2017

Assessing malware detection using hardware performance counters

BOSTON UNIVERSITY

COLLEGE OF ENGINEERING

Thesis

ASSESSING HARDWARE PERFORMANCE COUNTERS FOR MALWARE DETECTION

by

ANMOL GUPTA

B.E., Mumbai University, 2015

Submitted in partial fulfillment of the

requirements for the degree of

Master of Science

2017

Approved by

First Reader

Manuel Egele, PhD
Assistant Professor of Electrical and Computer Engineering

Second Reader

Ajay Joshi, PhD
Associate Professor of Electrical and Computer Engineering

Third Reader

Michel Kinsy, PhD
Assistant Professor of Electrical and Computer Engineering

Mens sana in corpore sano

Acknowledgments

First and foremost, I am endlessly thankful to my advisor, professor Manuel Egele and my co-advisor, professor Ajay Joshi – without whom this work would not have been possible. Their help and support, despite my constant resistance, was the only thing that brought me where I am right now. A special mention to the collaborators and contributors of this project. It goes without saying without the contributions of Boyou Zhou, this project would have never reached its summit. With him, I would also like to thank Leila Delshadtehrani who was always on her feet to help me out.

I would like to express gratitude to my parents, Brijesh and Shobha Gupta, as they were the ones who encouraged me to take on this path, and were morally very helpful throughout. My parents together with my young sibling brother, Devansh Gupta, were there when I needed them the most, and thus should always be thanked for believing in me.

Lastly, I would like to mention all my colleagues in the infamous PHO 340 and PHO 301 rooms, with whom I spent enormous amount of time having off-topic discussions on completely irrelevant topics - Yenai, Fulya, Saiful, Onur(x2), Ozan, QuingQuing, Rushi, Ahmed, Ethan, Zafar, Asselya, Kiran, Aravind, Furkhan and Sadullah. With the conclusion of this project began a new string of invaluable friendship.

Anmol Gupta
Boston University
ECE Department

ASSESSING HARDWARE PERFORMANCE COUNTERS FOR MALWARE DETECTION

ANMOL GUPTA

ABSTRACT

Despite the use of modern anti-virus (AV) software, malware is a prevailing threat to today's computing systems. AV software cannot cope with the increasing number of evasive malware, calling for more robust malware detection techniques. Out of the many proposed methods for malware detection, researchers have suggested microarchitecture-based mechanisms for detection of malicious software in a system. For example, Intel embeds a shadow stack in their modern architectures that maintains the integrity between function calls and their returns by tracking the function's return address. Any malicious program that exploits an application to overflow the return addresses can be restrained using the shadow stack. Researchers also propose the use of Hardware Performance Counters (HPCs). HPCs are counters embedded in modern computing architectures that count the occurrence of architectural events, such as cache hits, clock cycles, and integer instructions. Malware detectors that leverage HPCs create a profile of an application by reading the counter values periodically. Subsequently, researchers use supervised machine learning-based (ML) classification techniques to differentiate malicious profiles amongst benign ones. It is important to note that HPCs count the occurrence of microarchitectural events during execution of the program. However, whether a program is malicious or benign is the high-level behavior of a program. Since HPCs do not surveil the high-level behavior of an application, we hypothesize that the counters may fail to capture the difference in the behavioral semantics of a malicious and benign software.

To investigate whether HPCs capture the behavioral semantics of the program, we recreate the experimental setup from the previously proposed systems. To this end, we leverage HPCs to profile applications such as MS-Office and Chrome as benign applications and known malware binaries as malicious applications. Standard ML classifiers demand a normally distributed dataset, where the variance is independent of the mean of the data points. To transform the profile into more normal-like distribution and to avoid over-fitting the machine learning models, we employ power transform on the profiles of the applications. Moreover, HPCs can monitor a broad range of hardware-based events. We use Principal Component Analysis (PCA) for selecting the top performance events that show maximum variation in the least number of features amongst all the applications profiled. Finally, we train twelve supervised machine learning classifiers such as Support Vector Machine (SVM) and MultiLayer Perceptron (MLPs) on the profiles from the applications. We model each classifier as a binary classifier, where the two classes are 'Benignware' and 'Malware.' Our results show that for the 'Malware' class, the average recall and F2-score across the twelve classifiers is 0.22 and 0.70 respectively. The low recall score shows that the ML classifiers tag malware as benignware. Even though we exercise a statistical approach for selecting our features, the classifiers are not able to distinguish between malware and benignware based on the hardware-based events monitored by the HPCs. The incapability of the profiles from HPCs in capturing the behavioral characteristic of an application force us to question the use of HPCs as malware detectors.

Contents

List of Tables

List of Figures

List of Abbreviations

AMD	.....	Advanced Micro Devices
API	.....	Application Programming Interface
ARM	.....	Advanced RISC Machines
AV	.....	Anti-Virus
CWT	.....	Continuous Wavelet Transform
DT	.....	Decision Trees
DWT	.....	Discrete Wavelet Transform
EBP	.....	Event-based Profiling
FN	.....	False Negative
FP	.....	False Positive
GUI	.....	Graphical User Interface
HPC	.....	Hardware Performance Counter
ISA	.....	Instruction Set Architecture
KNN	.....	K-Nearest Neighbor
ML	.....	Machine Learning
MPL	.....	MultiLayer Perceptron
MSR	.....	Model Specific Registers
NI	.....	National Instruments
OS	.....	Operating System
PC	.....	Principal Component
PCA	.....	Principal Component Analysis
PMC	.....	Performance Monitoring Counter
PMU	.....	Performance Monitoring Unit
SD	.....	Standard Deviation
SMI	.....	System Management Interrupt
SMM	.....	System Management Mode
SVM	.....	Support Vector Machines
TBP	.....	Time-Based Profiling
TLB	.....	Translation Lookaside Buffer
TN	.....	True Negative
TP	.....	True Positive
VM	.....	Virtual Machine
WT	.....	Wavelet Transform
$\mathbb{R}^2$	.....	the Real plane

Chapter 1

Introduction

1.1 Motivation

Leveraging the low-level micro architectural features for providing security is a growing trend among hardware companies. For example, Advanced RISC Machines (ARM) [ARM, 2017a] provides *TrustZone* [ARM, 2017b]. *TrustZone* architecturally divides the hardware into a secure and insecure zone running secure OS and normal OS respectively. The hardware-separated 'secure' and 'insecure' zones are used to separate the execution of user-level applications from the trusted kernel operations. Security is thus maintained using the existing hardware of the device, without affecting system performance. Qualcomm's Snapdragon 835 Mobile Platform uses *Haven* [Qualcomm, 2017] which is a combination of hardware and biometric technologies to secure financial transactions over the Internet. Intel's 4th generation Instruction Set Architecture (ISA) has dedicated instructions, called *AES-NI* [Akdemir et al., 2010], for providing fast and secure encryption-decryption using Advanced Encryption Standards (AES) [Rijmen and Daemen, 2001]. Additionally, Intel also provides platform security by securing the BIOS, the firmware and hardware-based authentication using Boot Guard, BIOS Guard, and Identity Protection Technology (IPT) [Intel, 2017].

The hardware implementations of security features mentioned above provide energy-efficient, low-overhead, and high-performance solutions compared to their software counterparts. However, developing dedicated hardware support for security leads

to a substantially longer time-to-market than software products. National Instruments(NI) state that a hardware product takes on average a year or more to mature from the idea stage to production stage [Instruments, 2014]. Moreover, there is a persistent race between the security developers and their adversaries. One approach for keeping pace in this persistence race between the malware developers and security developers is leveraging the existing hardware units to implement defense mechanisms. Recently, researchers have proposed to use Hardware Performance Counters (HPCs) for malware detection [Demme et al., 2013], [Bahador et al., 2014], [Patel et al., 2017]. HPCs are physical counters embedded in modern processors. These counters are capable of counting the occurrence of a wide range of low-level architectural events such as branch mispredictions and data cache hits. Initially, software designers employed HPCs to characterize and optimize their code's performance on hardware. A malware detector exercising HPCs leverage Machine Learning (ML) classification-based techniques to differentiate between malware and benignware profiles. Each profile contains a time-series of the counter values sampled while monitoring a set of architectural events that transpire during the entire execution of an application.

One of the benefits of using HPCs for malware detection is that the profiling of an application does not interfere with application's execution. Additionally, HPCs incur low-overhead while reading the counts of the events. Recording the counter values of the HPCs is known as *sampling* and the frequency by which the profilers record counts is called the *sampling frequency*. However, there are three main design challenges while using HPCs in malware detectors. Firstly, the low-level instructions executed on a processor and the architectural resource utilization by an application does not reflect the high-level behavior of an application. Whether an application is malicious or benign is a high-level characteristic. Secondly, even though modern pro-

cessors provide an option to monitor more than 200 architectural events on the HPCs, it is unclear which event(s) a malware detector should use to profile an application for predicting its behavior (malignant or benignant). Lastly, a time-series profile of an application profiled using the HPCs is irreproducible in essence. The profiles are not reproducible because the count values corresponding to an architectural event may not repeat when counted multiple times. For example, the number of L1-data cache hits is not the same at every instant across multiple runs of an application. The number of L1-data-cache hits depends on the presence of all the active processes in a system that share one limited sized cache. As a result, the active processes continuously update the cache at every cache-reference. The irreproducible profiles may cause an inefficient training of the ML classifier models.

The challenges listed above raises a question - Do profiles from the HPCs reflect the behavioral difference between malicious and benign applications? In this thesis, we evaluate the robustness of HPCs in capturing the behavioral semantics of a program. To this end, we profile benignware and malware using HPCs and report the performance of a broad range of ML classifiers in classifying the application as benign or malicious. Based on our evaluation in Chapter 3, we have to answer the question in negative.

1.2 Contributions

In this project, we attempt to assess the capabilities of using HPCs for malware detection in challenges mentioned earlier. To this end, we use a methodological framework for profiling benignware and malware on HPCs and then train ML classifiers on these profiles. We profile applications on Intel and AMD processors, running 32-bit Windows 7 operating system (OS).

The following work was completed leading up to this thesis:

***Savitor* - A HPC profiler for AMD**

Our experimental setup on the AMD machine uses *Savitor*. We developed *Savitor* as a user-level application for profiling applications using HPCs. *Savitor* uses kernel-level APIs from AMD CodeAnalyst [Drongowski, 2008] to program the HPCs to monitor a list of desired events. Via the inputs to *Savitor*, a user can specifically monitor a single application while also setting the affinity of all the processes associated with this application. Using the process affinity, a user can force an application to run on one or many cores. *Savitor* will then sample only counters from these cores. Additionally, a user can set the sampling frequency *N*. By profiling the counters every N^{th} fraction of a second a time-series profile of an application is generated. *Savitor* features low overhead and high sampling frequency while creating time-series profiles of applications. Our application list includes 83 malware samples and 64 real-life benignware applications running on Windows. We use VirusTotal [Total, 2012] to download malware samples have known to affect the Windows-7 operating systems previously. For benignware samples, we include applications commonly used on Windows Operating System by consumers worldwide such as Microsoft Office and Internet browsers.

A statistical approach for Feature Selection using PCA

Modern processors provide more than 200 events to monitor on the HPCs. It is unclear which architectural events are an ideal fit for malware detection. Intuitively, the events that create distinct profiles of applications will generate more accurate machine learning models. We apply Principal Component Analysis (PCA) [Wold et al., 1987] on the time-series profile of all the events. PCA converts high-dimensional data into linearly uncorrelated components. Using this property of PCA, we can pick events that show maximum variation in the dataset using the least number of components. The chosen events are then used to profile the remaining applications.

Evaluating our system on a broad range of ML classifiers

We deploy twelve supervised ML algorithms including but not limited to K-Neighbor Classifiers (KNN), Support Vector Machines (SVMs), Decision Trees (DTs) Multi-Layer Perceptron (MLPs) and Logistic Regression. For pre-processing the data, we aggregate the raw samples into 32-binned histograms. Additionally, they are converted to a normally distributed data using power transforms. We report the prediction accuracy of ML classifiers on the testing samples from the datasets on all the twelve classifiers.

1.3 Related Work

Earlier work has proposed to use Hardware Performance Counters (HPCs) for malware detection for one or more classes of malware (such as rootkits). [Demme et al., 2013] first presented a detailed feasibility report on using HPCs for malware detection on Intel and ARM processors. Their analysis includes detection of Android malware, Linux rootkits, and cache side-channel attacks. They achieve prediction accuracies ranging from 100% to 25% across Android malware samples. Additionally, they lack convincing results for side-channel attacks and detection of rootkits. For Android malware, they classify malware based on classes of malware rather than the individual traces of malware. Moreover, their analysis lack any information based on the event selection from a large pool of architectural events, to monitor on the HPCs. As opposed to their method, we present a methodological system of feature selection and feature extraction. For feature extraction, we apply on the time-series profiles from the HPCs, power transform to extract more normally distributed profiles. The power transform extract maximum information while compressing the size of data. Our compression methodology is an extension to the histogram-based binning used by Demme. [Patel et al., 2017] rank a variety of classifiers used for malware detection. The classifiers range from simplistic KNNs to complex MultiLayer Perceptron (MLP). They measure different parameters including accuracy/area, Power Delay Profile (PDP), and testing latency. They include a systematic approach to select the top events from the entire set of available events using WEKA [Witten et al., 1999] and Pearsons correlation coefficient [Pearson, 1901] (Pearsons correlation coefficient determines the linear dependence between two variables). On the contrary, we do not assume that the samples in the data set will possess a strong linear correlation between them. Instead, we use Principal Component Analysis (PCA) to extract linearly uncorrelated 'components' in our dataset. These components reflect maximum

variations amongst samples using the least amount of features.

