Hardware Performance Counters Can Detect Malware: Myth or Fact?

Boyou Zhou, Anmol Gupta, Rasoul Jahanshahi, Manuel Egele, Ajay Joshi
{bobzhou,anmol.gupta1005,rasoulj,megele,joshi}@bu.edu
Eletrical and Computer Engineering Department, Boston University

ABSTRACT

The ever-increasing prevalence of malware has led to the explorations of various detection mechanisms. Several recent works propose to use Hardware Performance Counters (HPCs) values with machine learning classification models for malware detection. HPCs are hardware units that record low-level micro-architectural behavior, such as cache hits/misses, branch (mis)prediction, and load/store operations. However, this information does not reliably capture the nature of the application, i.e. whether it is benign or malicious. In this paper, we claim and experimentally support that using the micro-architectural level information obtained from HPCs cannot distinguish between benignware and malware. We evaluate the fidelity of malware detection using HPCs. We perform quantitative analysis using Principal Component Analysis (PCA) to systematically select micro-architectural events that have the most predictive powers. We then run 1,924 programs, 962 benignware and 962 malware, on our experimental setups. We achieve 83.39%, 84.84%, 83.59%, 75.01%, 78.75%, and 14.32% F1-score (a metric of detection rates) of Decision Tree (DT), Random Forest (RF), K Nearest Neighbors (KNN), Adaboost, Neural Net (NN), and Naive Bayes, respectively. We cross-validate our models 1,000 times to show the distributions of detection rates in various models. Our cross-validation analysis shows that many of the experiments produce low F1-scores. The F1-score of models in DT, RF, KNN, Adaboost, NN, and Naive Bayes is 80.22%, 81.29%, 80.22%, 70.32%, 35.66%, and 9.903%, respectively. To further highlight the incapability of malware detection using HPCs, we show that one benignware (Notepad++) infused with malware (ransomware) cannot be detected by HPC-based malware detection.

KEYWORDS

Malware Detection, Hardware Performance Counters, Machine Learning

ACM Reference Format:
Boyou Zhou, Anmol Gupta, Rasoul Jahanshahi, Manuel Egele, Ajay Joshi. 2018. Hardware Performance Counters Can Detect Malware: Myth or Fact?. In *ASIA CCS '18: 2018 ACM Asia Conference on Computer and Communications Security, June 4–8, 2018, Incheon, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3196494.3196515

ASIA CCS '18, June 4–8, 2018, Incheon, Republic of Korea

ACM ISBN 978-1-4503-5576-6/18/06…$15.00
https://doi.org/10.1145/3196494.3196515

1 INTRODUCTION

Distinguishing between malicious and benign software has remained one of the biggest challenges facing computer security over recent decades. As signature-based anti-virus scanners are easily thwarted by polymorphic malware, most commercial and academic anti-malware solutions rely on behavioral analysis. Behavioral analysis monitors programs as they execute, collects information on the process, and, upon a violation of a behavioral profile, classifies the program as malware. To this end, software-based behavioral analysis can draw from a wealth of semantically rich information sources, such as file names, registry keys, or network endpoints, which characterize the program's behavior. As software-level behavioral analysis performs malware detection at the cost of performance overhead, recent research proposes to reduce this performance overhead by leveraging Hardware Performance Counters (HPCs) to classify programs as benignware or malware.

HPCs are hardware units that count the occurrences of micro-architectural events such as instruction counts, hits/misses in various cache levels and branch (mis)predictions during runtime. Modern processors can capture more than 100 micro-architectural events, but a design-imposed strict limit of 4 (on Intel [1]) and 6 (on AMD [2]) counter registers dictates that HPCs can only monitor a small subset of these events at one time.

Under these constraints, previous works [3–6] leverage the measured HPC values to classify an unknown program as either benign or malicious. To this end, measured HPC values are sampled at a fixed frequency and the resulting data is aggregated into a time-series. Previous works record data of labeled programs in time-series, and use the HPC values in time-series to train various supervised machine learning models. The measured HPC values yield classifiers that can subsequently distinguish unknown programs as either benign or malicious.

