequently be used to construct the training and test datasets.

3.6.4 Simulation Host Environment

All simulations are conducted within a Singularity container running Ubuntu 22.04.3 LTS. The Gem5 simulations are executed on a system equipped with an AMD Ryzen 9 7950X CPU, with 3.0 GHz frequency, and 60 GB of RAM. The Listing 3.4 is the Bash script used to launch multiple simulations and define checkpoints.

```
1  #!/bin/bash
2
3  # Check if the number of arguments is correct
4  if [ "$#" -ne 2 ]; then
5      echo "Usage: $0 <Input file within the simulation> <Output folder
       >"
6      exit 1
7  fi
8
9  # Path to the rcS file where the input file will be written
10 SCRIPT_PATH="./configs/example/riscv/rcS"
11
12
```

```
13 # Function to modify the rcS script with the input file content
14 modify_script() {
15     echo "Writing input file content to rcS script..."
16     echo "$1" > "$SCRIPT_PATH"   # Overwrite rcS script with the
       input file
17     content
18 }
19
20 # Function to run the gem5 simulation
21 # $1: Output folder name
22 run_gem5() {
23     echo "Starting gem5 simulation..."
24     ./build/RISCV/gem5.opt -d stats_folder/"$1" configs/example/riscv
       /run_script.py \
25     ./resources/bbl ./resources/disk_image.img \
26     --restore=checkpoint --cpu_type=DerivO3
27 }
28
29 # Modify the rcS script with the input file content
30 modify_script "$1"
31
32 # Run the gem5 simulation in the background
33 pid_list=()  # Initialize array to store process IDs
34
35 run_gem5 "$2" &  # Run gem5 in the background
36
37 pid=$!  # Capture process ID of the background job
38
39 pid_list+=($pid)  # Add PID to the list
40
41 # Wait for all background processes to finish
42 echo "Waiting for all background processes to complete..."
43
```

```
44 for p in "${pid_list[@]}"
45 do
46     wait $p
47 done
```

Listing 3.4: LaunchScript snippet

Initially, the Bash script is executed using the AtomicCPU for faster simulation. After saving the checkpoint, the simulations are re-run with the DerivO3 CPU model to gather the necessary HPC data for analysis.

Chapter 4

Data Analysis

Once all the simulations are completed, the remaining key components of the evaluated framework 1.1 are data collection and preprocessing, and malware detection process.

Data collection and Preprocessing is the process of gathering HPCs data from both benign and malicious applications, while preprocessing prepares this data for model training and testing.

The **Malware Detection Process** is the core of this thesis, as it allows for the distinction between benign and malicious applications. In the evaluated HMD framework, the malware detection process relies on **Anomaly Detection** to identify patterns or behaviours that deviate from normal system activity. Unlike conventional solutions, anomaly detection uses only benign applications as the training set, which is then employed to train the ML classifiers.

In this chapter, each of these components will be explored in detail. The methods used to analyse the data to detect malware effectively will be highlighted, along with the presentation of the final results.

4.1 Data Collection and Preprocessing

After establishing the simulation environment and selecting both benign and malicious applications for analysis, the subsequent phase focuses on gathering, processing, and analysing the data to detect malicious behaviour. To achieve this, two types of executions are conducted:

- **Benign application mode**: tests the normal behaviour of the system under benign conditions.
- **Benign application with malware mode**: each benign application is executed followed by the execution of each malware, simulating potential attack scenarios.

In the first mode, for each benign application, a variety of inputs are applied to capture a comprehensive range of behaviours, including normal execution as well as scenarios involving errors or malfunctions. These inputs are designed to replicate diverse real-world conditions. The application is then tested with these inputs, where each test run constitutes a simulation. After each simulation, the HPC values, for that specific simulation, are saved in a file with the following format:
{application_name + input_parameter}
In the second mode, each benign application is executed with the same input parameters followed by the execution of malware. This process is repeated for all 6 malware. In this case, the format is:
{application_name_malware_name + input_parameter}
The captured data represents the system's behaviour under both benign and malicious conditions, serving as the foundation for the detection mechanism.
Data Collection and Preprocessing comprises two steps:

- **Feature extraction**: This step consists of capturing and storing the selected HPC values.

- **Feature selection**: also called feature reduction, this process involves identifying and selecting the most relevant attributes or characteristics from data in input to improve classification accuracy while minimizing computational complexity.

4.1.1 Feature Extraction

Feature Extraction involves extracting and saving the HPC data, which serve as the key features analysed by ML classifiers. HPCs can be extracted through two main methods: time-based and event-based extraction[15][25].

- **Time-based extraction**: collects HPC data at fixed intervals, either after a certain interval of time or a set number of processor cycles.
- **Event-based extraction**: collects data after a specific number of events have occurred or after a certain number of instructions have been executed.

There are several ways to extract HPC data: using specific kernel tools such as *Perf*; incorporating specialized libraries like *PAPI* directly into the source code; utilising proprietary kernel modules or drivers; or employing a simulator to emulate the processor and collect data. In this thesis, the last option is employed. The Gem5 simulator is used, along with the RunAppMalware in Listing 3.3, to extract and store HPC values. For the extraction process, a time-based approach has been chosen, specifically a **single analysis per detection** method[42]. This method differs from typical periodic sampling approaches, where data are collected at intervals throughout execution. Instead, HPC values are recorded only at the end of the application execution. This approach is used for both benign applications as well as for instances where benign and malware applications are executed together. Time-based sampling does not have strict rules for determining the exact intervals at which data should be collected. In the context of HMD, sampling intervals often

range from milliseconds to seconds. Lower sampling rates are less computationally demanding, producing fewer data points but potentially offering an incomplete representation of program behaviour. Higher sampling rates generate more detailed data, but at the cost of frequent system interruptions and increased computational overhead. The single analysis method is chosen because Gem5 simulates execution cycle-by-cycle, making it inherently slow. Implementing high-frequency sampling (e.g., every second or millisecond) would significantly slow down the simulation and impose an excessive computational overhead. Thus, the chosen approach balances the need for data collection with the practical constraints of the simulator's performance. Table 4.1 provides an example of feature extraction, illustrating the output file obtained from a simulation.

Metric	Value
Elapsed cycles	26,538,207
Elapsed time (ns)	883,721
Elapsed instructions	33,880,677
L1 instruction cache misses	**29,380**
L1 data cache misses	**59,412**
ITLB Misses	**664**
DTLB Misses	**3,674**
NumOfExceptions	**4,315**
ERET instructions	**446**
Pipeline Nukes	**489**
Branch Misfetches	**3,985**
Branch Mispredictions	**33,733**
Load instructions	**7,334,500**
Store instructions	**4,250,425**
Control flow instructions	**5,941,716**
L2 cache hits	**27,041**
L2 cache misses	**200,932**
Number of instructions committed (Count)	**33,880,677**
Data Memory Accesses	**12,041,262**
L1 Instruction cache access	**5,828,902**
L2 cache access - Instructions	0
L2 cache write-backs	**56,610**
L2 cache access - Data	0
L2 cache access - Page Walker(Data)	0
L2 cache access - Page Walker(Instructions)	0
L1 data cache accesses	**12,041,262**
D-TLB miss - Read	**2,694**
D-TLB miss - Write	**980**
L1 data cache write-backs	**7,287**
Branch Predictions	**7,753,603**
Conditional Branch Predictions	**3,722,222**

Table 4.1: Simulation Statistics example

4.1.2 Feature Selection

Feature selection (FS) involves reducing the initial set of features, in this case, the HPC values, to a subset of relevant inputs. This process minimizes the impact of noise and irrelevant variables, leading to more efficient analysis and better predictive performance. When working with ML algorithms, a large dataset

with many features can result in high-dimensional data processing, introducing significant computational overhead and complexity. This situation leads to the Curse of Dimensionality, where the performance of ML classifiers decreases as the number of features increases. Furthermore, including irrelevant features can further reduce the accuracy of the classifier. For these reasons, FS is essential. It also enhances data visualization and comprehension, decreases measurement and storage requirements, and minimizes training and execution times[43]. There are several ways to classify the FS techniques. In this context, they can be categorised into the following groups:

- **Wrapper Methods**: These methods use algorithms to search for an optimal subset of features by evaluating different combinations with a learning algorithm. This process is repeated until certain stopping criteria are satisfied, such as achieving the highest quality of FS or obtaining a predetermined number of features[44].