[Singh et al., 2017] and [Nomani and Szefer, 2015] use HPCs for detection of kernel-level rootkits and defending side-channel attacks, respectively. The former identifies 16 events to detect rootkits. The authors achieve high prediction accuracy in detecting five self-developed synthetic rootkits. All the synthetic rootkits used previously known attack mechanisms such as code-injection and function pointer hooking. Additionally, they collect samples from the HPCs only at the end of execution of the program. Our traces contain a time-series of the sample obtained from the HPCs, thus enabling us to extract the entire execution profile of the application. The latter proposes to use HPCs for segregating the applications into sections called 'phases.' The application is segregated into 'phases' based on the architectural resource employed by each section. Then, the scheduler schedules these 'phases' in a way to minimize the impact of side-channel attacks that leverage sharing of processor resources. Their experimental setup uses only SPEC benchmarks. We believe that SPEC benchmarks do not entirely represent the real-life applications running on modern processors. [Uhsadel et al., 2008] make use of HPCs to detect time-based cache attacks. They exploit cache-based events for every lookup at the L1-cache level. Their implementation monitors cache behavior while executing an OpenSSL [Young, 2017] version of AES [Rijmen and Daemen, 2001]. Unlike our method, they assume that cache is not affected by the processes running on a processor. This assumption can lead to a wide variety of traces for a single application. [Gulmezoglu et al., 2017] show how to exploit web privacy using HPCs. Adversaries can infer user websites running on browsers even in incognito mode. Their results show a prediction accuracy of 70% on average, across many browsers. Unlike our implementation, they lack sophisticated pre-processing of data before training the ML classifiers.

Researchers have suggested both the behavioral and anomaly-based malware detectors using the HPCs. [Bahador et al., 2014] proposes *HPCMalHunter*, a behavioral online malware detector that predicts with high accuracy of 90% with SVMs. They use Singular Value Decomposition (SVD) as their feature reduction method. Like many other papers, they also lack a detailed explanation of how they chose the four events that they monitor on the HPCs. Moreover, their dataset contains 20 benign applications and 11 malware programs. It is not clear if their evaluation can reflect the same results for a larger dataset as well. Tang [Tang et al., 2014] use samples from HPCs to train unsupervised machine learning methods for detecting deviations in program behavior that occur due to a potential malicious attack. They use F-Score as their feature selection and provide a comparison of performance while using different sampling frequencies for the HPCs. They use only two applications namely, Internet Explorer and Adobe Acrobat, in their proof of concept and Metasploit [Metasploit, 2017] to create the exploits for the applications. We avoid using F1-score [Sokolova et al., 2006] for feature selection as it does not reflect any mutual dependence amongst the features and rather highlights only the linear separation between them. One of the important goals of the feature selection in ML is to select only independent features for higher prediction accuracy. Our implementation selects informative features that are mutually exclusive.

1.4 Background

In this section, we briefly introduce the hardware and software-based components, used in our experiments. The former includes Hardware Performance Counter and the Hardware Performance Events. Next we discuss about malware is and classification of malware. Additionally, this section also includes a preface to the feature selection and feature extraction algorithms used in this thesis - i.e., Principal Component Analysis.

1.4.1 Hardware Performance Counters

Hardware Performance Counters (HPCs) are special-purpose counters embedded in a modern microprocessor die; which count a broad range of architectural and microarchitectural events. A variety of processor platforms such as Intel, ARM, and AMD include HPCs on their processors. Depending on the processor, each counter varies from 32 to 64 bits in size. The number of physical registers present on each core usually ranges from 2 to 8. These registers are capable enough to count a myriad of events such as L1/L2/L3 cache access & misses, TLB hits & misses, branch mispredictions, and core stalls of the chip. HPCs are easily programmable across all platforms. The counters are often programmed to throw an interrupt when a counter overflows or even be set to start the counter from the desired value. The start count can also be a negative count, for which the counters count up). The software handles these interrupts allowing programmers to analyze the hardware resource utilization by their applications at run time.

HPCs, thus find themselves a handy tool in tuning and optimizing the low-level architectural performance of running applications [Bulpin and Pratt, 2005]. From per-

formance analysis tools their usage has extended to detecting firmware modification in embedded systems [Wang et al., 2015], estimating system power utilization [Contreras and Martonosi, 2005], and even detection of malware [Demme et al., 2013]. Essentially, software engineers use HPCs for measuring the performance of their code and thus optimizing it.

In a ring-based security model [Wikipedia, 2017c] HPCs belong to ring-0. This makes them accessible only to the kernel (events like cycle-count and timestamp counter make an exception on some processors). The operating systems can program the HPCs using control registers, called Performance Monitoring Counters (PMCs) found in the Performance Monitoring Unit (PMU). These registers are known as Model Specific Registers (MSRs) on Intel processors. User-space applications can access the HPCs through software interfaces to PMUs and configure the HPCs using the PMCs. Some of the commonly used software interfaces include *PAPI* [Mucci et al., 1999] that provide standard APIs for accessing the HPCs. Also, there are different profilers for Linux and Windows operating systems. For example, *perf* [de Melo, 2010] based on perf_event, is a popular tool providing support for HPCs on Linux 2.6+ based hosts. On Windows, one can use Intel's VTunes [Reinders, 2005] for the Intel processors and AMD's CodeAnalyst [Drongowski, 2008] (now CodeXL) for the AMD processors. In our project, we use HPCs to construe a time-series trace of N microarchitectural events by profiling malware and benign applications. Each program executed on CPU may or may not generate a different performance counter trace.

1.4.2 Hardware Performance Events

As mentioned in the section 1.4.1, HPCs can monitor a broad spectrum of performance events across all modern processors. It is the responsibility of the PMUs to keep track of microarchitectural events that occur during process execution. The performance events provide a comprehensive snapshot of a processor's runtime behavior. Software developers may use the traces from performance events to enhance their systems. Based on the architectural aspect monitored, the events are classified as follows:

1. **Hardware Events** - Counters monitoring CPU-based event
2. **Software Events** - Events based on kernel events such as CPU migrations, minor and major faults
3. **Kernel Tracepoint Events** - Static tracepoints for the kernel
4. **User Statically-Defined Tracing (USDT)** - Static tracepoints for user-level applications
5. **Dynamic Tracing** - Dynamically instrumented software events for user-level applications using the 'uprobes' framework and kernel-level operations using the 'kprobes' framework

We consider Hardware Events for our malware detectors because all profilers provide the option of tracking hardware events on all the architectures on any operating system. Some profilers may not feature the remaining kinds of events. For example, the AMD's CodeAnalyst supports static Tracepoint events and does not feature dynamic tracing [Drongowski, 2008]. Figure 1·1 shows a broad range of events available to monitor on the hardware performance events.

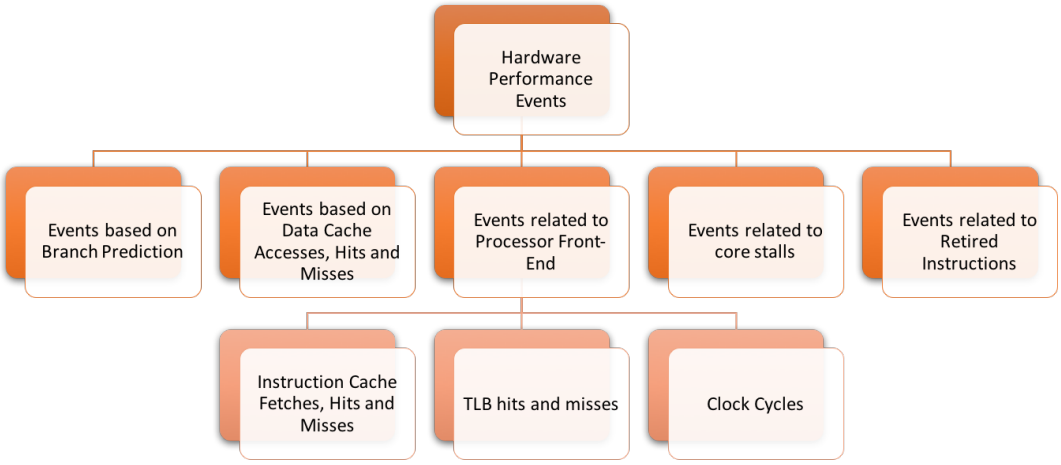


Figure 1·1: Various architectural events available to monitor on the HPCs

There is a limitation on the number of events that can be sampled on the HPCs at any instant across all platforms. The number of physical counters on each core of the processor imposes the restriction. For example, Intel provides the option of monitoring 468 and 519 on the Ivy-bridge and Intel Broadwell CPUs, respectively [Intel, 2011] However, only four events can be sampled simultaneously since the number of counters is limited to four on each core of the processor. On modern processors, the limitation is mitigated by multiplexing performance counters [May, 2001]. Multiplexing incorporates sampling of the events in a round robin manner. Round robin technique monitors the first series of events for a pre-decided time slice. On the expiration of the time slice, HPCs monitor the next sequence of events. Multiplexing then reduces the frequency by which each event is sampled and reduces the samples in the trace.

1.4.3 Malware and Malware Detection

Adversaries create malicious software or malware to unlawfully use a system or compromise the privacy of a victim. Table 1.1 [Demme et al., 2013], [Risks, 2011] describes different classes of malware. Adversaries develop malware for financial gains, espionage or personal information theft [Risks, 2011]. Some ways to publish malware to a potential victim includes phishing emails (broadcasted with malicious attachments), pdfs, software downloads from untrusted sources, accessing web pages injected with exploits, storage devices, or downloading applications from mobile stores. [Risks, 2011], [Demme et al., 2013]

A widely deployed malware detection technique is Signature-based detection [Griffin et al., 2009]. Signature-based detection techniques can detect known categories of malware such as viruses and worms. A typical Anti-Virus (AV) system that uses signature-based detection techniques scans files for known malware signatures, specifically code strings, which are responsible for the functionality of the malware. Signature-based techniques have several significant drawbacks. Firstly, static signature-based detection is computationally intensive as scanning a large size of the database of malware is steadily increasing at a fast pace [Harley and Lee, 2007]. Secondly, the rise of Polymorphic malware produces a new variant of malware using obfuscation techniques such as subroutine reordering and register reassignment. Since signature-based techniques are reactive, they are unable to defend against new malware samples, until the malware signature database is updated [Rad et al., 2011]. Lastly, encrypted malware can potentially delay or avoid detection by static code analysis [Rad et al., 2011].

The pitfalls of signature-based detection techniques motivated the defenders to track behavior or anomalies in applications during their execution. Literature shows

Malware Classes	Description
Virus	Malware found in programs or executables. The processor executes the malware code together with with the program
Worm	Are similar to virus in functional behavior except that they are stand-alone software which does not require any assistance from a host program or human aid for broadcasting
Polymorphic Virus	A virus that is capable of altering its payload to evade detection, while maintaining its functionality
Metamorphic Virus	A virus that alters both the payload and functionality
Trojan	Malware that appears legitimate but acts maliciously once activated
AdWare	Malware that floods a web-page with unwanted advertisements.
SpyWare	Malware that secretly gathers reports user's personal information and grants access to such information to another entity without the user's consent
Ransomware	Malware that blocks access to user data and threatens to publish it unless the user makes a predetermined price
Botnet	Malware that employs an infected system as a node in a network controlled by a central malicious unit called the bot herder
Rootkit	Malware that provides privileged access to a system while hiding its or any other malicious software's presence

Table 1.1: Classes of Malware [Demme et al., 2013], [Risks, 2011]

that behavior-based malware detection techniques [Christodorescu, 2007] track dynamic aspects of programs such as system call traces, control flow graphs, and data flow graphs. Behavior-based techniques overcome some disadvantages of static signature-based detection. For example, tracking behavioral patterns in the malware during runtime helps detect polymorphic malware [Zhao et al., 2010]. On the other hand, the using behavior-based techniques require a secure execution environment for the malware such as a Virtual Machines(VM). Tracking the malware sample in a virtual environment can be time-consuming and give rise to false-alarms [Tian et al., 2010].

Recently, security researchers have proposed to use hardware-based solutions for malware detection [Demme et al., 2013], [Tang et al., 2014], [Kirat et al., 2014]. Hardware-based detectors offer fast online detection, minimize hardware resource utilization, and are inaccessible to user-level applications (unless adversaries have access to the kernel-level privilege of the platform). Intuitively, such qualities make them suitable for mitigating both known and new threats. However, there are several design challenges with hardware-based detectors. Some of them include having the capability of tracking the malicious activities in parallel to the execution of user-level processes, a small logic area on the die and low power overhead for implementation on the processor. On top of that, hardware-based detection techniques often use ML classifiers [Demme et al., 2013], that adds overhead based on the classification algorithm used.

1.4.4 Principal Component Analysis

Principal Component Analysis (PCA) is an approach used to accentuate variation and highlight linear relation in a multivariate data set [Wold et al., 1987]. A broad range of fields such as neuroscience and computer graphics [Rao, 1964] leverage PCA for data analysis. PCA reduces the dimensions of a multivariate dataset, thus extracting relevant information from complex and large datasets using fewer variables. To be more specific, PCA maps a nonlinearly correlated data to orthogonal linearly correlated variables called principal components (PCs). The transformed dimension size of the data is less than or equal to the original dimensions of the data set. PCA arranges the PCs such that the first PC shows the maximum possible explained variance ratio. Explained variance ratio accounts for the proportion of variation in the transformed variable(s) as compared to the original variable(s). Each following PC

contains the next highest variation and is orthogonal to the preceding components.

Figure 1·2 shows a pictorial depiction of applying PCA to transform a 3-dimensional data a 2-dimensional components. It is evident that the data points in the original data space have large projections on the 3-dimensional planes. PCA creates new planes such that the data-points now have small projections on the newly transformed planes. The planes are orthogonal to each other and are linearly correlated. As a result, the same amount of information is carried in the transformed 2-dimensional data space when compared to the original data space.