The underlying assumption for previous HPC-based malware detectors is that *malicious behavior affects measured HPC values differently* than benign behavior. However, it is questionable, and in fact counter-intuitive, why the semantically high-level distinction between benign and malicious behavior would manifest itself in the micro-architectural events that are measured by HPCs. As a concrete example, consider that malware as well as benignware make use of the cryptographic APIs. While ransomware might maliciously encrypt the user data, the user might rely on encryption to safeguard privacy and data confidentiality. In both cases, ransomware and benignware, the program performs cryptographic operations. One cannot discriminate between malicious and benign

usage based on the measured HPC values. The semantic difference of whether the encryption was performed maliciously or not, exclusively depends on who holds the decryption keys, i.e., the attacker or the user. There is no indication that any HPC event would correlate with the ownership of the keys.

Given the substantial semantic difference between the high-level malicious behavior and the low-level micro-architectural events, it is expected from previous works that assert the utility of HPCs for malware detection to provide a rigorous analysis, interpretation, and justification of *why* the extracted features from measured HPC values identify the maliciousness of programs. This includes, for example, an analysis of the events found to be the most predictive of malicious behavior and a discussion of why these features capture behavioral information at all. Unfortunately, existing works elide any such discussions, and instead commit the logical fallacy of *"cum hoc ergo propter hoc"*[1] — or concluding causation from correlation. Moreover, the correlations and resulting detection capabilities reported by previous works frequently result from small sample sets and experimental setups that put the detection mechanism at an unrealistic advantage.

To shine a light on the feasibility of using HPCs for detecting malicious behavior, we survey the existing literature in this field, and identify common traits that exhibit impractical setups and misinterpretation of data analysis. Subsequently, we design, implement, and evaluate an experimental setup that allows us to reproduce previous works in this area, and compare these previous results with results obtained under more realistic scenarios.

In this work, we build an experimental setup close to the user environment, and evaluate fidelity of machine learning models. We run all experiments in a bare-metal environment instead of relying on virtualization techniques. This choice is motivated by two observations. First, our experiments indicate that measured HPC values collected for the same program running inside a virtual machine substantially differ from those collected on a bare metal system (comparisons in §2). Second, regular users likely execute programs directly on their systems outside of virtual machines. Further contributing to the realism of our experiments is the selection of training data for the machine learning models. Previous works [3, 5, 6] test their machine learning models using measured HPC values from the same programs used during training (In §4.3, we refer to this approach as **TTA1**). In a real-world deployment, this scenario would reflect a situation where all programs (benign and malicious) are known and labeled for training. In such situations, machine learning is unnecessary, as each program could be perfectly identified based on its hash. As Anti-Virus (AV) vendors report thousands of new malware samples every day, this scenario is highly unlikely to ever occur in reality. Thus, we test our models with measured HPC values from programs that have not been observed during training. This reflects a realistic scenario where, during training machine learning models, malware samples from the same category or family are available, but not the exact same malware that a user may encounter.

We train 6 different machine learning classifiers and compare the results obtained with both realistic and unrealistic approaches. Unsurprisingly, we observe that classifiers trained in the realistic scenario perform worse than those trained in an unrealistic scenario. To rigorously evaluate the performance of our classifiers, we perform 1,000 iterations of 10-fold cross-validations and consistently observe False Discovery Rate[2] of larger than 20%. Such high False Discovery Rates would disqualify HPC-based malware detectors from real-world deployments, as it would flag 264 programs in a default Windows 7 installation as malicious. Finally, we illustrate how fragile the resulting classifiers are by *simply* composing a benign program (Notepad++) with malicious functionality (ransomware). This straight-forward composition evades all our classifiers, even when they are trained with the benign and malicious components individually. In summary, this work makes the following contributions:

- We identify the prevalent unrealistic assumptions and the insufficient analysis used in prior works that leverage HPCs for malware detection (§2).
- We perform thorough experiments with a program count that exceeds prior works [3, 5–8] by a factor of $2\times \sim 3\times$, and the number of experiments in cross-validations that is 3 orders of magnitude more than previous works.
- We train and test dataset similar to what prior works have done, as well as, in a realistic setting where testing programs are not in the training programs. We compare the effects of this choice on the quality of the machine learning models (§ 5).
- Finally, to facilitate reproducibility, and enable future researchers to easily compare their experiments with ours, we make all code, data, and results of our project publicly available under an open-source license: https://github.com/bu-icsg/Hardware_Performance_Counters_Can_Detect_Malware_Myth_or_Fact

2 RELATED WORK AND MOTIVATION

Malware detection is the process of detecting malicious programs, for example, viruses. Many previous works commonly utilize *sub-semantic features* in malware detection [3, 5–11]. Ozsoy et al. defined the term *sub-semantic features* as "micro-architectural information about an executing program that does not require modeling or detecting program semantics" [9]. All these previous works have several drawbacks to various extent. We categorize the drawbacks that we observed into the following classes.