- **Filter Methods**: These methods do not rely on a specific learning algorithm, making them quicker and simpler compared to wrapper methods. They rank features according to specific evaluation criteria, assessing their relevancy through either univariate methods (ranking features individually) or multivariate methods (ranking multiple features simultaneously). Features that fall below a certain threshold are then removed from the dataset[44]. Several algorithms are based on filter methods, including **Principal Component Analysis (PCA)**, which is the method used in this thesis.

- **Embedded Methods**: These methods represent a trade-off between filter and wrapper methods, as they integrate FS into the training process[44].

- **Hybrid Methods**: These methods can be viewed as combining various FS algorithms (e.g., wrapper, filter, and embedded). Their primary goal is to

address the instability of many existing FS techniques[44].

To ensure effective FS a thorough understanding and description of the relevant hardware events is essential. All hardware events in the RISC-V HPM used for simulations are implemented through the *Zicntr* and *Zihpm* RISC-V extensions. These extensions provide pseudo-instructions, detailed in Figure 3.3, that allow access to HPM-related CSR registers, which manage the HPCs. Although approximately 60 hardware events are available for tracking, only 31 HPCs are implemented and available for practical use in Gem5[45]. The selection of events is performed manually by writing the corresponding code within the specific counter, as illustrated by the Listing 3.3. The hardware events monitored during simulations can be categorised based on the specific aspects of system performance they track. The tables from 4.2 to 4.9 provide an overview of the key hardware events, grouped by type.

Hardware events related to Performance:

Event	**Description**
Elapsed Cycles	Total number of cycles taken to execute the program.
Elapsed Time (ns)	Total execution time measured in nanoseconds.
Elapsed Instructions	Total number of instructions executed.

Table 4.2: Performance Metrics

Hardware events related to Instructions:

Event	Description
Number of Load Instructions	Count of load instructions executed.
Number of Store Instructions	Count of store instructions executed.
Number of Control Flow Instructions	Total control flow instructions executed.

Table 4.3: Instruction Metrics

Hardware events related to Cache:

Event	Description
L1 Instruction Cache Misses	Count of missed accesses in L1 instruction cache.
L1 Data Cache Misses	Count of missed accesses in L1 data cache.
L2 Cache Hits	Number of successful accesses in L2 cache.
L2 Cache Misses	Count of missed accesses in L2 cache.
L1 Data Cache Accesses	Total number of accesses to the L1 data cache.
L2 Cache Write-Backs	Number of write-backs to L2 cache.

Table 4.4: Cache Metrics

Hardware events related to Pipeline:

Event	Description
Number of Pipeline Nukes	Pipeline flushes resulting from mispredictions or exceptions.

Table 4.5: Pipeline Metrics

Hardware events related to TLB:

Event	Description
ITLB Misses	Count of instruction TLB misses.
DTLB Misses	Count of data TLB misses.
D-TLB Miss - Read	Number of read misses in the data TLB.
D-TLB Miss - Write	Number of write misses in the data TLB.

Table 4.6: TLB Metrics

Hardware events related to Branch Prediction:

Event	Description
Number of Branch Mispredictions	Count of incorrect branch predictions.
Number of Conditional Branches Predicted	Number of conditional branches predicted.

Table 4.7: Branch Prediction Metrics

Hardware events related to Exception and Error:

Event	Description
NumOfExceptions	Total number of exceptions encountered during execution.
ERET Instructions	Count of ERET (exception return) instructions executed.

Table 4.8: Exception and Error Metrics

Hardware events related to Memory Access:

Event	Description
Data Memory Access	Total number of data memory access operations.

Table 4.9: Memory Access Metrics

4.1.3 Applied Feature Selection

In this thesis, FS is initially conducted manually, as some HPCs are unsuitable for analysis. Specifically, events that always remain in zero, such as '**L2 cache access - Instructions**', '**L2 cache access - Data**', '**L2 cache access - Page Walker (Data)**', and '**L2 cache access - Page Walker (Instructions)**' are excluded. Furthermore, the first three hardware events '**Elapsed Cycles**', '**Elapsed time (ns)**', and '**Elapsed Instructions**' are excluded, as they typically capture general system performance and do not provide detailed insights into the program's behaviour or distinguish between benign and malicious activities[21][46].
The final set of features offered to the ML classifiers is reduced to 24, down from the initial 31, after excluding irrelevant and redundant events. Referring to Table 4.1, the metrics highlighted in bold represent the 24 features selected for further analysis following the FS process.
At this point, **PCA** was employed to rank the 24 features. PCA is a multivariate statistical technique used to reduce the dimensionality of datasets to a smaller set of features while enhancing the clarity of the results and preserving as much information as possible[15]. It accomplishes this by generating orthogonal (uncorrelated) components, called the **principal components**, that sequentially capture the maximum variance in the data.
PCA helps mitigate the curse of dimensionality, and the reduction of noise in the data allows better visualisation of the data set. To perform PCA, various tools can be employed; in this case, the Scikit-learn library in Python is utilised.

4.1.4 Training and Test Procedures

The final step in Data Collection and Preprocessing is the generation of the Training and Testing datasets. In this phase, raw HPC data from simulations is processed

and organized into a structured format for use in the subsequent analysis. Figure 4.1 provides an overview of the dataset generation process.

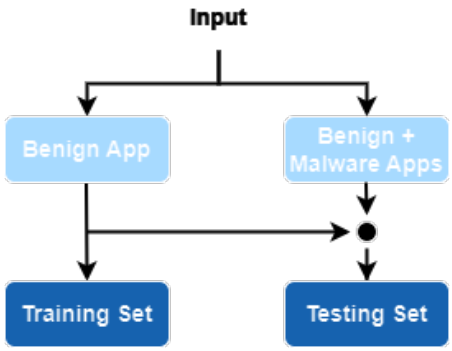

Figure 4.1: Overview Dataset Generation

Two datasets are created:

- **Training Dataset**: This dataset is used to train the ML classifiers to recognise the patterns of benign applications. In anomaly detection, the training set is crucial, as the algorithm learns exclusively from benign data, enabling it to become adept at identifying deviations from normal patterns.

- **Test Dataset**: This dataset is used to evaluate the performance of the trained classifier. It contains both benign and malicious applications and is used to assess the classifier's ability to correctly distinguish between benign and malicious behaviour. The dataset is balanced, with 50% of the samples randomly selected from the Training Set, while the remaining 50% are randomly drawn from the results of the app+malware executions. The six chosen malware are executed for each benign application, with each malware instance contributing to the data set. The malware samples are randomly drawn from these executions, ensuring that the malware portion is evenly represented across all six malicious applications and preventing any single malware from dominating the dataset. This approach guarantees a comprehensive coverage

of malicious behaviours allowing the classifier to be tested fairly across both benign and malicious data, and preventing bias in the evaluation process.

The tables 4.10 and 4.11 provide a detailed breakdown of the number of samples considered:

Application	**Training Set (Benign)**
SUSAN	10000 samples
GSM	200 samples
NMAP	250 samples
NPING	150 samples
FFMPEG	550 samples
RESTIC	200 samples
UNRAR	150 samples
AG	350 samples

Table 4.10: Training Set for each application

Application	**Training Set (50%)**	**Malware Set (50%)**	**Total Test Set**
SUSAN	5000 samples	5000 samples	10000 samples
GSM	100 samples	100 samples	200 samples
NMAP	125 samples	125 samples	250 samples
NPING	75 samples	75 samples	150 samples
FFMPEG	275 samples	275 samples	550 samples
RESTIC	100 samples	100 samples	200 samples
UNRAR	75 samples	75 samples	150 samples
AG	175 samples	175 samples	350 samples

Table 4.11: Test Set composition for each application
Note: The test set consists of 50% data from the training set and 50% from malware samples.

4.2 Malware Detection Process

The final step of the evaluated HMD framework involves employing ML classifiers. These classifiers are trained using the input training set, following the principles of anomaly detection. Once trained, the performance of these detectors is evaluated using the generated test sets.

4.2.1 Introduction to Machine Learning

ML is a subset of AI that includes a variety of techniques that enable systems to automatically identify patterns and structures in data. By leveraging these patterns, ML algorithms can make predictions, support decision-making, and automate tasks[47]. These algorithms learn from data and improve their performance over time without needing explicit programming. This learning process involves training a model on a dataset, enabling it to identify patterns and relationships within the data. Once trained, the model can be applied to predict or classify new, unseen data. In this thesis, ML algorithms are used to automatically classify applications as benign or malicious, allowing the development of an accurate and efficient malware detection system.