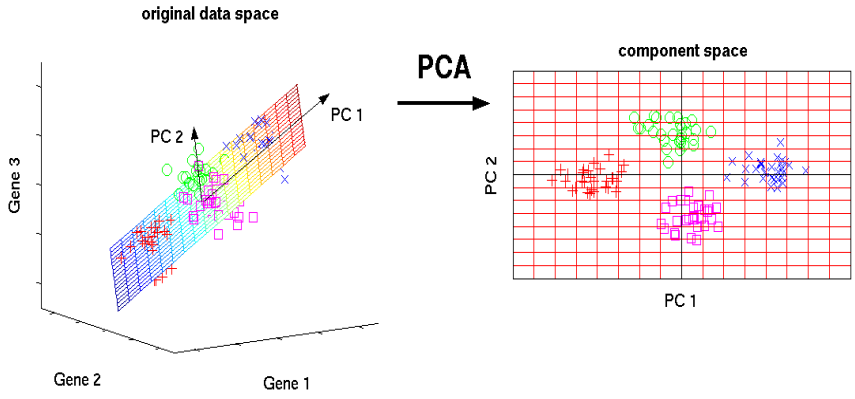


Figure 1·2: PCA demonstration - Converting a 3-dimensional original data space to a 2-dimensional linearly uncorrelated components [Vidhya, 2016]

Mathematically speaking, PCA uses eigenvalue decomposition of a data covariance (or correlation) matrix. We use the derivations from [Shlens, 2014] to explain how the results of PCA are eigenvectors of a matrix. Consider:

- A data set X be an $[m \times n]$ matrix, where m is the number of examples points, and n is the number of features

- $C_x \equiv \frac{1}{n} X X^T$ is covariance matrix of X

Goals of PCA: Find a covariance matrix $C_Y \equiv \frac{1}{n} Y Y^T$ of Y, where Y is a diagonal matrix in some orthonormal matrix P such that $Y = PX$. Then the rows of this orthonormal matrix P are the PCs of X.

Proof: Consider covariance matrix C_Y of Y:

$$\begin{aligned} C_Y &= \frac{1}{n} Y Y^T \\ &= \frac{1}{n} (PX) (PX)^T \\ &= \frac{1}{n} P X X^T P^T \\ &= P \left(\frac{1}{n} X X^T \right) P^T \end{aligned}$$

$$\therefore C_Y = P C_X P^T \tag{1.1}$$

Theorem: *Any symmetric matrix - A is diagonalized by an orthogonal matrix of its eigenvectors. For a symmetric matrix A, $A = EDE^T$, where D is a diagonal matrix and E is a matrix of eigenvectors of A arranged as columns [Shlens, 2014].*

Thus, select a matrix P where each row of P - $\{p_i\}$ is an eigenvector of $\frac{1}{n} X X^T$. From theorem, $P \equiv E_T$ and $P^{-1} = P^T$, C_Y now evaluates to:

$$\begin{aligned} C_Y &= P C_x P^T \\ &= P \left(E^T D E \right) P^T \\ &= P \left(P^T D P \right) P^T \\ &= \left(P P^T \right) D \left(P P^{-1} \right) C_Y \end{aligned}$$

$$\therefore C_Y = D \tag{1.2}$$

From equation 1.2, an orthonormal matrix P diagonalizes C_Y. The PCs of X are the eigenvectors of covariance matrix C_x of X. The i^{th} diagonal elements of covariance matrix C_Y of Y is the variance of X along $\{p_i\}$

The mathematical derivation[1] of PCA impose some limitations. Firstly, Dimensionality reduction using any algorithm causes loss of information, in general. PCA minimizes information loss depending on the original data. Secondly, PCA decomposes a data set into uncorrelated variables removing the second-order dependencies. There are more than one solutions to remove dependencies higher than second-order.

1.4.5 Power Transform

Power transform converts the data into a more normal-like distribution [Box and Cox, 1964]. The normalization using power transform stabilizes the variance in data. Power transform stabilizes the variance by applying a logarithmic function to a data set. A logarithmic function will magnify subtle variations in data and dampen high variations in the data. As a result we obtain an approximately normally-distributed data. A normally distributed data set benefits the linear statistical algorithms like Pearson's Correlation calculation used for processing and analyzing the data. Additionally, a power transform is a monotonic transformation. A monotonic transformation does not change the order of a dataset. What this means is that, if a function is monotonically increasing, then power transformation will preserve this order. The above two mentioned properties of power transform help to find application in a variety of data analysis fields such as medical research [Wikipedia, 2017a]

[1]Note that a discussion on the theorems and other derivation used in the above equations is outside the scope of this research. Refer [Shlens, 2014] and [Wold et al., 1987] for more details

For vectors $(y_1, , y_n)$ in which each $y_i > 0$, the the power transform is given by [Wikipedia, 2017a]:

$$y_i^{\{\lambda\}} = \begin{cases} \dfrac{y_i^{\lambda} - 1}{\lambda(GM(y))^{\lambda-1}} & \text{if } \lambda \neq 0 \\ (GM(y)) \ln y_i & \text{if } \lambda = 0 \end{cases} \tag{1.3}$$

$$\text{where,} \quad \text{GM}(y) = (y_1, y_n)^{\frac{1}{n}} \text{and } \lambda, \text{ is the power parameter}$$

λ determines the point at which the power function is continuous. There are many variations of Power Transform, namely, BoxCox transformation [Sakia, 1992] and Yeo-Johnson transformation [Weisberg,]. We use the Box-Cox version of Power Transform in our implementation. Box-Cox version is a logarithmic transformation. Refer figure 1·3 [Scibilia, 2015] for a plot showing the logarithmic plot. The Box-Cox implementation inflates or magnifies the smaller variations in the dataset (because the slope of the logarithmic function is steep when values are small), and the reduces the larger variations (due to a steady slope at larger values). As a result, we stabilize the variance in our dataset across the events. The stabilized variation enables comparable contribution by each feature while training the ML classifiers.

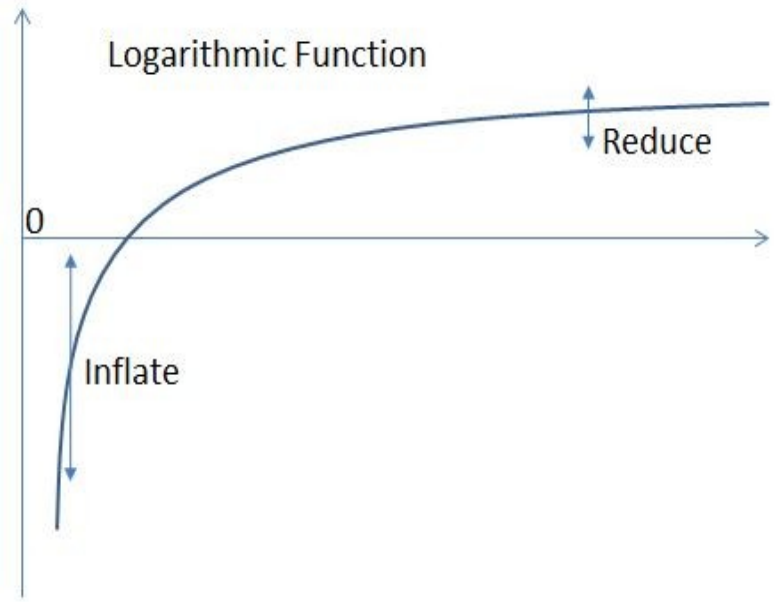


Figure 1·3: Logarithmic Function [Scibilia, 2015]

The remainder of the current dissertation is outlined as follows: Chapter 2 gives a detailed account of the proposed malware detector. Chapter 3 evaluates the performance of the ML classifiers in identifying between malicious and benign applications. Based on the evaluations in chapter 3, we present our conclusion, and possible extensions to this project as future work in chapter 4.

Chapter 2

Malware Detector using HPCs

2.1 Profilers - Introduction

The profilers play a major role in our implementation. Profilers are user-level applications that provide an interface to the PMU. The users can use the profilers to program the HPCs especially to leverage features such as selecting a set of hardware-events to monitor and fixing the sampling frequency for that event from only a particular core. Following are some of the common features provided by HPC-based profilers:

- Setting the **events** to monitor for profiling an application
- Providing a **CPU core mask** to count the occurrence of a hardware event on one or many cores
- Providing **process Ids (PID)** to monitor one or more processes. Some profilers have provisions to input the commands to run an application directly; the profiler will then monitor all the processes and child-processes associated with that command
- Selecting the **sampling frequency/count**. There are two ways to sample the events:
  - ***Counting*** - The kernel will probe the HPCs after an event has occurred for a specified number of times

- ***Sampling*** - The kernel will probe the HPCs after a specified interval of time

Profilers may also provide additional features to users. For example, AMD's CodeAnalyst [Drongowski, 2008] and perf [de Melo, 2010] can also measure the % CPU utilization and % memory utilization by an application.

In our implementation, we leverage Intel's VTunes [Reinders, 2005] on the Intel architecture and AMD's CodeAnalyst [Drongowski, 2008] (now CodeXL) on the AMD architecture. AMD implemented CodeAnalyst on both, Windows and Linux operating systems. CodeAnalyst provides a set of pre-defined profiling options to profile an application. Table 2.1 shows all of the options.

Assess Performance	Event-based Profiling
Instruction-based Profiling	Time-based Profiling
Instruction-based Sampling	Investigate L2 Cache Access
Investigate Branching	Investigate Data Access
Investigate Instruction Access	Thread Profiling

Table 2.1: Predefined Options to profile on AMD's CodeAnalyst

Each option mentioned in table 2.1 has a set of predefined events that the profiler will monitor on the HPCs. For example, the 'Assess Performance' profiling option presents a detailed analysis of system performance for the application under consideration [Drongowski, 2008]. The analysis includes a report on the instructions executed, the data & instruction cache accesses, the Translation Lookaside Buffer (TLB) hits & misses and Misaligned accesses in the main memory. To analyze such system features, HPCs monitor the following events, as shown in table 2.2.

Similarly, the Time-Based Profiling (TBP) option identifies the hotspots in a pro-

Retired Instructions	Data Cache Accesses
Data Cache Misses	Data Cache Misses
Data Cache Misses	IData Cache Misses
Unified TLB Hit	Unified TLB Miss
Misaligned access	

Table 2.2: Events profiled for Access Performance Profiling on AMD's CodeAnalyst

gram [Drongowski, 2008]. Hotspots are the maximum time-consuming phases of an application. The time-cost of a phase may be high due to potential memory bottlenecks, execution penalties or lack of optimization opportunities. To identify such hotspots, TBP tracks the following events, as shown in table 2.3:

CPU Clocks	Instruction per Cycle
Data Cache Miss Rate	Data TLB L1/L2 Miss Rate
Misalign Rate	Branch Mispredict Rate

Table 2.3: Events profiled for Time-based Profiling on AMD's CodeAnalyst

It is important to note that the events mentioned in Table 2.3 are preconfigured and cannot be reprogrammed by a user. In TBP, CodeAnalyst configures a hardware timer that periodically interrupts the program executing on a processor core. The PMU samples the counters when a timer interrupt occurs. Post-processing in TBP aggregates the raw sample into a histogram for easy visualization of the profile.

Event-based Profiling (EBP) counts the number of hardware events occurred [Drongowski, 2008]. The PMU needs the following information to configure each counter with the specified event :

- An event to be measured

- An event count (sampling count)
- Choice of OS-space sampling, user-space sampling, or both
- Choice of edge- or level-detect.

PMU will use the above set of information to configure an event on one of the counters in the HPCs. Once the profiling starts, the kernel interrupts the counter as soon as the count for an event reaches the specified sampling count. In EBP, there are no preconfigured events. A user can choose any events from the pool of events provided by the family of the processor. For examples, on the Intel machine the available events to monitor depend on the microarchitecture (such as Nehalem, Haswell, and Skylake) of the processors [Intel, 2011].

For our application, we need features from both, TBP and EBP; i.e., the capability of firing an interrupt once a timer is expired and the flexibility to monitor any desired event. This flexibility provides an opportunity to identify the top events that can predict the functional behavior of a program. Additionally, we desire to read the samples after a given sampling frequency, rather than sampling the HPCs after an event count has reached a certain sampling count. We impose such restriction because our malware detector demands a time-series of the count of occurrence of the architectural event. For example, suppose we monitor misaligned accesses with sampling count set as 1000 sample/second. The sampling count entails that the profiler will fetch the count of the number of misaligned accesses occurred once every millisecond. To facilitate profiling hardware performance events in the time domain, we introduce *Savitor*.

2.2 Savitor

Savitor is a HPC-based profiler implemented on Windows OS. A user can deploy Savitor on an AMD-based processor with any microarchitecture (such as Bulldozer and Athlon). As mentioned in section 2.1, the profiler features profiling of an application in the time-domain. Savitor uses AMD's CodeAnalyst APIs [Drongowski, 2008] at the back end. The CodeAnalyst APIs provide a kernel level functionality to configure the HPCs with a desired set of event and then sample the configured events with a specified sampling frequency. With Savitor the sampling rate can be as fast as 3000 samples per second. The front end of the profiler is an user-interface to provide different profiling options to the user. These options include:

- Provide the **hardware events** to monitor on the HPCs. The number of events inputted can range from 1 to the number of hardware counters available per core on the architecture.
- The **sampling frequency**. The sampling frequency determines the rate at which the HPCs are interrupted to read the counter values
- The **application** to profile. The application can execute on command prompt or could have a Graphical User Interface (GUI)
- The **output file** to store the sampled values. The samples are stored in plain-text and hence can be directly be used for further analysis of the data
- Total **time** to profile. The profiling stops when this timer expires, or the application terminates, whichever happens first.

One of the responsibilities of Savitor is to make sure that profiling of the application is not dependent on the presence of other system processes. To achieve this, we designed Savitor to leverage multithreading on the AMD processors. Each thread

runs on a different core and is assigned a specific task. The first thread called the *application_thread*, spawns a new process with the application to profile. Savitor executes all the process and the subprocesses of the application on the first core of the processor. The counting of the events on the HPCs occurs on a *per-process* basis. As a result, HPCs monitor only the threads of the processes or the subprocess of the application. Forcing an application to run on a single core enables the counters from that core to capture the entire execution profile sequentially; else, the profiles of multithreaded applications will not capture the correct time sequence of the occurrence of the events. Additionally, in a multi-threaded computing system, the resources are shared amongst the processes running on the system. A scheduler manages the scheduling of these processes. The scheduler assigns a priority to each process, and higher priority processes usually get more resource usage time. Since it not possible to alter the current scheduler implementation on Windows OS, Savitor sets these threads to maximum possible priority-*THREAD_PRIORITY_TIME_CRITICAL* belonging to the *REALTIME_PRIORITY_CLASS* [Center, 2017] on the Windows OS. By assigning the highest priority, we ensure minimum preemption of the application's processes (and their threads) by other user-level processes. Only OS-spawned threads can preempt the application's processes. The second thread called the *timer_thread* is responsible for keeping track of time of profiling. Recall that, the user provides a time in second to profile each application. After the time expires, Savitor terminates both the profiling of the application. The *timer thread* runs on the second core. This thread also manages a hardware timer. This hardware timer determines the sampling frequency of the profiles. Once the timer expires, an interrupt is sent to Savitor, asking to read the counter values. A third thread called the *sampling_thread*, on receiving an interrupt from the *timer_thread*, uses the CodeAnalyst APIs to read the counter values. Additionally, the *sampling_thread* also fetches the timestamp at

which the kernel sampled the HPCs. The next thread called the *write_thread* gets the counts from the *sampling_thread* and writes it these sampled counts from the HPCs to the file specified in user-input. The *write_thread* logs the counts is - **timestamp:event:count**. The table 2.4 summarizes the job distribution of each thread in Savitor.