I Dynamic Binary Instrumentation (DBI)
II Virtual Machines (VMs)
III Division of Data By Traces (TTA1 in § 4.3)
IV No Cross-Validations or Insufficient Validations
V Few Data Samples

Besides HPCs, sub-semantic features can be extracted with dynamic binary instrumentation (DBI) tools such as Intel's Pin [4, 12], QEMU [13], Valgrind [14], or DynamoRIO [15]. Khasawneh et al. use Pin to monitor the instructions executed on virtual machines in their experimental setup [9–11]. Though DBI can extract sub-semantic features that are not available from HPCs, DBI introduces a substantial amount of performance overhead and is thus not suited to run in an *always-on*, online protection setting, which is

[1] "with this, therefore because of this"

[2] $F_+/(F_+ + T_+)$, where F_+ is number of benignware classified as malware and T_+ is number of malware classified as malware

Paper	Tool Choice	Experimental Setups		Event Choice	Data Division		Cross Validation				Machine Learning Models					# of Programs	Opensource	
	Drawback I: DBI (Pin or QEMU)	**Drawback II**: Virtual Machine	Bare-metal Machine	Quantative Selection of Events	**Drawback III**: Data Divided By Traces (TTA1 in § 4.3)	Data Divided By Samples (TTA2 in § 4.3)	**Drawback IV**: No Cross-validation	**Drawback IV**: 60 – 20 – 20% Data Division	10-fold Cross-validation	1,000 10-fold Cross-validations	DT	RF	KNN	NN	Ensemble Model (a collection of models)	**Drawback V**: Fewer than 1,000 programs	Release of Data and Codes to Public	Number of Drawbacks
[3]	◇	◇	●	◇	●	◇	●	◇	◇	◇	●	●	●	●	◇	●	◇	3
[5]	◇	◇	●	◇	●	●	◇	◇	●	◇	●	●	●	◇	◇	●	◇	2
[6]	◇	●	◇	●	●	◇	●	◇	◇	◇	◇	◇	◇	◇	◇	●	◇	4
[7]	◇	●	◇	●	◇	●	●	◇	◇	◇	◇	◇	◇	◇	◇	●	◇	3
[8]	◇	●	◇	●	●	◇	◇	●	◇	◇	●	◇	◇	◇	◇	●	◇	4
[9]	●	●	◇	◇	◇	●	◇	●	◇	◇	◇	◇	◇	●	◇	◇	◇	2
[10]	●	●	◇	◇	◇	●	◇	●	◇	◇	◇	●	◇	◇	●	◇	◇	3
[11]	●	●	◇	◇	◇	●	◇	●	◇	◇	◇	●	◇	◇	●	◇	◇	3
⋆	◇	◇	●	●	●	●	◇	◇	◇	●	●	●	●	●	●	◇	●	-

Table 1: Comparison between various previous works: Rows are various works in HPC-based malware detection and columns are design choices. The alternative shaded and white background represents different categories of tool/setup/model in malware detection using HPCs. Red texts highlight drawbacks, and black texts express the suggested tool/setup/model from this work. Solid dots (•) indicate the use of that tool/setup/model (column) by the reference (row), and hollow dimonds (◇) indicate the non-use of that tool/setup/model by the reference. Star (⋆) is our work. Our work avoids the drawbacks discussed in the table, and quantitatively analyzes how these drawbacks lead to the conclusion that HPCs can reliably detect hardware.

the default use-case for current anti-malware suites. We denote the drawbacks of DBI as **Drawback I** in Table 1.