There are various learning techniques, and ML is generally categorised into two main types.

In **Predictive** or **Supervised learning** approach the model is trained on labelled data, meaning the input data is paired with the correct output. The goal is for the model to learn the relationship between inputs and outputs so it can predict outcomes for new, unseen data. Classification and regression are examples of supervised algorithms.

The second main type of ML is the **Descriptive** or **Unsupervised learning** approach, where the model is given data without explicit labels. The goal is to uncover hidden patterns, structures, or anomalies within the data itself. Popular

unsupervised learning algorithms include K-means clustering and anomaly detection. Another approach is the **Semi-supervised** learning method, which combines both labelled and unlabeled data. This technique leverages the smaller amount of labelled data to guide learning while making use of the larger, unlabeled portion to improve model accuracy.

In HMD, both supervised and unsupervised techniques are utilised. In the supervised approach, the ML classifier is trained on labelled data, where each input is explicitly labelled as either malware or a benign application. This allows the classifier to learn how to distinguish between the two categories.

In contrast, the unsupervised approach involves training the model on a dataset that consists solely of benign applications. The objective here is for the model to learn the normal patterns of benign behaviour. When new, unseen data is introduced, any significant deviations from the learned patterns may be flagged as potentially malicious.

In the HMD framework assessed in this thesis, anomaly detection, a technique closely related to unsupervised learning, is employed. This approach offers two key advantages: it removes the need for a labelled malware dataset during training, as the classifier is trained exclusively on benign applications, and it enables the system to detect unknown or zero-day malware. However, the downside of this method is that the analysis is more challenging to manage and requires a more complex hardware implementation.

4.2.2 Machine Learning Classifiers

In the realm of machine learning, various classification algorithms can be applied within the HMD framework, typically categorised into different families. For this thesis, four distinct unsupervised anomaly detection algorithms have been chosen: **One-class SVM**, **Local Outlier Factor**, **Isolation Forest**, and **Elliptic**

Envelope. These algorithms have been implemented and experimentally validated within the HMD framework to evaluate their accuracy in detecting malicious applications.

- **One-Class Support Vector Machine (OC-SVM)**: It is a variant of the standard SVM algorithm. It works by constructing a hyperplane that encloses the majority of normal data points, assuming that normal data points are tightly clustered and closer to each other, while anomalies (such as malicious activities) are further away from this cluster[42]. Data points that fall outside this boundary are flagged as anomalies. OC-SVM is particularly effective when the number of anomalous instances is relatively small compared to normal data.

- **Local Outlier Factor(LOF)**: It works by calculating the local density of each data point and comparing it to the densities of its nearest neighbours[48][42]. If a data point's local density is significantly lower than that of its neighbours, it is considered an outlier. This suggests that the point does not belong to the same cluster or distribution as its neighbours, indicating it could be an anomaly. These outliers could represent new, unseen malware samples that exhibit abnormal behaviour compared to benign applications. LOF is particularly effective in datasets where normal points form dense clusters, and outliers are sparse or isolated.

- **Isolation Forest (IF)**: It works by recursively partitioning the data into a tree structure, aiming to isolate each point[49][42]. The idea is that anomalies, due to their rarity and distinction from the rest of the data, require fewer splits to be isolated compared to normal instances. This makes anomalies easier to separate. IF is particularly efficient for detecting outliers in high-dimensional datasets with many irrelevant or redundant features. Its scalability makes it well-suited to handle such complex data structures.

- **Elliptic Envelope (EE)**: It works by constructing an elliptical boundary around the normal data points in a high-dimensional space[50][42]. The model assumes that the data follows a Gaussian distribution and estimates the covariance matrix of the data to determine the shape and orientation of the ellipse. Any data points that fall outside this elliptical region are flagged as anomalies, which can be indicative of malicious activity. EE is particularly effective when the data points are normally distributed and can be accurately represented by an elliptical shape.

These models play a crucial role in ensuring the effectiveness of the HMD framework by providing diverse approaches for anomaly detection. Each model offers distinct advantages and trade-offs, making it important to evaluate their performance in the context of the specific characteristics of the data. All four models can be implemented using the Scikit-learn library in Python. The following key configurations were applied for the classifiers used in Scikit-learn:

- OC-SVM: A non-linear kernel (Radial Basis Function (RBF)) with nu set to 0.01.
- LOF: contamination set to 0.01 and novelty enabled.
- IF and EE: Both used contamination set to 0.01 and random_state set to 0.

These configurations were chosen to handle rare anomalies (with contamination set to 0.01) and to capture complex patterns in the data (non-linear kernel in OC-SVM). The novelty setting in LOF allows the detection of new outliers, and random_state ensures reproducibility in IF and EE.

Chapter 5

Experimental Results

In this final section, the experimental results are presented, highlighting key performance metrics and visualisations, such as PCA, to demonstrate the effectiveness of the classifiers. The primary metric used to evaluate ML classifier performance is *Accuracy*, measured by analysing subsets of 1, 2, 4, 8, 16, and 24 hardware events. In most cases, the classifiers achieved accuracy rates exceeding 80%. An intriguing observation regarding HPC values is that some simulation results, particularly for complex inputs or those requiring extensive execution time in Gem5, exhibit either negative HPC values or values significantly larger than others. To address this, both the training and test sets for each application (eight in total) were analysed under two conditions.

- Keeping the negative values as they are.
- Removing the negative values and retaining only the positive ones.

In both scenarios, the extremely large values were retained, as they may reflect the extended execution time and the increased computational resources utilised by the system during these demanding simulations.

Feature Selection Contribution

The FS process helps identify the most critical HPCs for malware detection. After initial manual removal of irrelevant or always-zero HPCs, FS ranks the remaining features based on their significance. The table 5.1 presents the HPCs rankings according to PCA. Notably, the **Number of instructions committed** is consistently ranked as the most important feature across all applications. This suggests that malware often behaves differently from benign applications in terms of instruction execution, either by executing an unusually high volume of instructions or generating abnormal instruction patterns, both of which can signal malicious behaviour. Other highly ranked HPCs include **L1 data cache accesses** and **Data Memory Access**, frequently ranked 2nd or 3rd. These counters indicate that memory access patterns are highly informative. Malware often exhibits irregular memory access behaviour, such as frequent cache access or unusual memory usage, particularly in attacks like buffer overflows, where memory manipulation is common. Conversely, least-ranked HPCs, such as **ERET instructions**, **pipeline nukes**, and **branch misfetches**, represent hardware events that occur infrequently or have minimal impact on distinguishing benign from malicious applications. These counters, while not completely irrelevant, probably offer little predictive value in malware detection due to their rarity and limited variability during typical execution.

HPC	**susan**	**gsm**	**nmap**	**nping**	**ffmpeg**	**restic**	**unrar**	**ag**
Number of instructions committed (Count)	1	1	1	1	1	1	1	1
L1 data cache accesses	2	3	2	4	4	2	2	2
Data Memory Access	3	4	3	5	5	3	3	3
Number of BP lookups	5	5	5	3	8	4	5	4
Number of load instructions	4	2	4	2	3	9	4	5
Number of control flow instructions	6	8	7	6	7	5	7	6
L1 Instruction cache access	7	7	9	7	2	6	6	7
Number of conditional branches predicted	9	9	8	8	10	7	8	8
Number of store instructions	8	6	6	9	6	8	9	9
L2 cache misses	14	11	10	10	12	12	12	10
L1 data cache misses	10	13	11	13	9	10	10	11
Number of branch mispredictions	11	10	12	12	11	13	11	12
L1 instruction cache misses	13	14	21	15	16	14	18	13
L2 cache write-backs	17	15	13	11	13	18	13	14
L2 cache hits	21	12	16	14	15	20	16	15
L1 data cache write-backs	12	18	17	21	14	19	14	16
DTLB Misses	15	19	14	18	17	17	15	17
D-TLB miss - Read	18	20	15	19	18	22	17	18
NumOfExceptions	19	17	19	16	19	11	20	19
Number of branch misfetches	20	16	20	17	21	16	21	20
D-TLB miss - Write	16	23	18	24	20	15	19	21
ITLB Misses	22	24	22	20	22	21	22	22
Number of pipeline nukes	24	21	23	22	23	23	23	23
ERET instructions	23	22	24	23	24	24	24	24

Table 5.1: HPCs Rankings based on PCA for the target applications

Figures from 5.1 to 5.4 show the PCA decomposition for some interesting cases. The figures present the results of the PCA analysis, illustrating the distribution of data points. A clear separation between malware and benign clusters suggests that PCA has captured important features that can aid in detection. However, the effectiveness of detection also depends on the choice of classification algorithm and its ability to leverage these features.