Threads	Thread Job
application_thread	Spawns a new process to run the application to profile
timer_thread	Manages a hardware timer to achieve the sampling frequency desired by the profiler. Additionally, it also terminates profiling of the application based on the time to profile entered as an user-input
sampling_thread	Issues a system call to the kernel using the AMD CodeAnalyst APIs to sample the counter values
write_thread	Write the sampled values to a file of user choice

Table 2.4: Task List of the threads in Savitor

In principle, the sampling of the HPCs and the writing of the sampled values to a file are sequential tasks, even though Savitor assigns the two steps to two different threads. This is because, unless the *sampling_thread* fetches the count value, the *write_thread* cannot log the counts into a file. On the other hand, the until the *write_thread* finishes writing the samples to the output file, the *sampling_thread* has to hold the next fetched samples. To parallelize this process, we use a queue in the memory to hold the samples from the HPCs temporarily. As soon as the *timer_thread* issues an interrupt to the *sampling_thread*, the *sampling_thread* fetches the timestamp from the kernel and the counts from the HPCs. The *sampling_thread* pushes the timestamp, the sampled counts and the event monitored to the queue. This process is

repeated at the rate of the sampling frequency specified by the user. The *write_thread* pops the three data fields (timestamp, events to monitor and the sampled counts) to log and write them to an output file. Using the queue, the sampling and the writing process are isolated. The queue is protected by a mutex to prevent pushing and popping elements from occurring at the same instant. Parallelizing the process of sampling the HPCs and writing the sample to the thread helped us achieve high sampling frequency of 3000 Hz.

2.3 Benignware and Malware Used

In our dataset, we profile 83 malware and 64 benignware. For benignware, we profile applications and services that are commonly used by consumers. We use the Futuremarks PCMark [NIEMEL, 2005] benchmark suite to profile applications running on a Windows machine. PCMark classifies its benchmarks into:

- Essentials
- Productivity
- Digital Content Creation
- Gaming

We use the benchmarks from the 'Essentials' and 'Productivity' class. The Essential workload includes benchmarks like web browsing, one-to-one video calls, and video Conferencing. The Productivity workloads include spreadsheets and other writing related applications. We do not use benchmarks from the Digital Content Creation and Gaming class because of the workloads in these groups test GPUs extensively. On the other hand, we also profile web-based applications as benignware. In July 2017, a survey from Netcraft [Netcraft, 2017] received responses from 1,767,964,429 sites from 6,593,508 web-facing computers. The study highlights the large consumer-base of web users. According to a recent survey from PyCharm [PyCharm, 2016], 38% of web developers, use Python for web development. Another survey from PyCharm [PyCharm, 2016] shows Django is most popular web framework used by Python developers. The numbers motivate us to use the services provided by Django as our benignware. To explore further, we also include services from Tornado in our setup. Additionally, we add a few features from SQLite to profile SQL-based applications. Table 2.5 shows some of the services that we profile on the HPCs. Web-Based applications commonly use one or more of these services.

chameleon	chaos	crypto pyaes	django template	regex effbot	dulwich log	unpack sequence
fannkuch	float	genshi	go	regex v8	hexiom	unpickle
hg startup	html5lib	json dumps	json loads	r ichards	logging	unpickle list
mako	meteor contest	nbody	nqueens	scimark	pathlib	unpickle pure python
pickle	pickle dict	pickle ist	pidigits	spambayes	pyflate	xml tree
python startup	python startup no site	raytrace	regex compile	spectral norm	regex dna	tornado http
sqlalchemy declara-tive	sqlalchemy impera-tive	sqlite synth	sympy	telco		

Table 2.5: Django, Tornado and SQLite services used as benignware

For malware samples, we used the VirusTotal [Total, 2012] database to search for all the malware with the following tags:

- **File OS** - Win32
- **File Type** - Win EXE
- **Machine Type** - Intel 386 or later, and compatibles
- **Subsystem** - Windows CMD

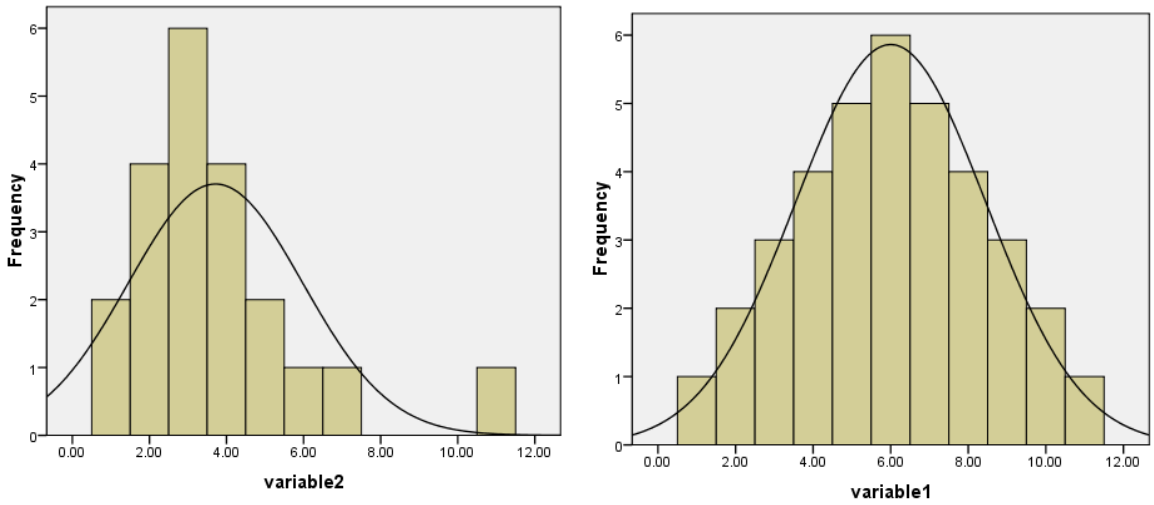
These tags are important to make sure that we can run the malware samples via a command prompt on a 32-bit Windows 7 machine. The use of command prompt to execute an application aids in automating the profiling of the malware samples. Thus, we do not consider any malware samples whose subsystem is 'Windows GUI.' Our samples include a mix of malware such as worm, virus, and rootkits.

2.4 Feature Extraction

Feature extraction aims at extracting informative, non-redundant, and low-dimensional features from the HPCs. Each HPC-based profile is a multidimensional time series of the hardware events monitored on the HPCs. The raw samples collected from the HPCs cannot be used directly to train on the ML classifiers because the execution time being different for any application results in a different number of samples recorded across such applications. The inconsistency in the number of samples recorded by the HPCs will lead to each profile in our dataset to have a distinct dimension. Only an equidimensional dataset will facilitate comparison amongst the profiles. It thus becomes a hard requirement for each sample in the dataset to be equidimensional. To achieve an equidimensional dataset, we aggregate the raw samples from the HPC's into 32 binned-histograms and then normalize the bins. Each bin is an aggregation of the number of samples recorded in a given time interval. The time interval is determined by the number of bins. Thus, histogram gives the probability distribution of the occurrence of an event across the application's entire execution time. Moreover, the aggregation of raw samples decreases the sparsity of our dataset. The sparsity of a matrix is the ratio of the number of zero elements in the matrix to non-zero elements [Golub and Van Loan, 2012]. Depending on the data set, sparsity may reduce the information content of the dataset. In our case, a sparse data set comprising of the profiles from the HPCs indicates low or no occurrence of the corresponding event. A low count of a hardware event across the data set reduces the discriminating power of that event. Aggregating the samples into histograms increases the bin value of the bins by decreasing the number of zero elements in the dataset.

Additionally, during the execution of an application, the application behavior can differ substantially. As a result, the profile of each application shows inconsistent vari-

ation across the execution profile. For examples, assume that we measure L1-Data Cache Hits to profile an application. During the entire execution of the application, the number of L1-Data Hits can vary extensively based on the number of memory accesses made by that application. The inconsistent variation in the profile makes our data non-normal. In such situations, the ML classifiers with Gaussian kernel functions (such as SVMs and Gaussian Process) [Gärtner, 2003] show a bias towards features with more variation than the features with least variation. Gaussian kernels measure the Euclidean distance between the features in the dataset [Vert et al., 2004]. Intuitively, a strong statistical analysis of a data favors a perfect bell-shaped curve (normal distribution). Figure 2·1 shows a non-normal curve and a normal curve. To transform our dataset from non-normal distribution to a more approximate normal distribution, we use Power Transform.



(a) (b)

Figure 2·1: Variation in dataset: (a) A non-normal distribution #1; and (b) Normal (Gaussian) Distribution #2. [University of Minnesota, 2017]

2.5 Feature Selection

HPCs on both Intel and AMD architectures provide over 200 architectural events. The ideal number of events to monitor simultaneously on the HPCs is equal to the number of counters present on each core in the architecture. As mentioned in section 1.4, we can monitor any number of events on the HPCs while the PMU will sample these events in a round-robin manner. Multiplexing multiple events will reduce the sampling frequency of the profiler, which affects the prediction accuracy of the ML classifiers. Subsequently, we monitor six events simultaneously on our AMD machines and four events on the Intel machines.

While training the ML classifiers, we consider each bin in the profile as a feature. The facility to monitor a large number of architectural events increases the number of potential features that can be used to train the ML classifiers. On the Intel and AMD machines, we can monitor 600 and 200 events respectively. On the other hand, the number of counters in each architecture usually ranges from four (like in Intel's Nehalem microarchitecture) to eight (like in Intel's Haswell microarchitecture and AMD's Bulldozer). We thus need a feature selection algorithm to select the best events from the pool of all the available hardware performance events. Note that, any number of events can be extracted using our feature selection algorithm.

Additionally, if each bin is considered a feature, then each event contributes 32 features in the dataset (since each event is a vector of 32 binned histogram). As a result, the total number of features in the dataset will be equal to 128/192/256 for $4, 6$ and 8 events monitored simultaneously. A dataset with a large number of features and low training examples may cause the problem of over-fitting [Hawkins, 2004] and train on the ML classifiers inefficiently. Secondly, the storage cost of the

data set during the training phase increases with the increase in number of features. To estimate the memory requirements by our implementation, we observed that our file was somewhere around 512 KB in size. In this scenario, each file stores traces for an event monitored on the HPC for an application executing for a minute. An application that is profiled using 8 events then requires 4 MB of memory. If we profile 1000 application on a system, the profile will take 4GB of space. This cost scales based on the number of applications in the dataset. The problems mentioned above while using a large number of features calls for decomposition of the data set using Principal Component Analysis (PCA). PCA yields the following advantages:

- Dimensionality reduction makes the dataset easier to comprehend by researchers/ users
- Shorter training times on complex classifiers such as SVM with RBF kernel
- Dimensionality reduction avoids the curse of dimensionality [Köppen, 2000]
- Enhanced generalization by reducing over fitting of the samples while training the ML classifiers

As mentioned in the section 1.4, PCA divides a multivariate dataset into components that are linearly uncorrelated. To choose the top events, we first run a small dataset of 10 benchmarks on both Intel and AMD machines for all the events possible. We discard events that yield no samples for one or more applications in our dataset. Next, we create datasets comprising of the profiles of each event. We apply PCA on the datasets of the events, individually. The results from the PCA yields the percentage of variance explained by each component. Intuitively, we want high variance ratio explained by each component. To this end, we sort the events in the order of the variance explained by each component. Our best events show a total of more

than 99% variance in the first two components itself. To this end, we chose the top four events for the Intel machine and six events for the AMD machines. The number of events chosen is in correspondence to the number of physical counters available on each core. For training our ML classifiers, we use the eigenvalues of the first two principal components of each of the chosen best events to obtain the transformed data points as our features. As a result, we have reduced a 128/192/256 dimensional to an 8/12-dimensional dataset on Intel and AMD machines respectively. Additionally, PCA minimizes the redundancy in our dataset.

2.6 Machine Learning Classifiers

In our implementation, we explore twelve classifiers to estimate their potential for malware identification. The classifiers are trained to behave as binary classifiers. The two classes are Malware and Benignware. We start training our dataset with linear algorithms like KNN and SVM(Linear Kernel). Additionally, we also use complex classifiers like SVMs (with Poly and RBF kernels), Decision Trees and Random Forests, MLP, AdaBoost and Logistic Regression. Table 2.6 gives a complete list of the classifiers used in our experimental setup. We use scikit-learn [Pedregosa et al., 2011] machine learning tools to train our classifiers. We follow use the 70-30% split to train our classifiers, i.e., 70% of the samples for training and the remaining 30% for testing. We cross-validate our results using k-fold cross-validation [Refaeilzadeh et al., 2009] technique with $k = 10$.

K Nearest Neighbors	Decision Trees	Random Forest	Naive Bayes
SVM Linear	SVM Poly	SVM RBF	SVM Sigmoid
Gaussian Process	Logistic Regression	AdaBoost	MultiLayer Perceptron

Table 2.6: Supervised Machine Learning Classifiers

Chapter 3

Evaluation

3.1 Experimental Setup

The end goal of this project is to test the capabilities of ML classifiers in differentiating between benign and malicious HPC based profiles. We setup our experiments on Intel and AMD-based processor. For our Intel machine, we use the Intel I7-2600 processor which belongs to the Sandy Bridge microarchitecture and contain four physical counters per core. Additionally, the Intel machine had 4GB of RAM, private L1/L2-cache, and a shared L3-cache. For the AMD machine, we use the AMD FX-8150 processor which belongs to the Bulldozer family of processors. A bulldozer microarchitecture has six physical counters per core. Each machine had 8GB of RAM, a private L1-cache and shared L2-cache. Our experimental setup uses 32-bit Windows 7 OS. The main motivation for using Windows is the existence of a large dataset of malware developed over the years.