While DBI is not feasible in online detection systems, other methods in sampling HPCs can incur inaccurate measurements. A plethora of previous works run the evaluated programs on VMs [3, 8–11]. While VMs provide significant benefits to analyze unknown programs (e.g., strong isolation guarantees), HPCs are limited and shared resource between the host and all VMs. Thus, virtualizing HPCs is a challenge in itself [16]. We conducted an experiment and found out that none of the micro-architectural events that HPCs monitor results in identical measurements in the VM and the host machine. In our experiments, we measured HPC values of the SPEC Benchmark Suite[17] 10 times both on a bare-metal machine (Intel i7-6700 CPU with 8Gb RAM, Ubuntu 16.04 Linux, and VMWare Workstation version 14.0.0 as the hypervisor), and subsequently in a VM hosted on the same machine. We used 2 programs *bzip* and *hmmer* from the SPEC Benchmark Suite as examples to show the difference in measured HPC values from VM and bare-metal. We used the *perf* [18] utility to sample HPC values at a maximum rate of 100Hz in each case. We downsampled the measured HPC values

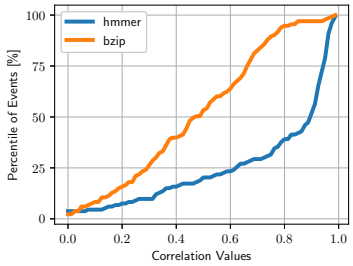

Figure 1: Correlation values between measured HPC values from VM and bare-metal machine versus percentile of events. The plot shows that none of the micro-architectural events has a correlation value of 100% in *hmmer* and *bzip*.

from VM experiments to match the length of the sequence with the measure HPC values from bare-metal experiments. We averaged the measured HPC values to increase Signal-to-Noise Ratio (SNR=mean/variation). We excluded the measured HPC values with only zero values in both VM and bare-metal environment. Figure 1 shows the Cumulative Distribution Function (CDF) of correlation values versus percentiles of events. The correlation values here refer to the Pearson's Correlation between the measured HPC values gathered in VM and bare-metal environment. We observe that none of the events has a correlation of 100% . No events have identical HPC values, in fact, most of the events have low correlations. Thus, the measured HPC values obtained in VM are substantially different from HPC values obtained from the bare-metal environment that real-life users have. To make matters worse, an evasive malware can detect whether it is running in a VM and ceases to exhibit malicious behavior (Kirat et al. [19]). These observations motivate our experimental setup (§3) to run all experiments on bare-metal systems. We label the use of VM in the experimental setups as **Drawback II** in Table 1.

Due to inaccurate HPC measurements [20], previous works [3, 5–7] choose to maximize the measuring granularity by using HPCs without time-multiplexing. Recall that as modern CPUs only have 6 (AMD) or 4 (Intel) registers for HPCs, malware detection methods must select the events from more than 100 available micro-architectural events (130 in AMD Bulldozer and 196 in Intel Skylake). Previous works [3, 5, 9–11] have not provided a numerical analysis on how micro-architectural events are selected. In our experiments, we perform a Principal Component Analysis (PCA) based approach to select the micro-architectural events. After the selection of events, we use HPCs to track these events, and transform the measured HPC values to examples in machine learning models, i.e. feature extraction. We divide examples into training and testing datasets for machine learning models (training-and-testing split). Previous works [3, 5, 6, 8] have training-and-testing split based on the examples (TTA1 in § 4.3) that the testing dataset can have the same examples produced by programs in training dataset. However, in real-life, it is unlikely that the offline training dataset can include all the malware that a user might encounter. We mark the use of data division based on examples as **Drawback III** in Table 1.

In this work, we evaluate our model with 1,000 repetitions of 10-fold cross-validations. The cross-validation examines the machine

learning models with different input training-and-testing examples, which prevent machine learning models from overfitting[3]. We observe that there is no cross-validation in some of the previous works [3, 6, 7], while other works [8–11] present insufficient cross-validation, i.e. not every example in the dataset is validated. None of these works report standard deviations of detection rates with cross-validations. Without a substantial amount of cross-validation, we cannot assert the reproducibility of detection rates, since a model can have its high detection rates with specific training and testing datasets. We refer to no cross-validation or insufficient validations as **Drawback IV** in Table 1.