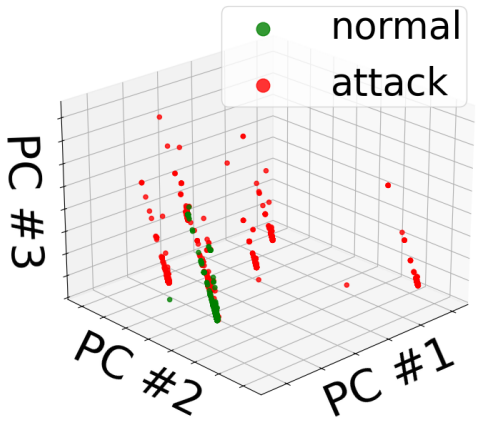

Figure 5.1: Susan without negative values PCA

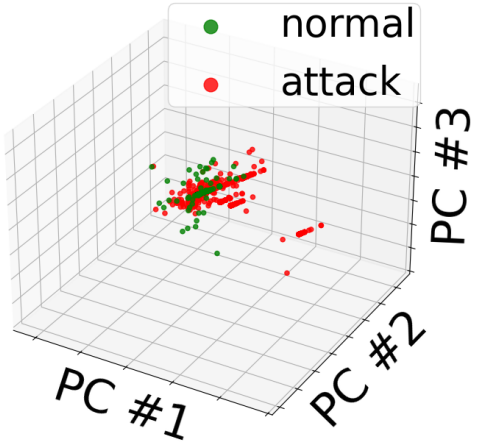

Figure 5.2: Susan with negative values PCA

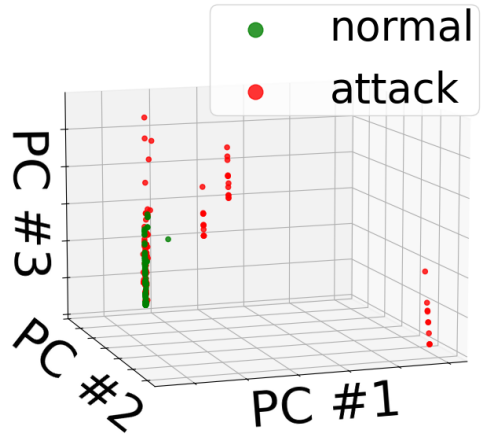

Figure 5.3: Gsm without negative values PCA

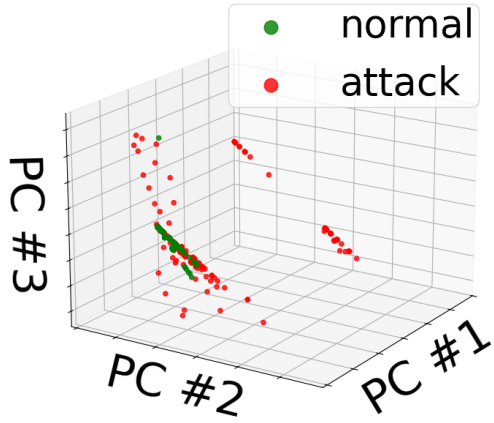

Figure 5.4: Gsm with negative values PCA

Compared to other applications, whose graphs exhibited minimal or unclear separation, the *Susan* and *Gsm* PCA graphs display more discernible clusters, despite some overlap. This suggests that the most important features, captured by the principal components, are effectively differentiating between data points. This aids in distinguishing patterns and anomalies, potentially enhancing detector performance in identifying normal and anomalous data.

Detection Performance

The performance of malware detection heavily depends on the type of ML classifier employed, as well as the number and types of HPCs used to collect features for classification. Figures from 5.5 to 5.8 present the accuracy results for all the tested applications, considering all four ML classifiers employed. The datasets presented in the graphs contain negative values, though similar results were observed with datasets containing only positive values. Generally speaking:

- Elliptic Envelope: Performs inconsistently across different applications. While it achieves high accuracy in the *ag* dataset (up to 94-95%) and *unrar* (up to 77%), its performance in other datasets like *Ffmpeg*, *Nmap*, and *Nping* remains relatively low, with accuracy generally below 60-70%. Although there is some improvement in accuracy with an increasing number of HPCs, the overall accuracy rarely exceeds 80%, indicating that EE does not consistently benefit from additional HPCs and is less effective in many applications.

- Isolation Forest: Shows varying performance, with some applications benefiting significantly from its use, particularly in the *ag* dataset, where it maintains an accuracy of around 97-98% across all HPC levels. However, in the *restic* and *susan* datasets, IF's accuracy improvement is more moderate, moving from 53% with 1 HPC to 75-77% with 16 and 24 HPCs. Other datasets, such as *gsm* and *unrar*, exhibit smaller gains, with accuracy improving gradually from around 50% to around 75% as HPCs increase, but not as dramatically. This suggests that IF benefits from larger HPC configurations across most datasets, but the degree of accuracy gain is not as pronounced in some applications compared to others. Compared to EE, IF shows more consistent performance across HPC levels, with fewer dramatic spikes or drops.

- OneClass SVM: Consistently achieves high accuracy(above 80%), especially with larger numbers of HPCs. This suggests that OC-SVM is better at

leveraging more complex feature sets to enhance detection accuracy. For example, in the *nping* dataset, it achieves 97% accuracy with 16 HPCs, up from 56% with only 1 HPC, indicating a 73% improvement. Similarly, in *restic*, OC-SVM moves from 55% (1 HPC) to 81% (24 HPCs), a 47% accuracy boost.

- Local Outilier Forest: Often has lower accuracy than the other algorithms, suggesting it might be less effective in distinguishing between benign and malicious applications based on HPC features. Its performance is particularly poor for a small number of HPCs, and its accuracy remains around 50-55% in most datasets, even as the number of HPCs increases. In the *unrar* data set, for example, the LOF only increases from 50% to 67% when moving from 1 to 24 HPCs, highlighting its limited ability to leverage additional HPC data. The trend is similar in *ffmpeg*, where LOF remains consistently around 50%.

A particularly interesting aspect of the results is observed with the *Susan* application, with a much larger dataset of 10,000 samples compared to the others, which shows a notable trend in accuracy improvement as the number of HPCs increases. Across all classifiers, even with a small number of HPCs, the initial accuracy is relatively high. For example, with just 1 HPC, accuracies start between 54% and 75%, depending on the algorithm used. As the number of HPCs increases, there is a significant jump in accuracy. By 16 HPCs, most classifiers achieve or exceed 90% accuracy. This demonstrates that larger datasets with more samples provide more detailed information for the classifiers, allowing them to make better distinctions between benign and malicious behaviour.
Overall, the *Susan* dataset highlights the importance of both a sufficiently large sample size and the availability of HPCs to achieve high malware detection accuracy.