The presence of other processes can affect the profiles collected by the HPCs since all the processes share the same hardware resources of the processor. We make sure that do not execute any other application in parallel to the application needed for profiling. Therefore during profiling, only the processes spawned by the application, the profiler and the OS are scheduled. Further, we run each application for 32 times on the same machine. Here, the motivation is to analyze the reproducibility of an HPCs based profile.

We ensure that the malware samples are successful in executing their malicious behavior by taking the following steps - we turned off both the Windows firewall settings and the Windows Defender[1]. Additionally, we did not have any third-party AV software installed on our system. To select top malware samples, we ran a preliminary test on the AMD machine, with randomly downloaded 1000 malware samples. We then time the execution of each malware sample and create a subset of all the malware samples that terminate within a minute of execution. We profile this subset of malware samples on the HPCs using AMDs CodeAnalyst profiler. AMDs CodeAnalyst features an option of monitoring %CPU utilization and %Memory Utilization while executing a certain application. The motivation was that irrespective of the malware behavior, it is bound to use some CPU resources and some memory (at the least, the instruction memory). We ranked the Malware in the decreasing order of %CPU Utilization and %Memory Utilization in the first minute of the malware execution.

Potentially a malware can entirely or partially compromise the OS it is running on. To prevent profile creation on a faulty OS, we restore a clean copy of the OS after every single execution of malware. We divide our hard disk into two partitions; one partition has Windows 7 running all the benignware and malware. The second partition has Ubuntu 16.04 LTS running on it. The Linux partition stores a clean image of the Windows partition. After every single run of malware, on the Windows partition, the Linux partition restores the affected Window's partition with the original clean image. To automate the above process, we modify the Linux GRUB bootloader to alternate booting into the two partitions on every restart. A restart signal is sent to the Windows operating system after the execution of the malware completes or after a minute, whichever happens first. As a result, the HPCs definitely monitor complete

[1]We do not include any malware sample that required an active Internet connection.

or partial malware activity in the first minute of execution. After booting into the Linux OS, a startup script restores the fresh copy of the Windows image into the partition and then sends the power off signal with the restart option. The GRUB bootloader will now boot into the Windows partition. Figure 3·1 provides the cycle followed by the experimental setup to profile malware using HPCs.

Figure 3·1: Executing and Profiling Malware using Savitor and Intel's VTunes

The implementation stated in the above two paragraphs, i.e., running each application sequentially, and restoring a clean copy of the Windows while executing malware - causes a substantial increase in the total time to profile all applications in our dataset. The total time to profile a single run of malware takes $\approx$ 6 minutes to profile. To parallelize the profiling of the applications we create a cluster of the Intel and AMD machines with each machine in the cluster having the exact same hardware and software specifications. The Intel cluster has 12 nodes while the AMD cluster

has 21 nodes. Note that the number of nodes does not hold any importance in our implementation. Each cluster has a master node that is responsible for scheduling the profiling task on the slave nodes. The master node uses RabbitMQ [Videla and Williams, 2012] for scheduling the jobs to the slave nodes. RabbitMQ is an open-source message broker that can schedule tasks from a master entity to the available clients. Each slave node, after booting into the Windows partition informs the master of its availability to profile an application. The master node then responds to the slave with the next application to be scheduled.

On the AMD machines, we use Savitor to profile the applications. We set the sampling frequency to 1000 Hz, and monitor six events simultaneously. On the Intel machines, we use VTunes. On Windows 7 32-bit version, VTunes only provides command line interface for measuring hardware performance events. In our experiments, we used *runsa* mode in VTunes, which records all the samples in the configured frequency. Even on the Intel machines, we set the sampling rate to 1000 Hz. After sampling the HPCs, we use VTunes report to extract the data into 32 bins with identical time intervals during the one-minute experiment. The profiles from the HPCs are compressed using the Principal Component Analysis to eight components on the Intel machine and 12 components on the AMD machine. Each event contributes two [1] features in our dataset, and the Intel's SandyBridge and AMD's Bulldozer architecture have four and six physical counters respectively. Table 3.1 summarizes the experimental setup described in this section.

[1]Please refer section 3.2 to understand why each event contributes two components.

Specifications	**AMD Setup**	**Intel Setup**
Processor	AMD FX-8150	Intel I7-2600
Microarichitecture	Bulldozer (Family 15h)	Sandy-Bridge
RAM Size	8GB	4GB
OS	Windows 7 32-Bit	Windows 7 32-Bit
Partition Loader OS	Ubuntu 16.04 LTS	Ubuntu 16.04 LTS
#HP Counters	6	4
Profiler	Savitor	Intel VTunes
Sampling Frequency	1000Hz	1000Hz
Feature Size (Post-PCA)	12	8
Cluster Size	21	15

Table 3.1: Experimental Setup - Specifications

3.2 Results after Feature Selection using PCA

Table 3.2 and 3.3 show the top four and six events selected for the Intel and AMD experimental setups. These events show more than 99% of variance in their top two components.

Event Code	Event Description
0x04000	The number of accesses to the data cache for load and store references
0x03000	The number of CLFLUSH instructions executed
0x02B00	Counts the number of SMIs received
0x02904	Counts the number of Load operations dispatched to the Load-Store unit
0x02902	Counts the number of Store operations dispatched to the Load-Store unit
0x02700	The number of CPUID instructions retired

Table 3.2: List of Top 6 AMD Events Code & Description

Event Code	Event Description
L2 LINES OUT DEMAND DIRTY	Dirty L2 cache lines evicted by demand
LOCK CYCLES SPLIT LOCK UC LOCK DURATION	Cycles in which the L1D and L2 are locked, due to a UC lock or split lock
FP COMP OPS EXE SSE PACKED SINGLE	Counts number of SSE single precision Floating Point scalar uops executed
L2 LINES OUT PF DIRTY	Dirty L2 cache lines evicted by L2 prefetch

Table 3.3: List of Top 4 Intel Events Code & Description

To analyze the result of PCA, we can either use a Scree Plot or a Cumulative Variation plot. The ideal pattern in a scree plot is a steep fall curve, followed by

a bend and then a flat or horizontal line. We retain those components in the steep curve before the first point that starts the flat line trend. Another way of analyzing the same data is using the Cumulative Variation plot. The cumulative variation plot shows the proportion of variance explained by each component. Thus, from the Cumulative Variation plots, we select components that cumulatively explain a certain percentage of variation. Refer figure 3·2 and 3·3. In these plots, the blue bar plots the cumulative variation plot and the red dotted line shows the scree plot. We observe that the events from table 3.2 and 3.3 require two components to explain a cumulative variance of more than 99%.

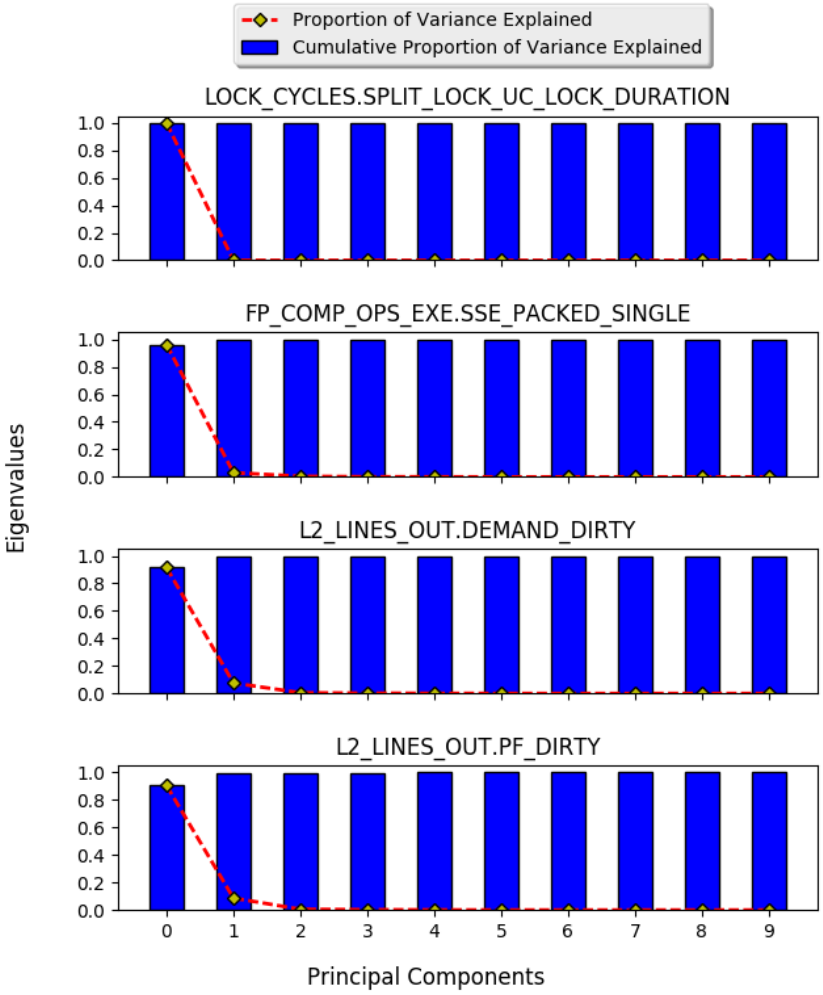


Figure 3·2: Scree and Cumulative Variation Plot for Intel's Top 4 events

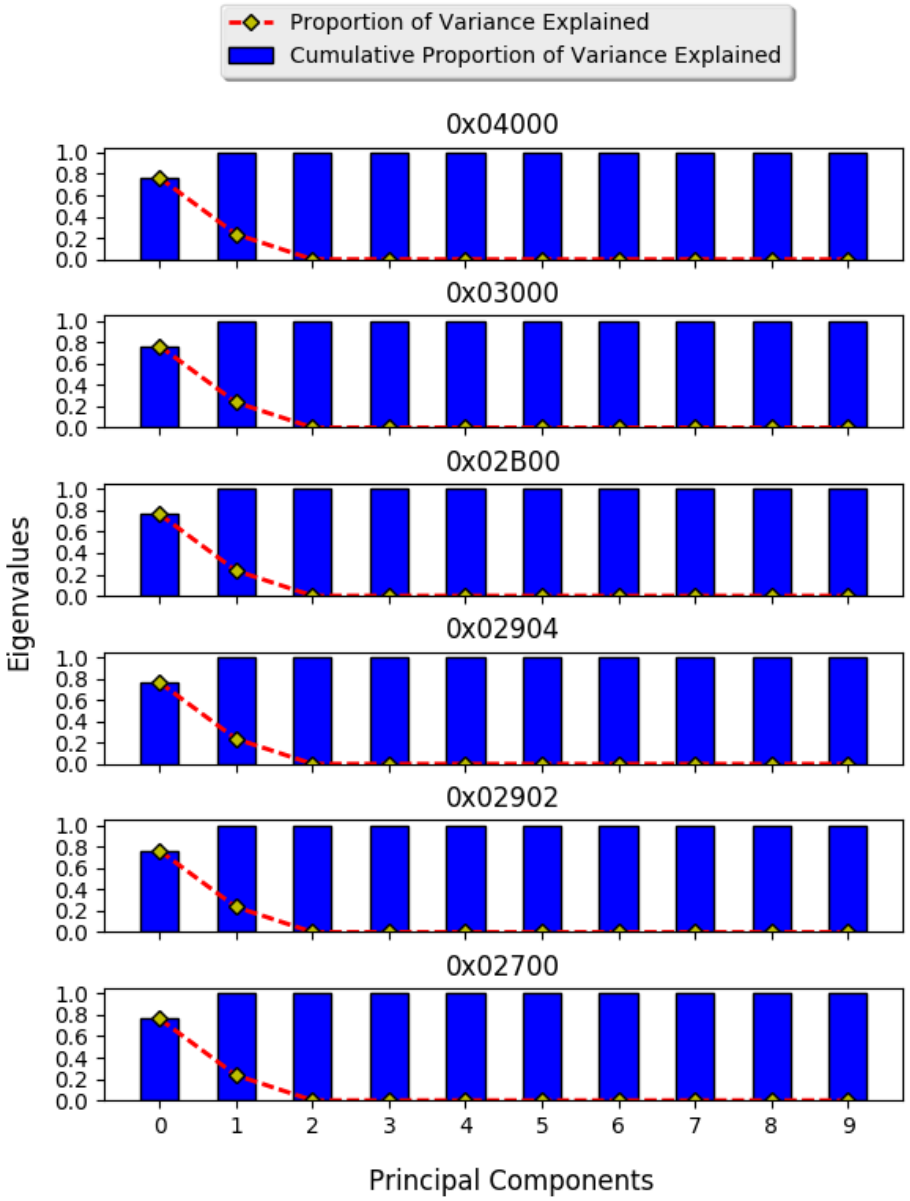


Figure 3·3: Scree and Cumulative Variation Plot for AMD's Top 6 events

3.3 Metrics to measure Classification Accuracies

We measure the performance of our classifiers using the True Positive (TP), False Positive (FP), True Negative (TN) and False Negative (FN) parameters. To define each parameter, assume that we have a Binary classifier with classes A and B. With respect to class A,

- **True Positive**: When a testing condition correctly predicts class A
- **False Positive**: When a testing condition predicts class B but the sample is class A
- **True Negative**: When a testing condition correctly predicts class B
- **False Negative**: When a testing condition predicts class A but the sample is class B

Based on the above parameters, the metrics used to analyze the ability of a classifier to predict the classes successfully are defined as follows:

- **Precision (P)**: Is the ratio of number of positive testing samples classified into a class to the total number of samples classified to the class, i.e., $P = \frac{TP}{TP+FP}$
- **Recall (R)**: Is the ratio of number of positive testing samples classified into a class to the total number of samples belonging to that class, i.e., $R = \frac{TP}{TP+FN}$
- **F2-Score**: Weighted Average of Precision and Recall. Gives the prediction accuracy in classifying testing samples into a class.
- **Support**: Total number of occurrence of each class in the testing dataset

For an ideal binary classifier, the precision, recall (also called sensitivity) and F2-Score is 1.0 implying the perfect true positive rate. An ideal classifier classifies

all the samples into their respective classes. On the contrary, recall and precision value decreases when the false negative or false positive rate increases, i.e., when the classifier identifies the testing sample into the wrong class.