The prevalence of the above-mentioned drawbacks motivates us to perform rigorous, quantitative, and reproducible analytics for HPC-based malware detection in Table 1. In order to perform a fair comparison with works in Table 1, we use the following machine learning models all used in previous works: Decision Tree (DT), Random Forest (RF), K Nearest Neighbors (KNN), Neural Nets (NN), Naive Bayes and AdaBoost. DT, RF, and KNN are designed to identify outliers, which fit the application of malware detection [21]. NN, NB, and AdaBoost (ensemble machine learning models) are used in many previous works. We evaluate the detection rates of all these machine learning models in our work and compare the results with previous works.

Previous works reported their results with double decimal precision [3]. However, double decimal precision require at least 100 experiments in testing. With 10-fold cross-validation in the experiments, the total number of programs (benignware and malware) should be more than 1,000 programs. Thus, at least 1,000 programs are required to evaluate the machine learning models within numerical rounding error of less than 1%. As a result, we consider the works with fewer than 1,000 programs as over-generalization (training and testing with insufficient cross-validation), or over-interpretation of the results (comparisons beyond rounding errors) [3, 5–8]. This insufficient number of programs in the experiments is **Drawback V** in Table 1.

In addition to the drawbacks of the previous works, we found that there is no public access to their data or codes, which presents a direct comparison and examinations of the methods applied in these works. To ease the reproducibility and advance the community's efforts to assess the utility of HPC-based malware detection, we release all the code and data produced for this work under open-source license.

We present all the tools/setups/models in various previous works in Table 1. In Table 1, rows are various works in HPC-based malware detection and columns are design choices of the tools/setups/models. The alternative shaded and white background represents different categories of tool/setup/model in malware detection using HPCs. Red texts highlight drawbacks, and black texts express the suggested tool/setup/model from this work. Solid dots (•) indicate the use of that tool/setup/model (column) by the reference (row), and hollow dimonds (⋄) indicate the non-use of that tool/setup/model by the reference. Star (⋆) is our work. The last column counts the drawbacks of the corresponding work. Table 1 shows that there are at least 2 drawbacks in each work. Based on our work, we provide 3 guidelines for evaluating HPC-based malware detection in this area. First, measurements of HPCs should be done in a bare-metal environment, without VMs or any DBI. Second, building machine learning models for malware detection requires the data division by programs (TTA2 in §4.3) instead of division by traces (TTA1 in §4.3). Third, repeated cross-validations are required to prevent overfitting of machine learning models.

[3]The model corresponds closely or exactly to a particular data and fail to predict other data reliably.

3 EXPERIMENTAL SETUP

In this section, we explain how we set up the experiments to gather values of HPCs from benignware and malware. We ran our experiments on a cluster with 15 machines as worker nodes, and a master node to distribute jobs to measure and to collect data from worker nodes. We dispatched our jobs to the worker nodes using the Rabbitmq message system [22]. We collected the data back from the worker nodes using a Samba [23] server on the master node. We used Bindfs [24] to fuse the permission bits of Samba server storage folder to be writable, not modifiable, not readable, and not executable. Note that the Portable Operating System Interface (POSIX) permission structure cannot provide the above-mentioned permission bits. These permission bits allowed the worker nodes to record the measured HPC values, while these permission settings prevented malware from overwriting or deleting the measured HPC values. On the worker nodes, we ran our experiments in Windows 7 32-bit operating system to be compatible with malware experiments in other works [9–11]. Previous works applied time-based HPC sampling, i.e., they gathered values at a fixed sampling frequency [3]. We used AMD CodeAnalyst APIs to build a time-based HPC monitoring tool, *Savitor*, since AMD CodeAnalyst itself cannot provide time-based measured HPC values [25].

3.1 *Savitor* (HPC measuring tool)

We designed *Savitor*, a tool that monitors a target process and gathers HPC values related to the process. *Savitor* runs the target process, pins the process to one core, reads the HPC values from that core and then writes the measured HPCs values to files on another core, in order to reduce the noise during the sampling. *Savitor* records 6 HPCs at a time, which is the maximum number of HPCs that can be recorded on the AMD Bulldozer micro-architecture without time-multiplexing. *Savitor* performs time-based sampling and kills the target process at the end of each experiment. Following the frequency limits in CodeAnalyst, we used the maximum possible sampling frequency of 1 KHz for *Savitor*. Considering our limited resources (time and hardware), we only ran each experiment of both malware and benignware for 1 minute.