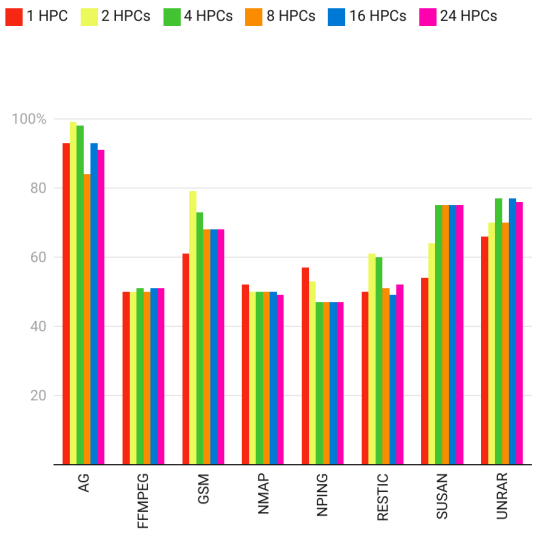

Figure 5.5: EE Accuracy

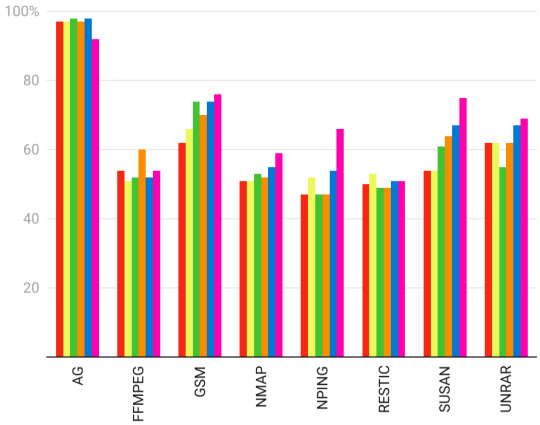

Figure 5.6: IF accuracy

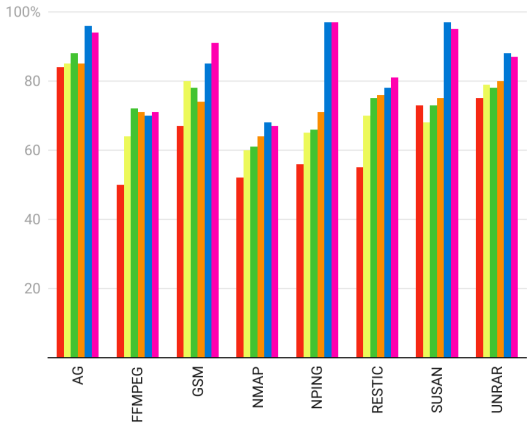

Figure 5.7: OCSVM Accuracy

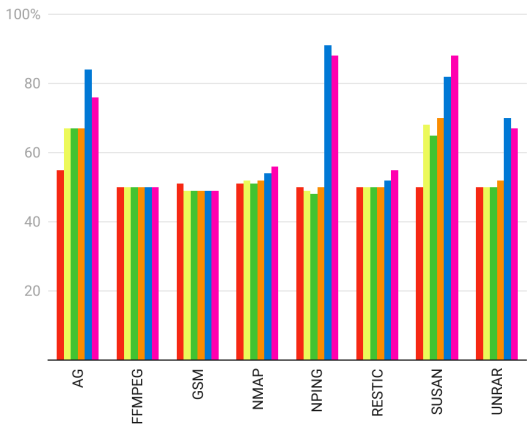

Figure 5.8: LOF Accuracy

The findings suggest that to exceed 80% accuracy across most of the ML classifiers examined, a significant number of HPCs is necessary(16-24), as a limited set of HPCs proves insufficient. Most of the classifiers perform poorly for sets with few features. However, this requirement presents a significant challenge, as modern processors do not provide access to such a large number of HPCs.

Figures from 5.9 to 5.16 show in detail the accuracies of each application. To further complement the analysis presented earlier, these figures provide a more granular view of algorithm accuracy within the context of individual applications. This representation enhances the overall understanding of the algorithms' behaviour and accuracy for each specific application, completing the analysis.