Additionally, we plot the Receiver Operating Characteristics (ROC) curve for each classifier. A ROC curve plots the true positive rate versus the false positive rate. Mathematically speaking, the false positive rate is $(1 - \text{recall})$ [Wikipedia, 2017b]. A model that is able to fit the data points in the classifier accurately lies in the top left corner of the ROC curve [Sokolova et al., 2006]. Such models show true positive rate of ≈ 1.0 and false positive rate of ≈ 0.0. As the curve comes closer to $45°$, the less accurate the machine learning model becomes. Additionally, the area under the curve is a measure of prediction capability of the classifier, i.e., how well the test separates the two classes.

3.4 Classification Results - AMD

Table 3.4 shows the four metrics to measure the classification accuracies for each classifier while classifying the malware and benignware samples in our testing data set. Figure 3·4 plots the precision and recall value for each classifier. For benignware classification, all classifiers yield precision & recall >0.80 and >0.80 respectively. As a result, the F2-Score for benignware is >0.90 for all the classifiers. For malware classification, all the classifiers yield high precision of >0.75 except Naive Bayes. High precision shows success in classifying malware and benignware samples. However, the classifiers also produce recall values ranging from 0.74 to as low as 0.10. Low recall value shows that classifiers label malicious sample as benignware.

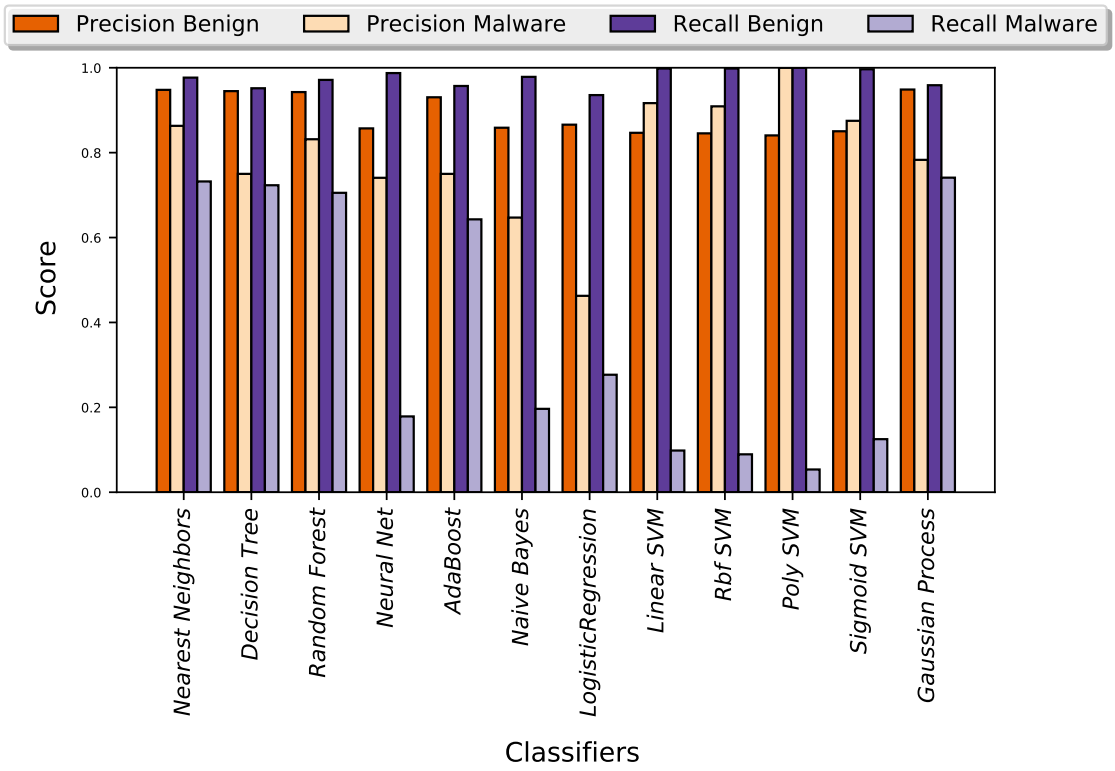


Figure 3·4: Precision and Recall Scores for classification of benignware and Malware Classes for the AMD experimental setup.

Classifier	Class	Precision	Recall	F2-Score	Support
KNN	Benignware	0.95	0.98	0.96	559
	Malware	0.86	0.73	0.79	112
	Average	0.93	0.94	0.93	671
DECISION TREE	Benignware	0.94	0.95	0.95	559
	Malware	0.75	0.72	0.74	112
	Average	0.91	0.91	0.91	671
RANDOM FOREST	Benignware	0.94	0.97	0.96	559
	Malware	0.83	0.71	0.76	112
	Average	0.92	0.93	0.92	671
NEURAL NETWORK	Benignware	0.86	0.99	0.92	559
	Malware	0.74	0.18	0.29	112
	Average	0.84	0.85	0.81	671
ADABOOST	Benignware	0.93	0.93	0.94	559
	Malware	0.75	0.64	0.69	112
	Average	0.90	0.90	0.90	671
NAIVE BAYES	Benignware	0.86	0.98	0.91	559
	Malware	0.65	0.28	0.35	112
	Average	0.80	0.83	0.81	671
LOGISTIC REGRESSION	Benginware	0.88	0.99	0.93	559
	Malware	0.79	0.12	0.21	63
	Average	0.86	0.87	0.83	671
LINEAR SVM	Benginware	0.85	1	0.92	559
	Malware	0.92	0.10	0.18	112
	Average	0.86	0.85	0.79	671
POLY SVM	Benginware	0.84	1.00	0.91	559
	Malware	1	0.15	0.10	112
	Average	0.87	0.84	0.78	671
RBF SVM	Benginware	0.85	1.00	0.92	559
	Malware	0.91	0.19	0.16	112
	Average	0.86	0.85	0.79	671
Sigmoid SVM	Benginware	0.85	1.00	0.92	559
	Malware	0.881	0.12	0.22	112
	Average	0.85	0.85	0.80	671
Gaussian Process	Benignware	0.95	0.96	0.95	559
	Malware	0.78	0.74	0.76	112
	Average	0.92	0.92	0.92	671

Table 3.4: Classifier results in terms of Precision, Recall, F2-Score and Support for each class on the AMD machine. [F2-Score is weighted harmonic mean for each class, whereas the Average is weighted mean of Precision, Recall, and F2-Score values for benignware and the malware classes individually.]

To explain why we get such results, refer figure 3·5. We plot a two-dimensional

representation of the original eight-dimensional data space from the AMD setup. Additionally, it also shows the training and testing samples extracted randomly using 70% and 30% split. The plot is used just for the visualization of the data space and to explain the observed classification results. For decomposing the twelve-dimensional dataset, we use scikit's implementation of Manifold decomposition using multidimensional scaling [Kruskal, 1964]. In the plot, the blue triangle represents the benignware samples, whereas the orange asterisk represents the malware samples. The data space comprises of benignware samples spread across whereas the malware samples accumulate in a small cluster. The clustering of malware samples is because of the events monitored on the HPCs while profiling the malware samples.

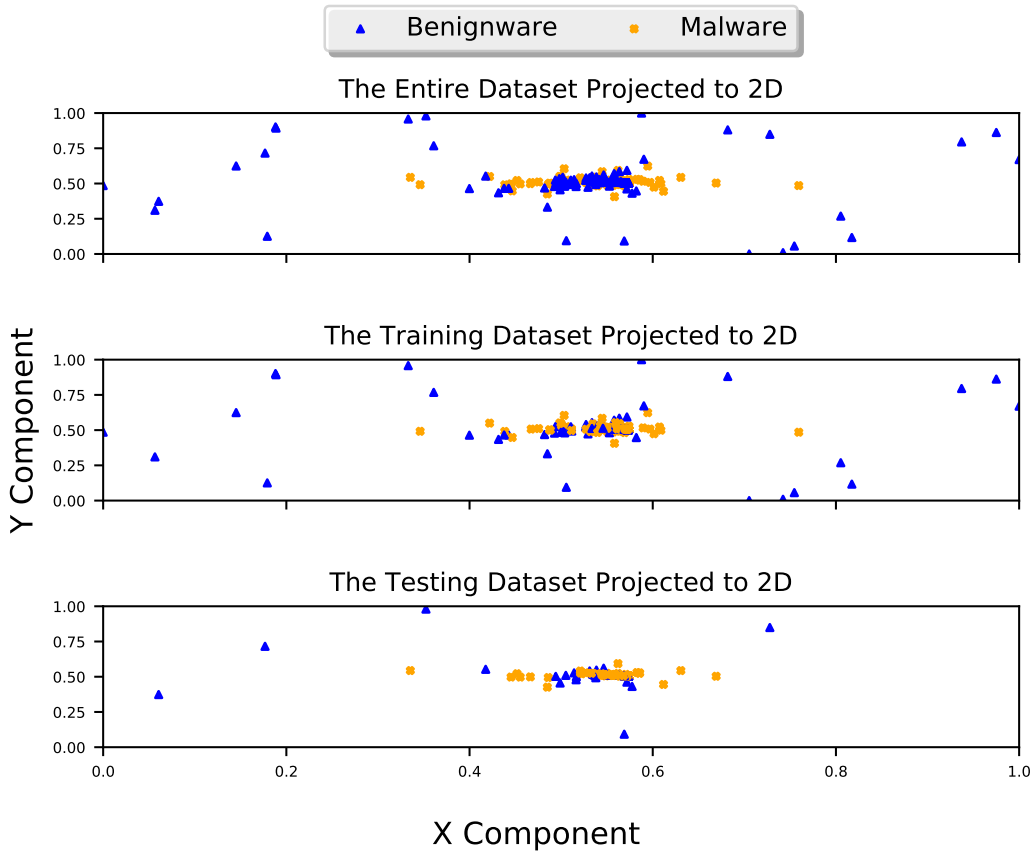


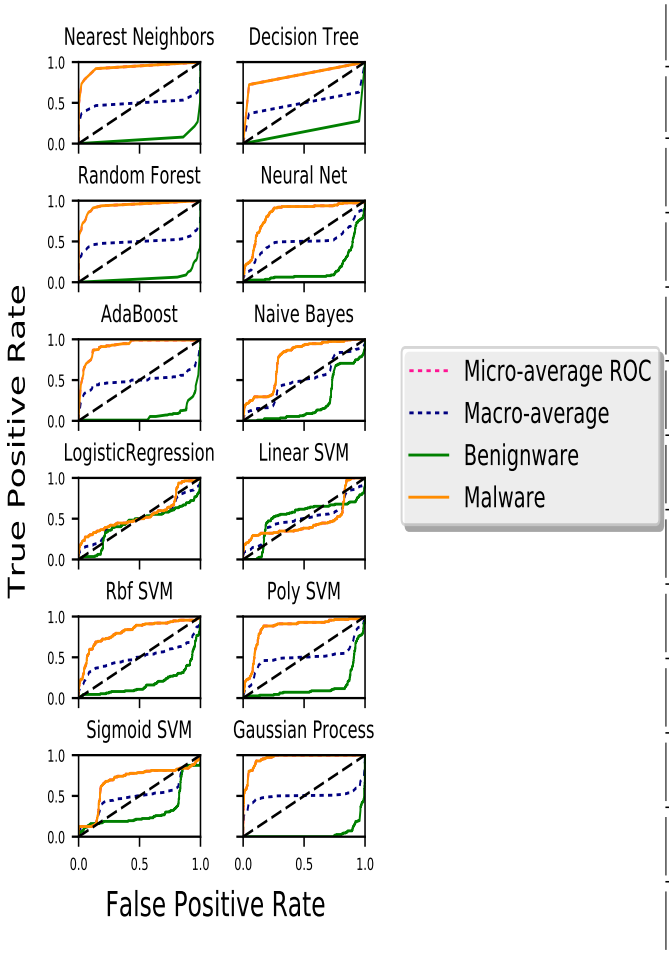
Figure 3·5: 2D representation of the entire Data space, the training data space and the testing data space.

Refer table 3.2 for the top events measured on the AMD setup. Three of the six events measure the number of load and stores references dispatched to the load-store unit. The next two instructions track the number of CLFLUSH and CPUID instructions executed by a program. A program can flush a cache line from the L2-cache using the CLFLUSH instruction. The CPUID instruction provides details about the hardware to an user-level application. Two other event measure the number of System Management Interrupts(SMI) issued to the kernel. SMI interrupts to enter the system into the System Management Mode (SMM). SMM is a part of the firmware used to debug the hardware on the processor's motherboard [Favor and Weber, 2000]. Embleton [Embleton et al., 2013] demonstrate how they deployed rootkits in the SMM

code. The clustering of malware samples may occur due to multiple reasons. Malware samples may regularly issue CLFUSH instructions to evict a cache line [Yarom and Falkner, 2014] for malicious purpose. Secondly, caching enables a program to minimizes the number of references to the external memory. Moreover, a malware sample that resides in the system's memory also tries to reduces the number of accesses to external memory to evade detection [Ligh et al., 2014]. The profile of such malware samples may see an overlap in the number of load-store references. Thirdly, there is no alternative to enter the SMM, except by issuing an SMI instruction. The profiles from the HPCs depict no inherent difference amongst the malware samples. On the contrary, the benignware samples in our data set perform a myriad of tasks resulting in different memory usage. The inconsistent memory utilization of the benignware yields a distinct number of load-store references, CLFLUSH instructions or request for hardware information via the CPUID instruction. As a result, the benignware samples spread across the data space. The clustering of malware samples and the widespread benignware samples in the data-space cause mispredictions while classifying malware samples. The classifiers may tag all the benignware sample that lie close to the cluster, as malware samples or the malware samples as benignware.

Further in figure 3·6 we observe that simple linear classifiers such as Linear SVMs have an area under the ROC curve of $\approx$ 70%for the malware class. As a result, an increasing true positive rate will almost linearly increase the false positive rate. As a result, Linear SVM was unsuccessful in efficiently fitting our data set. Additionally, for complex classifiers such as Decision Trees, Random Forests, AdaBoost Classifier and Neural Network (using MultiLayer Perceptron) the ROC curve is >80% implying small variation in false positive rate on increasing the true positive rate. Such classifiers with high area under the curve indicate a well-trained model on our data-set.