3.2 Malware and Benignware

For forming the set of malware, we downloaded 1,000 malware from Virustotal [26], and performed a test run of those 1,000 malware on worker nodes. After the test run, we identified 962 malware which could run for more than 1 minute and used them in our malware experiments. According to *AVClass tool* [27], our dataset consisted of 35 distinct malware families.

In order to collect benignware programs, we first installed all the packages and software from Futuremark [28], python performance module [29], ninite.com [30], and Npackd [31] on the worker nodes.

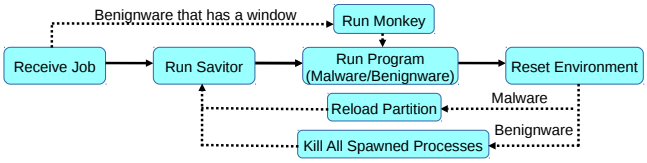

Figure 2: Our workflow of benignware/malware experiments: The worker node receives the dispatched jobs of experiments from the master node. The worker node spawns a *Savitor* process, and then *Savitor* runs the target process (benignware/malware). The dotted arrow (⇢) means that the action does not always happen. If the application has a window for interaction, we attach a monkey tester to the window. The solid arrow (→) shows that actions always happen. We reset the environment after each experiment. the worker node kills any other processes spawned by the target process after each benignware experiment. At the end of each malware experiment, we reboot the machine into the Debian partition to reload a clean Windows image.

After installation, we traversed all the files in "Startup Menu" and "C:\Program Files" folder to include all the unique executable programs in our benignware dataset. We avoided the complication of re-installation by excluding all the executable program files with "uninstall" in their names. We performed a test run of all these programs, and selected 1,382 benignware that could run for 1 minute.

To avoid the classification bias, we matched the number of malware and benignware used in our experiments. Classification bias exists in classification problems if the number of items in each class is different. For example, in a classification problem with two classes, A and B, if class A makes up 80% of the data set and class B makes up 20% of the dataset, the baseline of precision in classifying A is 80%. Any designed machine learning models whose precision is lower than 80% are worse than the precision estimated with prior probability. In our work, we matched the number of benignware and malware; at the same time, we reported precision, recall and F1-score to eliminate any bias.

3.3 Method for Running Experiments

We ran our benignware and malware experiments on identical hardware and operating system. However, there are a few differences between malware and benignware experiments. We explain the workflow of malware and benignware experiments using one dispatched job in Figure 2. The boxes are the steps that we follow, and the solid arrow means that the next step always happens. The dotted arrow means that the action happens under the conditions of the labels.

3.3.1 Malware Experiment. We follow the steps in Figure 2 to run the experiments. Before any malware experiments, we dropped all the requests to any network outside the master node, to ensure that malware does not affect other machines. At the beginning of each experiment, the worker node runs a clean copy of Windows and waits for a new job. Once the worker node receives the job from the master node, *Savitor* runs the malware and records the measured HPC values. After running each malware experiment, we provide an identical, malware-free environment for the next malware experiment by reloading the Windows partition. In order to reload Windows image, we installed Debian 8 in the other partition of the hard drive on each worker node. Whenever a worker node boots into the Debian partition, the worker node copies a clean Windows image to the other partition. We modified the GNU GRand Unified Bootloader (GRUB) to make the machine boot into an alternate partition every time it reboots. After reloading the image, the system reboots into Windows again and runs the next job dispatched from the master node.

3.3.2 Benignware Experiment. Similar to the malware experiments, benignware experiments also follow the workflow in Figure 2. We connected the worker nodes to the outside network to ensure the benignware receives network responses. Programs, such as browsers, require network responses to perform similarly as in a user environment. When the worker node receives a job from the master node, *Savitor* starts the target process (benignware program), and a monkey tester is attached to the target process if the target process has an interactive window. The Monkey tester works similar to Android's Monkey tester [32], as it interacts with the target process by periodically sending random keystroke, mouse clicks, and scrolling operations to the window of the target process. The behavior of the monkey tester simulates the interaction between a user and the programs. After *Savitor* samples the measured HPC values, the system resets by killing any processes spawned during the experiments. Since the benignware does not try to infect the Windows partition and perform malicious operations, we do not reload the Windows partition. After killing the spawned processes, the worker node receives the next job from the master node and starts the next experiment.