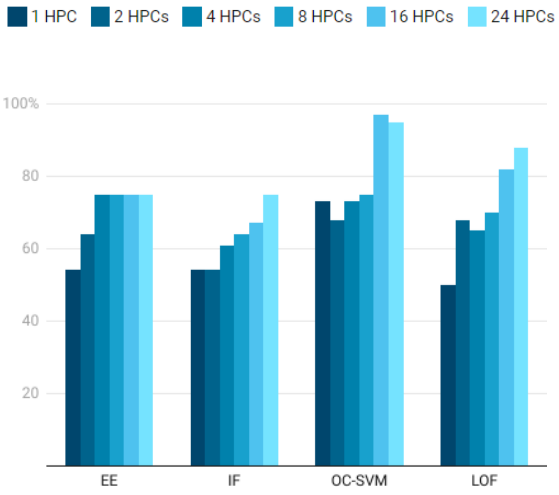

Figure 5.9: Susan Accuracy

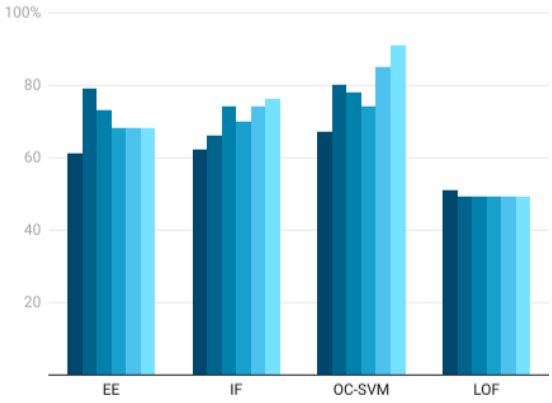

Figure 5.10: Gsm Accuracy

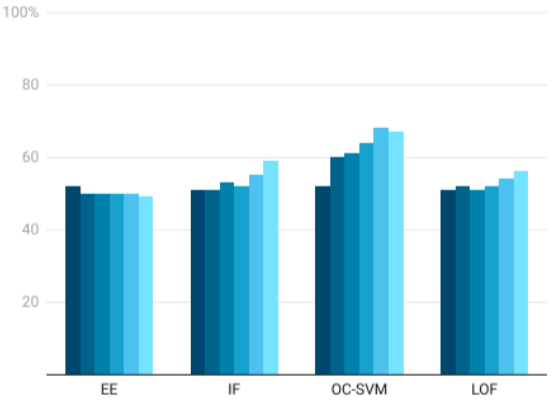

Figure 5.11: Nmap Accuracy

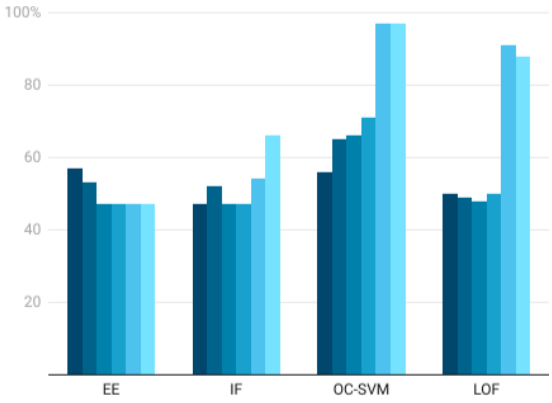

Figure 5.12: Nping Accuracy

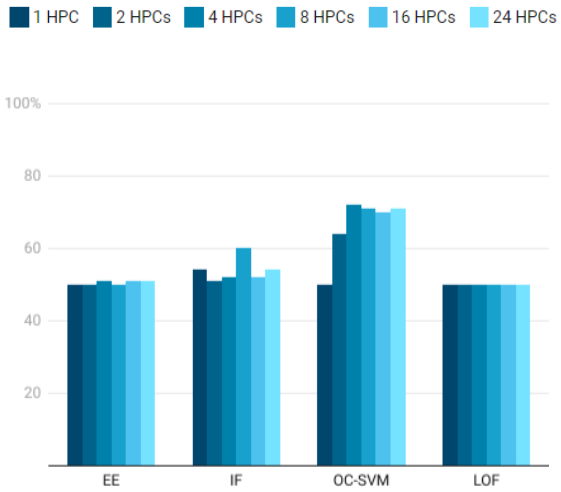

Figure 5.13: FFmpeg Accuracy

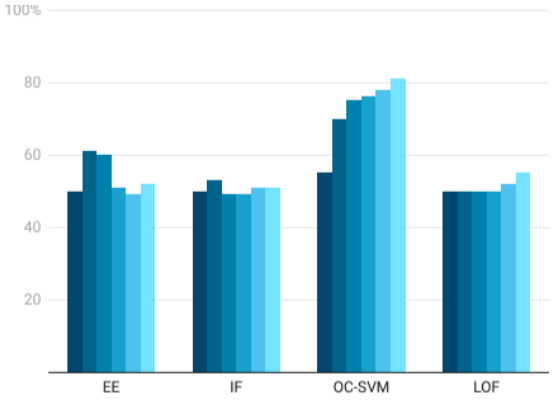

Figure 5.14: Restic Accuracy

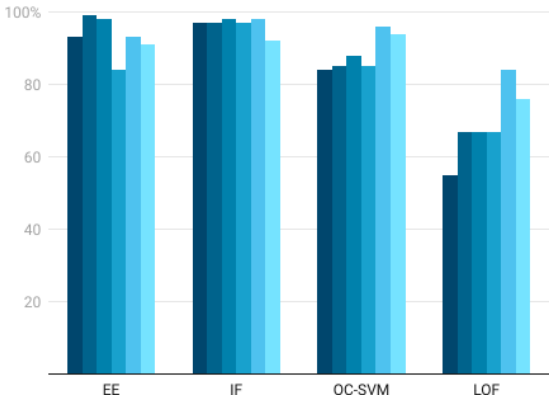

Figure 5.15: Unrar Accuracy

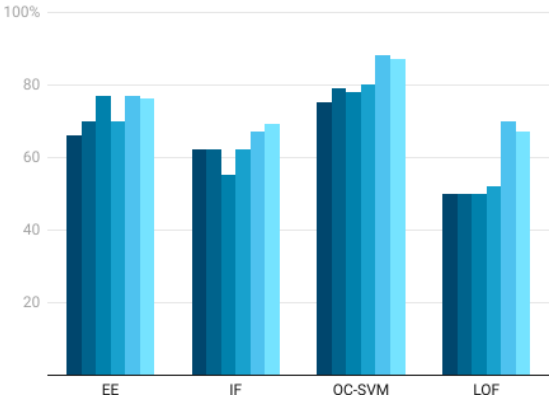

Figure 5.16: Ag Accuracy

Chapter 6

Conclusion and Future Works

This thesis presents a comprehensive exploration of the malware landscape, including an overview of malware classification methods and a review of current detection techniques widely adopted in the research community.

A significant focus was placed on HMD detection, particularly through the use of HPCs and ML techniques. By highlighting the potential of leveraging HPC data, this thesis demonstrated the promising advantages of HMD in terms of resilience to obfuscation, low computational overhead, cost efficiency, and run-time detection. However, despite these advantages, HMD also presents several challenges. One of the primary obstacles is the non-determinism and complexity associated with setting up and maintaining HPC systems. The unpredictable nature of the OS can further complicate this, making it challenging to ensure consistent performance across different configurations and setups. In this context, the simulation environment employed in this thesis is thoroughly analyzed, proposing a full system setup utilizing the Gem5 simulator, based on the RISC-V ISA.

RISC-V, which has gained increasing adoption in modern cloud-computing solutions, offers significant flexibility and scalability, positioning itself as a viable alternative to traditional proprietary architectures.

The experimental results demonstrated varying levels of accuracy across different ML algorithms and target applications. with most classifiers achieving an average accuracy of around 80%. Certain classifiers, such as OC-SVM and IF, achieve higher performance on some datasets. In particular, OC-SVM consistently showed higher accuracy across a broader range of applications, while IF performed well in specific cases. However, for some applications, the accuracy levels, while promising, still require improvement to meet the stringent requirements of real-time malware detection. Differences in the datasets also highlighted the variability in malware behaviour, which suggests that further refinement of FS and classifier optimization is needed to ensure robust and consistent detection performance across all use cases. However, this also presents a challenge, as modern CPUs are still limited in the number of HPCs available, which can restrict the potential for further accuracy improvements. The final findings suggest that with continued development and refinement, an HMD framework built upon HPC and ML could become an invaluable tool in the field of cybersecurity. Such a framework would offer robust defence mechanisms against evolving threats, including unknown and zero-day malware.

Despite the promising potential of HMD, several challenges remain, with detection accuracy being the most significant. The inherent statistical nature of classifiers introduces non-deterministic results, which can lead to variability in detection performance. Ongoing research is focused on reducing these errors by exploring more sophisticated and advanced classifier models.

Additionally, scaling HMD frameworks for cloud computing and large-scale data centres is crucial as these environments continue to expand. Research into distributed systems capable of efficiently gathering and analyzing HPC data across

multiple nodes in real-time is essential for strengthening defences against advanced and sophisticated cyber threats.

Moreover, ensuring the consistency, accuracy, and standardization of HPCs, is essential for building trust and reliability in HMD systems. In this regard, chip manufacturers play a pivotal role by designing dedicated monitoring modules and providing detailed documentation to ensure seamless integration and operation. However, the limited availability of HPCs in mobile and IOT devices introduces a feasibility challenge, particularly in resource-constrained environments. Addressing these challenges will not only drive innovation and enhance the effectiveness of malware detection but also play a pivotal role in securing the evolving cloud infrastructure, ensuring a safer and more resilient digital ecosystem.