3·6(b), shows the prediction accuracy of K-Fold validation technique with $k = 10$ and the standard deviation (SD) of the results. The prediction accuracy of all the classifiers match the cross-validation results. Additionally, the cross-validation shows an SD of < 0.1, implying no variation in the prediction score in 10-fold Cross-validation.



(a)

Classifier	AUC	CV Score
KNN	0.94	0.93 (+/-0.05)
Decision Tree	0.84	0.91 (+/- 0.06)
Random Forest	0.94	0.92 (+/- 0.06)
MLP	0.84	0.84 (+/- 0.03
AdaBoost	0.92	0.88 (+/- 0.05)
Naive Bayes	0.76	0.84 (+/- 0.05)
Logistic Reg	0.56	0.50 (+/- 0.07)
Linear SVM	0.48	0.51 (+/- 0.08)
RBF SVM	0.81	0.82 (+/- 0.04)
Poly SVM	0.86	0.86 (+/- 0.1)
Sigmoid SVM	0.68	0.65 (+/- 0.07)
Gaussian Process	0.96	0.94 (+/- 0.03)

(b)

Figure 3·6: a. ROC Curve for all the classifiers. **b.** ROC-Area Under Curve (AUC) and Cross-validation score for all the classifiers

3.5 Classification Results - Intel

For the Intel setup, we use the same strategy to explain our results as in section 3.4.Table 3.5 shows the four metrics to measure the classification accuracies for each classifier while classifying the malware and benignware samples in our testing data set. Note the precision and recall for both benignware and malware classes for each classifier. For benignware classification, all classifiers but Gaussian Naive Bayes yield precision & recall >0.85. As a result, the F2-Score for benignware is >0.90 for all the classifiers except for Naive Bayes and SVM with the linear kernel. For malware classification, all the classifiers yield high precision of more than ≈0.80 except for Naive Bayes and SVM Trees classifiers. However, the classifiers also produce recall

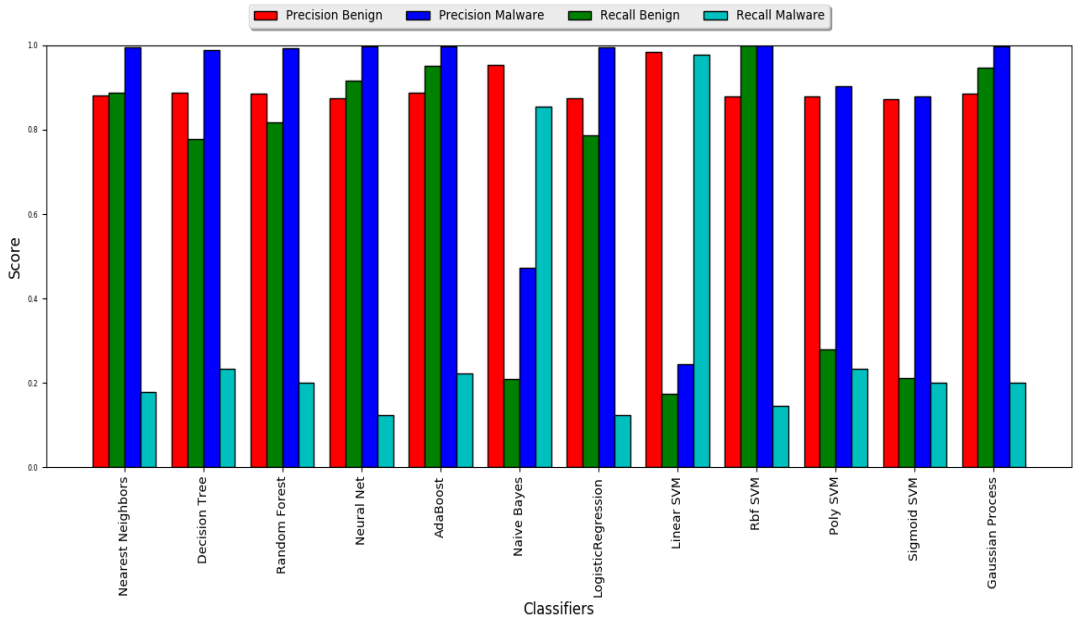


Figure 3·7: Precision and Recall Scores for classification of benignware and Malware Classes for the Intel's experimental setup.

Classifier	Class	Precision	Recall	F2-Score	Support
KNN	Benignware	0.88	1.0	0.94	556
	Malware	0.89	0.18	0.30	90
	Average	0.88	0.88	0.85	646
DECISION TREE	Benignware	0.89	0.99	0.94	556
	Malware	0.78	0.23	0.36	90
	Average	0.87	0.88	0.86	646
RANDOM FOREST	Benignware	0.88	0.99	0.94	556
	Malware	0.82	0.20	0.32	90
	Average	0.88	0.88	0.85	646
NEURAL NETWORK	Benignware	0.88	1.00	0.93	556
	Malware	0.92	0.12	0.22	90
	Average	0.88	0.88	0.83	646
ADABOOST	Benignware	0.89	1.00	0.94	556
	Malware	0.95	0.22	0.36	90
	Average	0.90	0.89	0.86	646
NAIVE BAYES	Benignware	0.95	0.47	0.63	556
	Malware	0.21	0.86	0.33	90
	Average	0.85	0.53	0.59	646
LOGISTIC REGRESSION	Benginware	0.88	0.99	0.93	556
	Malware	0.79	0.12	0.21	63
	Average	0.86	0.87	0.83	646
LINEAR SVM	Benginware	0.99	0.24	0.39	556
	Malware	0.17	0.98	0.29	90
	Average	0.87	0.35	0.38	646
POLY SVM	Benginware	0.88	0.90	0.89	556
	Malware	0.28	0.23	0.25	90
	Average	0.80	0.81	0.80	646
RBF SVM	Benginware	0.88	1.00	0.89	556
	Malware	1.00	0.14	0.25	90
	Average	0.90	0.88	0.84	646
Sigmoid SVM	Benginware	0.87	0.88	0.88	556
	Malware	0.21	0.20	0.21	90
	Average	0.78	0.78	0.78	646
Gaussian Process	Benignware	0.89	1.00	0.94	556
	Malware	0.95	0.20	0.33	90
	Average	0.89	0.89	0.85	646

Table 3.5: Classifier results in terms of Precision, Recall, F2-Score and Support for each class on the Intel machine

To explain why we get such results, refer figure 3·8. Similar to results on the AMD setup as seen in figure 3·5, the data space comprises of widespread benignware samples and clustered malware samples.

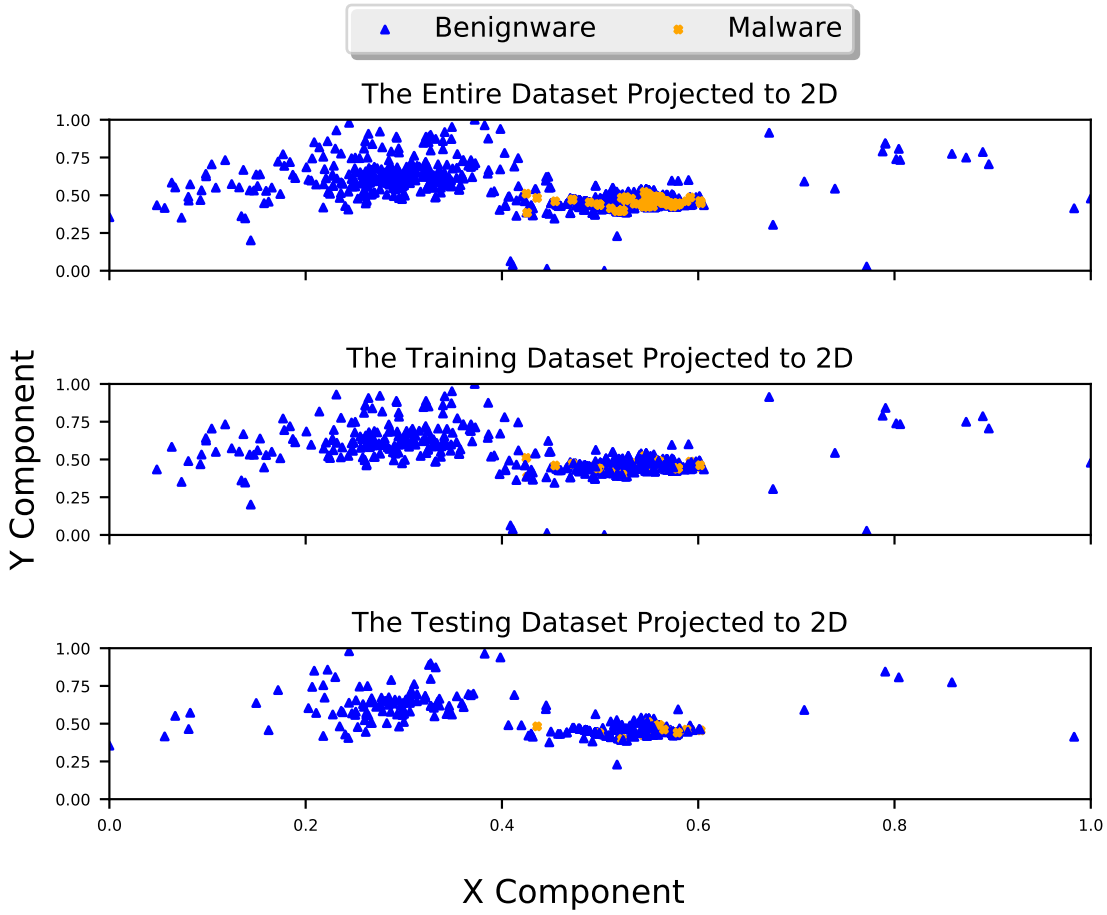
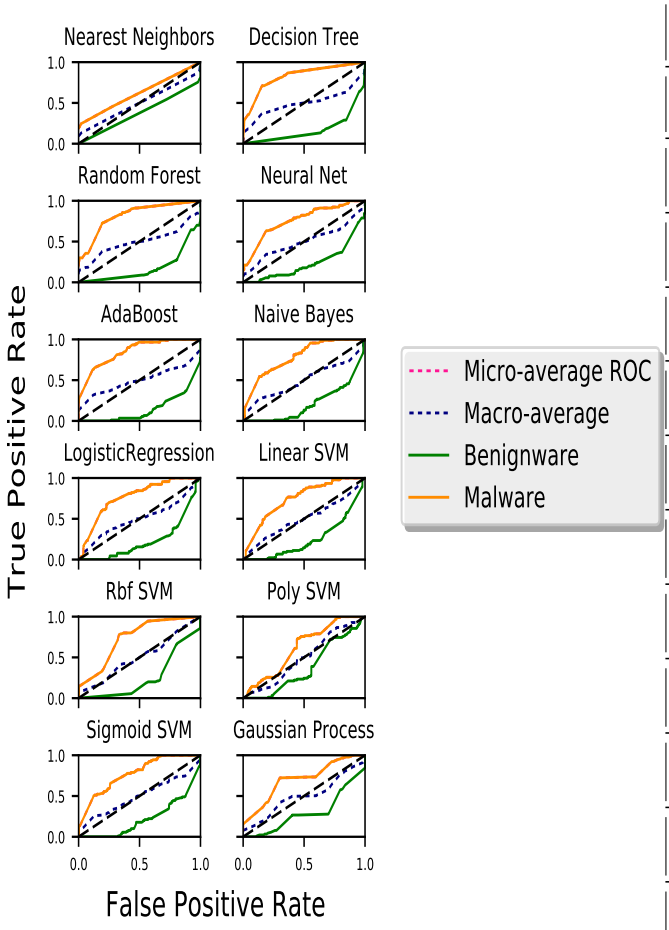


Figure 3·8: 2D representation of the entire Data space, the training data space and the testing data space.

Refer table 3.2 for the top events measured on the Intel setup. Two of the four events measure the number of times L2-cache line are evicted. One event measures the number of SSE single precision floating point instructions executed. The clustering occurs because the malware samples have a similar memory utilization or perform a similar number of floating point calculations. The similarity in memory access by the malware samples results in a similar profile for all such malware specimens. On the contrary, the benignware sample performs a myriad of tasks resulting in different memory usage and floating point tasks. The benignware samples, thus spread across the data space.

Further in figure 3·9, we plot the Receiver Operating Characteristics (ROC) curve for each classifier. From figure 3·9 we observe that simple linear classifiers like SVMs have an area under ROC curve $< 70\%$ for the malware class and complex classifiers such as Decision Trees and AdaBoost Classifier, the ROC curve is $> 80\%$. The results show that incapability of the linear classifiers to fit our dataset into a model and the need for complex classifiers.

3·9(b), shows the prediction accuracy of K-Fold validation technique with $k = 10$ and the standard deviation (SD) of the results. The cross-validation result shows an SD of < 0.1, implying no variation in the prediction score in 10-fold Cross-validation.



(a)

Classifier	AUC	CV Score
KNN	0.62	0.72 (+/-0.05)
Decision Tree	0.83	0.80 (+/- 0.04)
Random Forest	0.82	0.85 (+/- 0.01)
MLP	0.78	0.80 (+/- 0.03
AdaBoost	0.85	0.88 (+/- 0.05)
Naive Bayes	0.79	0.74 (+/- 0.02)
Logistic Reg	0.77	0.78 (+/- 0.08)
Linear SVM	0.75	0.69 (+/- 0.08)
RBF SVM	0.74	0.78 (+/- 0.04)
Poly SVM	0.62	0.66 (+/- 0.01)
Sigmoid SVM	0.78	0.78 (+/- 0.02)
Gaussian Process	0.70	0.80 (+/- 0.03)

(b)

Figure 3·9: a. ROC Curve for all the classifiers. **b.** ROC-Area Under Curve (AUC) and Cross-validation score for all the classifiers

Chapter 4

Conclusions

4.1 Summary of the thesis

In this thesis, we present an assessment to advice against the usage HPCs and ML classifiers to classify the benign and malware applications. We intend to stress the fact that, HPCs being low-level hardware components do not capture any behavioral semantics of a high-level program. In our assessment, we use Savitor and Intel's VTunes to profile application on HPCs. Our dataset contains applications that are commonly used real-life applications as benignware and pre-existing known malware. Moreover, we apply power transform our data to convert non-normally distributed samples into normally distributed samples. Then, we use PCA to select the top four events that best distinguishes across applications in our dataset. Finally, we train our dataset on twelve classifiers that include both simple linear classifiers to complex classifiers such as MultiLayer Perceptron. The prediction accuracy of an ML classifier to detect benignware is is more than 85% across all ML classifiers. Linear classifiers such as Linear SVMs and Naive Bayes were not able to fit a model around our dataset. This calls for more complex classification techniques. However, all the classifiers failed to flag known malicious application as malware. The results show that while testing the malware samples, the recall value of all the classifiers is a ≈20%. The results we show are on a dataset that has 83 malware samples and 64 benign samples. The failure to detect a small dataset, as such, begs us to consider using HPCs as malware detectors on a larger dataset. The failure to predict malware effectively forces us

to conclude that HPCs are incompetent in capturing the high-level semantics of an application.