Appendix A

Gem5 Configuration Script

This appendix contains the configuration script used to simulate a RISC-V full system using gem5. The script defines the system's architecture, including the CPU, memory hierarchy, cache setup, and peripheral devices. It also includes methods for initialising the platform, handling interrupts, and specifying the bootloader and disk images required for running a Linux workload on a RISC-V architecture.
The modular script allows different CPU types (such as AtomicSimple or O3 CPUs) and supports flexible cache hierarchies. The boot options include parameters to initialize the console and mount the root filesystem from a virtual disk image. This configuration is specifically designed for a HiFive platform, providing a detailed setup for VirtIO devices and memory-mapped I/O (MMIO) components.

```
# RISCV Full System configuration class.
# Attributes:
# bbl (str): Path to the bootloader.
# disk (str): Path to the disk image.
# cpu_type (str): Type of the CPU (e.g., "atomic", "DerivO3").
# num_cpus (int): Number of CPUs to configure.
# script (str): Path to the initialization script
```

```
8
9  class RiscvSystem(System):
10     def __init__(self, bbl, disk, cpu_type, num_cpus,script):
11         super(RiscvSystem, self).__init__()
12
13         ######################################
14         # Initialization and Setup
15         ######################################
16
17         # Set up the clock domain and the voltage domain
18         self.clk_domain = SrcClockDomain()
19         self.clk_domain.clock = '3GHz'
20         self.clk_domain.voltage_domain = VoltageDomain()
21
22         # Create the memory range (2GB starting at 0x80000000)
23         self.mem_ranges = [AddrRange(start=0x80000000, size='2GB')]
24
25         # Create the main memory bus
26         self.membus = SystemXBar() # 64-byte width
27         self.membus.badaddr_responder = BadAddr()
28         self.membus.default =self.membus.badaddr_responder.pio
29
30         # Set up the system port for functional access from the
       simulator
31         self.system_port = self.membus.cpu_side_ports
32
33         if script is not None:
34             self.readfile = script
35
36         # Create the CPUs for the system.
37         self.createCPU(cpu_type, num_cpus)
38
39         # HiFive platform
```

```
40          self.platform = HiFive()
41          self.platform.pci_host.pio = self.membus.mem_side_ports #
    added by me
42
43          # create and intialize devices
44          self.initDevices(self.membus, disk,num_cpus)
45
46          # Create the cache heirarchy for the system.
47          self.createCacheHierarchy()
48
49          self.createMemoryControllerDDR4()
50
51          self.setupInterrupts()
52
53          # using RiscvLinux as the base full system workload
54          self.workload = RiscvLinux()
55
56          # workload object is the bbl
57          self.workload.object_file = bbl
58
59          generateDtb(self) #function from linux_fs.py
60          self.workload.dtb_filename = path.join(m5.options.outdir, '
    device.dtb')
61          self.workload.dtb_addr = 0x87e00000
62
63          # Boot options: Console device, root filesystem, and read/
    write mode
64          boot_options = [
65              "console=ttyS0",
66              "root=/dev/vda",
67              "rw"
68          ]
69          self.workload.command_line = " ".join(boot_options)
```

```
70
71      ####################################
72      # CPU Creation and Cache Hierarchy
73      ####################################
74
75      def createCPU(self, cpu_type, num_cpus):
76          if cpu_type == "atomic":
77              self.cpu = [AtomicSimpleCPU(cpu_id = i)
78                          for i in range(num_cpus)]
79              self.mem_mode = 'atomic'
80          elif cpu_type == "DerivO3":
81              self.cpu = [RiscvO3CPU(cpu_id = i)
82                          for i in range(num_cpus)]
83              self.mem_mode = 'timing'
84          else:
85              m5.fatal("No CPU type {}".format(cpu_type))
86
87          for cpu in self.cpu:
88              cpu.createThreads()
89
90      def createCacheHierarchy(self):
91          class L1Cache(Cache):
92              """Simple L1 Cache """
93              assoc = 16
94              size = '64kB' #512
95              tag_latency = 1
96              data_latency = 1
97              response_latency = 1
98              mshrs = 30 # 16
99              tgts_per_mshr = 20
100
101             def __init__(self):
102                 super(L1Cache, self).__init__()
```

```
103
104             def connectBus(self, bus):
105                 """Connect this cache to a memory-side bus"""
106                 self.mem_side = bus.cpu_side_ports
107
108             def connectCPU(self, cpu):
109                 raise NotImplementedError
110
111         class L1ICache(L1Cache):
112            def connectCPU(self, cpu):
113                 """Connect this cache's port to a CPU icache port"""
114                 self.cpu_side = cpu.icache_port
115
116         class L1DCache(L1Cache):
117             def connectCPU(self, cpu):
118                """Connect this cache's port to a CPU icache port"""
119                self.cpu_side = cpu.dcache_port
120
121         class L2Cache(Cache):
122              """Simple L2 Cache"""
123              size = '128kB' #1024
124              assoc = 8
125              tag_latency = 20
126              data_latency = 20
127              response_latency = 20
128              mshrs = 20
129              tgts_per_mshr = 12
130
131              def __init__(self):
132                super(L2Cache, self).__init__()
133              def connectCPUSideBus(self, bus):
134                self.cpu_side = bus.mem_side_ports
135
```

```
136             def connectMemSideBus(self, bus):
137                 self.mem_side = bus.cpu_side_ports
138
139
140         self.l2_cache=L2Cache()
141         self.tol2bus = L2XBar()
142         self.l2_cache.connectCPUSideBus(self.tol2bus)
143         self.l2_cache.connectMemSideBus(self.membus)
144
145         for cpu in self.cpu:
146             # Create an L1 instruction, data and mmu cache
147             cpu.icache = L1ICache() # cache for Instruction
148             cpu.dcache = L1DCache() # cache for Data L1Cache
149             cpu.mmucache = L1Cache()
150             cpu.icache.connectCPU(cpu)
151             cpu.dcache.connectCPU(cpu)
152             cpu.icache.connectBus(self.tol2bus)
153             cpu.dcache.connectBus(self.tol2bus)
154             cpu.mmucache.mmubus = L2XBar()
155             cpu.mmucache.cpu_side = cpu.mmucache.mmubus.
    mem_side_ports
156             cpu.mmucache.mem_side = self.membus.cpu_side_ports
157
158             # Connect the itb and dtb to mmucache
159             cpu.mmu.connectWalkerPorts(
160                 cpu.mmucache.mmubus.cpu_side_ports, cpu.mmucache.
    mmubus.cpu_side_ports)
161
162     # create the interrupt controller CPU and connect to the membus
163     def setupInterrupts(self):
164         for cpu in self.cpu:
165             cpu.createInterruptController()
166
```

```
167     def createMemoryControllerDDR4(self):
168         self.mem_cntrls = [
169             MemCtrl(dram = DDR4_2400_8x8(range = self.mem_ranges[0]),
170                     port = self.membus.mem_side_ports)]
171 
172     ##############################################
173     # Device Initialization (VirtIO, Bridge, MMU)
174     ##############################################
175 
176     def initVirtIO(self, disk):
177         #Initialize VirtIO and set up the disk image.
178         # Create the CowDiskImage with the given disk path
179         image = CowDiskImage(child=RawDiskImage(read_only=True),
    read_only=False)
180         image.child.image_file = disk
181 
182         # Initialize VirtIOMMIO device for the platform
183         self.platform.disk = RiscvMmioVirtIO(
184             vio=VirtIOBlock(image=image),
185             interrupt_id=0x8,
186             pio_size=4096,
187             pio_addr=0x10008000  # Using reserved memory space
188         )
189 
190     def initBridge(self):
191         #Initialize bridge between memory and I/O buses.
192         # Create the I/O crossbar (IOXBar)
193         self.iobus = IOXBar()
194 
195         # Set up the Real-Time Clock (RTC) for the platform
196         self.platform.rtc = RiscvRTC(frequency=Frequency("100MHz"))
197         self.platform.clint.int_pin = self.platform.rtc.int_pin  #
    Connect RTC to CLINT
```

```
198
199         # Create the Bridge between the I/O bus and the memory bus
200         self.bridge = Bridge(delay='50ns')
201         self.bridge.mem_side_port = self.iobus.cpu_side_ports
202         self.bridge.cpu_side_port = self.membus.mem_side_ports
203         self.bridge.ranges = self.platform._off_chip_ranges()
204
205         # Connect on-chip and off-chip IO
206         self.platform.attachOnChipIO(self.membus)
207         self.platform.attachOffChipIO(self.iobus)
208
209         # Attach the PLIC (Platform-Level Interrupt Controller)
210         self.platform.attachPlic()
211
212     def setNumCores(self, num_cpus):
213         #set the number of cpu
214         self.platform.setNumCores(num_cpus)
215
216     def initMMU(self):
217         #Initialize MMU and set up PMA checker for each CPU.
218         uncacheable_range = [
219             *self.platform._on_chip_ranges(),
220             *self.platform._off_chip_ranges()
221         ]
222         for cpu in self.cpu:
223             cpu.mmu.pma_checker = PMAChecker(uncacheable=
    uncacheable_range)
224
225     def initDevices(self, membus, disk, num_cpus):
226         # Initialize devices, bridge, and platform components.
227         # Store the memory bus for later use
228         self.membus = membus
229
```

```
230         # Initialize VirtIO and disk
231         self.initVirtIO(disk)
232 
233         # Set up the bridge between memory and I/O buses
234         self.initBridge()
235 
236         # Set the number of cores on the platform
237         self.setNumCores(num_cpus)
238 
239         # Set up PMA checkers (Physical Memory Attributes)
240         self.initMMU()
```

Bibliography

[1] International Data Corporation. *IDC Worldwide Semiannual Public Cloud Services Tracker 2023*. Available at: https://www.idc.com/getdoc.jsp?containerId=prUS52343224. Accessed: 2024-10-01. 2024 (cit. on p. 1).

[2] Forbes. *Prediction: Microsoft Azure To Reach $200 Billion In Revenue By 2028*. Available at: https://www.forbes.com/sites/bethkindig/2024/09/05/prediction-microsoft-azure-to-reach-200-billion-in-revenue-by-2028/. Accessed: 2024-10-01. 2024 (cit. on p. 2).

[3] A. K. Marnerides, M. R. Watson, N. Shirazi, A. MAuthe, and D. Hutchison. «Malware Analysis in Cloud Computing: Network and System Characteristics». In: *School of Computing and Communications, Lancaster University, UK* (2013) (cit. on p. 2).

[4] NIST. *Glossary*. Available at: https://csrc.nist.gov/glossary. Accessed: 2024-09-31 (cit. on pp. 2, 7).

[5] M. Nieles, K. Dempsey, and V. Y. Pillitteri. «An Introduction to Information Security». In: *NIST Special Publication 800-12 Revision 1* (2018). Accessed: 2024-09-31, pp. 23–24 (cit. on pp. 2, 7).

[6] N. Binkert et al. «The gem5 Simulator». In: *ACM SIGARCH Computer Architecture News* 39 (2011), pp. 1–7 (cit. on pp. 4, 25, 26).

[7] Statista Ani Petrosyan. *State of malware worldwide - Statistics & Facts*. Available at: https://www.statista.com/topics/8338/malware/#:~\protect\protect\leavevmode@ifvmode\kern+.2222em\relaxtext=In%202023%2C%206.06%20billion%20malware,ransomware%2C%20trojans%2C%20and%20backdoor.. Accessed: 2024-10-02. 2024 (cit. on p. 7).

[8] IBM X-Force. *IBM X-Force Threat Intelligence Index 2024*. 2024 (cit. on p. 7).

[9] Norton Security Clare Stouffer. *115 cybersecurity statistics + trends to know in 2024*. Available at: https://us.norton.com/blog/emerging-threats/cybersecurity-statistics. Accessed: 2024-10-02. 2022 (cit. on p. 7).

[10] Cybersecurity Ventures Steve Morgan. *Cybercrime To Cost The World $10.5 Trillion Annually By 2025*. Available at: https://cybersecurityventures.com/cybercrime-damages-6-trillion-by-2021/. Accessed: 2024-10-02. 2020 (cit. on p. 8).

[11] K. A. Rhodes. «Information Security: Code Red, Code Red II, and SirCam Attacks Highlight Need for Proactive Measures». In: *United States General Accounting Office(GAO)* (2001). Accessed: 2024-09-31 (cit. on p. 8).