4.2 Future Work

In this research, we evaluate the performance of twelve classifiers in their attempt to differentiate between benign and malicious software using the profiles from the HPCs. We base our results on Intel and AMD processors. Additionally, we plan to implement our system on Android-based Devices as well. Demme [Demme et al., 2013] reported a classification accuracy ranging from 100% to 25% across a wide variety of Android malware. They enlist samples that successfully evade their proposed systems calls for a more thorough analysis of the proposed system. Moreover, the proposed system of malware detection using HPCs are deployed to detect specific malware families such as rootkits. We should inform the security community of the downfalls of using HPCs for malware detection.

In our implementation, we used 83 malware samples and 63 benign samples. It is not clear how the classifiers would fare in case of a larger dataset of both malware and benign samples. Intuitively, a small data space such as ours, was dense enough for 80% of the malware samples to evade detection, then a larger data space will increase the false positive and false negative rate subs

Using profilers like Savitor and VTunes, we profile an application on a single core to create a time-series profile of the application. Modern processors support hyper-threading and applications leverage this feature to parallelize their payload. The currently, implemented systems will not yield a time-series profile of the application in case of a hyper-threaded application. A novel system needs to be evaluated to incorporate hyper-threading such that HPCs can still construe a time-series profile of an application.

In summary, we would like to evaluate the proposed system of malware detection using HPCs in the following scenarios:

- Implement and test the proposed system on an Android-based Device
- Evaluate the system on a larger data set
  - Say, 1000 samples of Benignware and 1000 samples of Malware
- Evaluate the system for Hyper-threaded processors

References

Akdemir, K., Dixon, M., Feghali, W., Fay, P., Gopal, V., Guilford, J., Ozturk, E., Wolrich, G., and Zohar, R. (2010). Breakthrough aes performance with intel aes new instructions. *White paper, June*. `https://www.intel.ua/content/dam/www/public/us/en/documents/white-papers/aes-breakthrough-performance-paper.pdf`.

ARM (2017a). Advanced risc machines. `https://www.arm.com/`.

ARM (2017b). Trustzone. `https://www.arm.com/products/security-on-arm/trustzone`.

Bahador, M. B., Abadi, M., and Tajoddin, A. (2014). Hpcmalhunter: Behavioral malware detection using hardware performance counters and singular value decomposition. In *4th International eConference on Computer and Knowledge Engineering (ICCKE), 2014*, pages 703–708. IEEE.

Box, G. E. P. and Cox, D. R. (1964). An analysis of transformations. *Journal of the Royal Statistical Society. Series B (Methodological)*, 26(2):211–252.

Bulpin, J. R. and Pratt, I. (2005). Hyper-threading aware process scheduling heuristics. In *USENIX Annual Technical Conference, General Track*, pages 399–402.

Center, M. D. (2017). Scheduling priorities. `https://msdn.microsoft.com/en-us/library/windows/desktop/ms685100(v=vs.85).aspx`.

Christodorescu, M. (2007). *Behavior-based Malware Detection*. PhD thesis, Madison, WI, USA. AAI3278879.

Contreras, G. and Martonosi, M. (2005). Power prediction for intel xscale/spl reg/processors using performance monitoring unit events. In *Proceedings of the 2005 International Symposium on Low Power Electronics and Design, 2005. ISLPED'05.*, pages 221–226. IEEE.

de Melo, A. C. (2010). The new linuxperftools. In *Slides from Linux Kongress*, volume 18. `http://vger.kernel.org/~acme/perf/lk2010-perf-acme.pdf`.

Demme, J., Maycock, M., Schmitz, J., Tang, A., Waksman, A., Sethumadhavan, S., and Stolfo, S. (2013). On the feasibility of online malware detection with performance counters. *SIGARCH Computer Architecture News*, 41(3):559–570.

Drongowski, P. J. AMD CodeAnalyst Team, Boston Design Center. (2008). An introduction to analysis and optimization with amd codeanalyst performance analyzer. *Advanced Micro Devices, Inc*, pages 1–20. `https://developer.amd.com/wordpress/media/2012/10/Introduction_to_CodeAnalyst.pdf`.

Embleton, S., Sparks, S., and Zou, C. C. (2013). Smm rootkit: a new breed of os independent malware. *Security and Communication Networks*, 6(12):1590–1605.

Favor, J. G. and Weber, F. D. (2000). Flexible implementation of a system management mode (smm) in a processor. US Patent 6,093,213.

Gärtner, T. (2003). A survey of kernels for structured data. *ACM SIGKDD Explorations Newsletter*, 5(1):49–58.

Golub, G. H. and Van Loan, C. F. (2012). *Matrix computations*, volume 3. JHU Press.

Griffin, K., Schneider, S., Hu, X., and Chiueh, T.-c. (2009). Automatic generation of string signatures for malware detection. In Kirda, E., Jha, S., and Balzarotti, D., editors, *12th International Symposium on Recent Advances in Intrusion Detection. RAID 2009, Saint-Malo, France, September 23-25, 2009. Proceedings*, volume 5758, pages 101–120. Springer Berlin Heidelberg.

Gulmezoglu, B., Zankl, A., Eisenbarth, T., and Sunar, B. (2017). Perfweb: How to violate web privacy with hardware performance events. *arXiv preprint arXiv:1705.04437*.

Harley, D. and Lee, A. (2007). Heuristic analysis–detecting unknown viruses. `https://www.welivesecurity.com/media_files/white-papers/Heuristic_Analysis.pdf`.

Hawkins, D. M. (2004). The problem of overfitting. *Journal of chemical information and computer sciences*, 44(1):1–12.

Instruments, N. (2014). Hardware product life cycle policies. `http://www.ni.com/life-cycle/hardware.htm`.

Intel (2011). Intel® 64 and IA-32 Architectures Software Developers Manual. *Volume 3B: System programming Guide, Part*, 2:18–1–19–220.

Intel (2017). Intel hardware-based security technologies for intelligent retail devices. `https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/security-technologies-4th-gen-core-retail-paper.pdf`.

Kirat, D., Vigna, G., and Kruegel, C. (2014). Barecloud: Bare-metal analysis-based evasive malware detection. In *USENIX Security Symposium*, pages 287–301.

Köppen, M. (2000). The curse of dimensionality. In *5th Online World Conference on Soft Computing in Industrial Applications (WSC5)*, pages 4–8.

Kruskal, J. B. (1964). Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis. *Psychometrika*, 29(1):1–27.

Ligh, M. H., Case, A., Levy, J., and Walters, A. (2014). *The art of memory forensics: detecting malware and threats in windows, linux, and Mac memory*. John Wiley & Sons.

May, J. M. (2001). Mpx: Software for multiplexing hardware performance counters in multithreaded programs. In *15th International Parallel and Distributed Processing Symposium IPDPS 2001: Proceedings. Los Alamitos, CA: IEEE Computer Society.*, pages 8 pp.–.

Metasploit (2017). Metasploit. `https://www.metasploit.com/`.

Mucci, P. J., Browne, S., Deane, C., and Ho, G. (1999). Papi: A portable interface to hardware performance counters. In *Proceedings of the department of defense HPCMP users group conference*. `http://www.icl.utk.edu/publications/papi-portable-interface-hardware-performance-counters`.

Netcraft (2017). July 2017 web server survey. `https://news.netcraft.com/archives/2017/07/20/july-2017-web-server-survey.html`.

NIEMEL, A. (2005). S. pcmark 05 pc performance analysis (white paper). *Futuremark Corporation*.

Nomani, J. and Szefer, J. (2015). Predicting program phases and defending against side-channel attacks using hardware performance counters. In *Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy*, page 9. ACM.

Patel, N., Sasan, A., and Homayoun, H. (2017). Analyzing hardware based malware detectors. In *Proceedings of the 54th Annual Design Automation Conference 2017*, page 25. ACM.

Pearson, K. (1901). Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11):559–572.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

PyCharm (2016). Python developers survey 2016: Findings. https://www.jetbrains.com/pycharm/python-developers-survey-2016/.

Qualcomm (2017). Qualcomm mobile security. https://www.qualcomm.com/products/features/security/mobile-security.

Rad, B. B., Masrom, M., and Ibrahim, S. (2011). Evolution of computer virus concealment and anti-virus techniques: a short survey. *arXiv preprint arXiv:1104.1070*.

Rao, C. R. (1964). The use and interpretation of principal component analysis in applied research. *Sankhyā: The Indian Journal of Statistics, Series A (1961-2002)*, 26(4):329–358.

Refaeilzadeh, P., Tang, L., and Liu, H. (2009). Cross-validation. In *Encyclopedia of database systems*, pages 532–538. Springer.

Reinders, J. (2005). Vtune performance analyzer essentials. http://nacad.ufrj.br/online/intel/vtune/Essentials_Excerpts.pdf.

Rijmen, V. and Daemen, J. (2001). Advanced encryption standard. *Proceedings of Federal Information Processing Standards Publications, National Institute of Standards and Technology*, pages 19–22.

Risks, M. (2011). Malware risks and mitigation report. bits-the financial services roundtable (2011). https://www.nist.gov/sites/default/files/documents/itl/BITS-Malware-Report-Jun2011.pdf.

Sakia, R. (1992). The box-cox transformation technique: A review. *Journal of the Royal Statistical Society. Series D (The Statistician)*, 41(2):169–178.

Scibilia, B. (2015). How could you benefit from a box-cox transformation? http://blog.minitab.com/blog/applying-statistics-in-quality-projects/how-could-you-benefit-from-a-box-cox-transformation.

Shlens, J. (2014). A tutorial on principal component analysis. *arXiv preprint arXiv:1404.1100*.

Singh, B., Evtyushkin, D., Elwell, J., Riley, R., and Cervesato, I. (2017). On the detection of kernel-level rootkits using hardware performance counters. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 483–493. ACM.

Sokolova, M., Japkowicz, N., and Szpakowicz, S. (2006). Beyond Accuracy, F-Score and ROC: A Family of Discriminant Measures for Performance Evaluation. In Sattar, A. and Kang, B.-h., editors, *AI 2006: Advances in Artificial Intelligence: 19th*

Australian Joint Conference on Artificial Intelligence, Hobart, Australia, December 4-8, 2006. Proceedings, pages 1015–1021. Springer Berlin Heidelberg.

Tang, A., Sethumadhavan, S., and Stolfo, S. J. (2014). Unsupervised anomaly-based malware detection using hardware features. In *International Workshop on Recent Advances in Intrusion Detection*, pages 109–129. Springer.

Tian, R., Islam, R., Batten, L., and Versteeg, S. (2010). Differentiating malware from cleanware using behavioural analysis. In *Proceedings of the 5th IEEE International Conference on Malicious and Unwanted Software, 2010*, pages 23–30.

Total, V. (2012). VirusTotal-Free online virus, malware and URL scanner. `https://www.virustotal.com/en`.

Uhsadel, L., Georges, A., and Verbauwhede, I. (2008). Exploiting hardware performance counters. In *5th Workshop on Fault Diagnosis and Tolerance in Cryptography, 2008*, pages 59–67. IEEE.

University of Minnesota (2017). Normal and non-normal distribution of data. `https://cyfar.org/inferential-analysis`.

Vert, J.-P., Tsuda, K., and Schölkopf, B. (2004). A primer on kernel methods. pages 35–70. Cambridge, MA: MIT Press.

Videla, A. and Williams, J. J. (2012). *RabbitMQ in action: distributed messaging for everyone*. Manning.

Vidhya, A. (2016). Practical guide to principal component analysis (pca) in r & python. `https://www.analyticsvidhya.com/blog/2016/03/practical-guide-principal-component-analysis-python/`.

Wang, X., Konstantinou, C., Maniatakos, M., and Karri, R. (2015). Confirm: Detecting firmware modifications in embedded systems using hardware performance counters. In *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 544–551. IEEE.

Weisberg, S. Yeo-johnson power transformations. `https://www.stat.umn.edu/arc/yjpower.pdf`.

Wikipedia (2017a). Power transform — wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Power_transform&oldid=785388432` [Online; accessed 12-September-2017].

Wikipedia (2017b). Precision and recall — wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Precision_and_recall&oldid=796650552` [Online; accessed 20-September-2017].

Wikipedia (2017c). Protection ring — wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Protection_ring&oldid=795376872[Online;accessed11-September-2017].

Witten, I. H., Frank, E., Trigg, L. E., Hall, M. A., Holmes, G., and Cunningham, S. J. (1999). Weka: Practical machine learning tools and techniques with java implementations. (Working paper 99/11). Hamilton, New Zealand: University of Waikato, Department of Computer Science.

Wold, S., Esbensen, K., and Geladi, P. (1987). Principal component analysis. *Chemometrics and intelligent laboratory systems*, 2(1-3):37–52.

Yarom, Y. and Falkner, K. (2014). Flush+ reload: A high resolution, low noise, l3 cache side-channel attack. In *USENIX Security Symposium*, pages 719–732.

Young, E. (2017). Openssl. https://www.openssl.org/.

Zhao, H., Xu, M., Zheng, N., Yao, J., and Ho, Q. (2010). Malicious executables classification based on behavioral factor analysis. In *International Conference on e-Education, e-Business, e-Management and e-Learning IC4E'10*, pages 502–506. IEEE.

Curriculum Vitae