[12] E. Cavaciuti-Wishart, S. Heading, K. Kohler, and S. Zahidi. «The Global Risks Report 2024: 19th Edition». In: *World Economic Forum Global Risks* (2024) (cit. on p. 8).

[13] A. P. Namanya, A. Cullen, I. U. Awan, and J. P. Disso. «The World of Malware: An Overview». In: (2018), pp. 420–421 (cit. on pp. 9, 14, 15).

[14] M. Christodorescu, S. Jha, D. Maughan, D. Song, and C. Wang. *Advances in Information Security: Malware detection 1st ed.* Springer, 2007 (cit. on p. 9).

[15] C.P. Chenet, A. Savino, and S. Di Carlo. «A survey on hardware-based malware detection approaches». In: *IEEE Transactions on Emerging Topics in Computing* (Apr. 2024). DOI: 10.1109/TETC.2024.3388716 (cit. on pp. 9, 12, 13, 16–18, 20–23, 49, 56).

[16] Trend Micro. *EMOTET*. Available at: https://www.trendmicro.com/vinfo/us/threat-encyclopedia/malware/emotet (cit. on p. 11).

[17] B. B. Rad, S. Ibrahim, and M. Masrom. «Camouflage In Malware: From Encryption to Metamorphism». In: *IJCSNS International Journal of Computer Science and Network Security* 12 (Aug. 2012) (cit. on p. 12).

[18] P. Mathur and N. Idika. «A survey of malware detection techniques». In: (2007) (cit. on pp. 14, 17).

[19] R. Sihwail and K.Omar K. A. Z. Ariffin. «A Survey on Malware Analysis Techniques: Static, Dynamic, Hybrid and Memory Analysis». In: *International Journal on Advanced Science Engineering Information Technology* 2018 (2018) (cit. on p. 15).

[20] O Aslan and R. Samet. «A Comprehensive Review on Malware Detection Approaches». In: *IEEE* (2020) (cit. on pp. 15–17).

[21] N. Patel, A. Sasan, and H. Homayoun. «Analyzing Hardware Based Malware Detectors». In: *Proc. 54th ACM/EDAC/IEEE Design Autom. Conf. (DAC)* (2017), pp. 1–6 (cit. on pp. 17, 56).

[22] M. Alonso et al. «Validation, verification, and testing (VVT) of future RISC-V powered cloud infrastructures: The vitamin-V horizon Europe project perspective D2.1». In: *Proc. IEEE Eur. Test Symp. (ETS)* (2023), pp. 17–20 (cit. on pp. 17, 18, 29).

[23] T. Sherwood, E. Perelman, G. Hamerly, S.Sair, and B. Calder. «Discovering and exploiting program phases». In: *IEE Micro* 23.6 (2003), pp. 84–93 (cit. on p. 17).

[24] E. W. L. Leng, M. Zwolinski, and B. Halak. «Hardware Performance Counters for System Reliability Monitoring». In: *IEEE International Verification and Security Workshop (IVSW)* (2017), pp. 1–2 (cit. on p. 21).

[25] B. Sprunt. «The basics of performance-monitoring hardware». In: *IEE Micro* 22.4 (2002), pp. 64–71 (cit. on pp. 21, 22, 49).

[26] H. Sayadi, N. Patel, S Manoj, A. Sasan, S. Rafatirad, and H. Homayoun. «Ensemble Learning for Effective Run-Time Hardware-Based Malware Detection: A Comprehensive Analysis and Classification». In: *Proc. 55th ACM/ESDA/IEEE Design Autom. Conf. (DAC)* (June 2018), pp. 1–6 (cit. on p. 22).

[27] S. Das, J. Werner, M. Antonakakis, M. Polychronakis, and F. Monrose. «SoK: The Challenges, Pitfalls, and Perils of Using Hardware Performance Counters for Security». In: *2013 IEEE Symposium on Security and Privacy(SP)* (2019), pp. 20–38 (cit. on pp. 22, 23).

[28] J. Lowe-Power. *Gem5 Documentation*. Available at: `https://www.gem5.org/documentation/` (cit. on p. 25).

[29] J. Lowe-Power, A. M. Ahmad, A. Armejach, A. Herrera, A. Roelke, et al. «The gem5 Simulator: Version 20.0+». In: *Preprint submitted on HAL* (2021). URL: `https://inria.hal.science/hal-03100818` (cit. on p. 28).

[30] gem5 Community. *Learning gem5 - Part 1: Cache Configuration*. Accessed: September 27, 2024. 2024. URL: `https://www.gem5.org/documentation/learning_gem5/part1/cache_config/` (cit. on p. 35).

[31] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. «MiBench: A free, commercially representative embedded benchmark suite». In: *IEEE* (), pp. 2–3 (cit. on p. 38).

[32] *FFmpeg Documentation*. Available at: `https://www.ffmpeg.org/`. Accessed: 2024-10-10 (cit. on p. 38).

[33] *Restic Documentation*. Available at: `https://restic.net/`. Accessed: 2024-10-10 (cit. on p. 38).

[34] *Ag Documentation*. Available at: `https://man.archlinux.org/man/extra/the_silver_searcher/ag.1.en`. Accessed: 2024-10-10 (cit. on p. 39).

[35] *Unrar Documentation*. Available at: `https://linux.die.net/man/1/unrar`. Accessed: 2024-10-10 (cit. on p. 39).

[36] *Nmap Documentation*. Available at: `https://nmap.org/`. Accessed: 2024-10-10 (cit. on p. 39).

[37] *Nping Documentation*. Available at: https://nmap.org/nping/. Accessed: 2024-10-10 (cit. on p. 39).

[38] *MalwareDatabase*. Available at: https://github.com/Endermanch/MalwareDatabase. Accessed: 2024-09-10 (cit. on p. 40).

[39] *SourceFinder*. Available at: https://www.source-finder.org/sourcefinder. Accessed: 2024-09-10 (cit. on pp. 40, 41).

[40] *Vx Underground, MalwareSourceCode*. Available at: https://github.com/vxunderground/MalwareSourceCode. Accessed: 2024-09-10 (cit. on p. 41).

[41] *Gcc-DDOS-Attacks*. Available at: https://github.com/Dev0uss/Gcc-DDOS-Attacks/tree/master. Accessed: 2024-09-10 (cit. on p. 42).

[42] Cristiano Pegoraro Chenet, Ziteng Zhang, Alessandro Savino, and Stefano Di Carlo. *Hardware-based stack buffer overflow attack detection on RISC-V architectures*. 2024. arXiv: 2406.10282 [cs.CR]. URL: https://arxiv.org/abs/2406.10282 (cit. on pp. 49, 61, 62).

[43] I. Guyon and A. Elisseeff. «An introduction to variable and feature selection». In: *Journal of Machine Learning Research* (Mar. 2003) (cit. on p. 52).

[44] J. Li, K. Cheng, S. Wang, F. Morstatter, R. P. Trevino, J. Tang, and H. Liu. «Feature Selection: A Data Perspective». In: *ACM Comput. Surv.* 50.6 (Dec. 2017). URL: http://arxiv.org/abs/1601.07996 (cit. on pp. 52, 53).

[45] G. Papadimitriou et al. «Initial hardware interface requirements and virtual environment specifications D1.1». In: *Proc. IEEE Eur. Test Symp. (ETS)* (2023), pp. 26–30 (cit. on p. 53).

[46] C. Malone, M. Zahran, and R. Karri. «Are Hardware Performance Counters a Cost Effective Way for Integrity Checking of Programs?» In: *Proceedings of the Sixth ACM Workshop on Scalable Trusted Computing, ser. STC '11. New York, NY, USA: Association for Computing Machinery* (2011), pp. 71–76 (cit. on p. 56).

[47] K. P. Murphy. *Machine Learning: a Probabilistic Perspective*. Massachusetts Institute of Technology, 2012 (cit. on p. 59).

[48] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sanderörg. «LOF: identifying density-based local outliers». In: *SIGMOD Rec.* 29.2 (May 2000), pp. 93–104 (cit. on p. 61).

[49] F.T. Liu, K. M. Ting, and Z.-H. Zhou. «Isolation Forest». In: *2008 Eighth IEEE International Conference on Data Mining* (2008), pp. 413–422 (cit. on p. 61).

[50] P. J. Rousseeuw and K. V. Driessen. «A fast algorithm for the minimum covariance determinant». In: *Technometrics* 41.3 (1999) (cit. on p. 62).