4 MACHINE LEARNING MODELS

In this section, we present how we apply machine learning models on measured HPC values. One of the common problems in machine learning is the *Curse of Dimensionality*. *Curse of Dimensionality* means that machine learning models in a high-dimensional space have lower detection rates compared to models in lower-dimensional spaces [21]. The redundant dimensions in high dimensions contribute to the measurement of noise in the training dataset, which result in a decrease in the detection rates of testing. *Curse of Dimensionality* motivates the reduction of dimension; however, reducing dimensions may cause underfitting due to the lack of representation during training. In order to overcome both overfitting and underfitting, the design of machine learning models requires the minimum number of features that represent most of the measured HPC values. To this end, we perform a quantitative analysis to extract features from the measured HPC values of our selected micro-architectural events.

4.1 Reduction of Dimensions

In this work, we use Principal Component Analysis (PCA) to reduce the dimensions. By reducing the dimensions, the machine learning models can use the linearly independent components to easily classify the examples into different classes. Here, we show one *synthetic* dataset (a subset from our experiments) separated from overlapping measured HPC values by applying PCA results. In the next subsection (§4.2), we explain how we choose the sizes of examples and features.

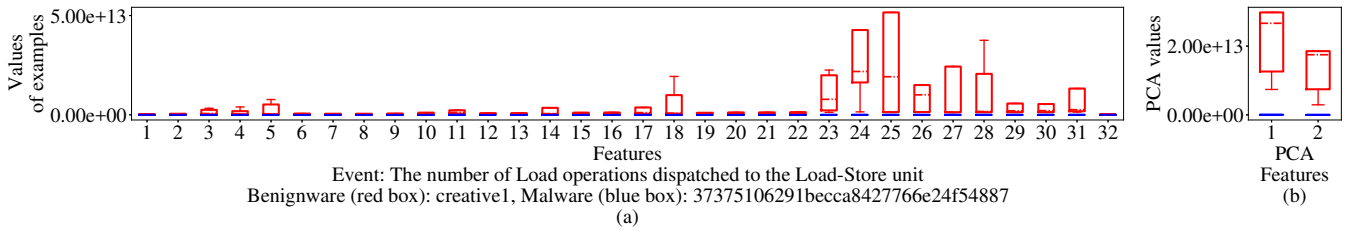

Figure 3: X axis is the feature number and Y axis is the values of each example. Red box corresponds to the malware and blue box corresponds to the benignware. The dashed line is the mean of each distribution. The boxes represent 25% ∼ 75% of the distributions. The whiskers (the short, horizontal lines outside the boxes) represent the confidence interval equivalent to $\mu \pm 3\sigma$ of Gaussian Distribution (0.3% ∼ 99.7%). We measure *The number of Load operations dispatched to the Load-Store unit* event 5 times in one benignware (creative2 from Futuremark) and one malware. The distributions of the two subplots represent 5 examples in the experiments. (a) Distributions of sampled values before the reduction of dimensions: We cannot distinguish between the 5 malware examples and the 5 benignware examples. (b) Distributions of sampled values after the reduction of dimensions: We apply the reduction of dimensions to examples in (a) to get examples in (b). We can separate all the examples in (b) due to the gaps between values of malware and benignware in both features.

PCA applies eigen-decomposition to decompose the training standard matrix (A), where columns are features and rows are examples, into the multiplication of eigenvectors (V) and eigenvalues (λ) in Equation 1. The standard matrix (A) is transformed into lower-dimensional data space by multiplying the eigenvector matrix V, which can also be approximated with the major eigenvector matrix (V').

$$A = V\lambda V^{-1} \approx V'\lambda V'^{-1} \tag{1}$$

We present the distributions of examples before and after the reduction of dimensions, $A_{5\times32}$ in Figure 3(a) and $A_{5\times32}V'_{32\times2}$ in Figure 3(b). We measure *the number of Load operations dispatched to the Load-Store unit* (Table 2) event 5 times in one benignware (creative2 from Futuremark) and one malware [4]. The input matrices (A) of both benignware and malware have 32 features and 5 examples. In Figure 3, X axis shows the feature number and Y axis shows the values of each example. Red box refers to the malware and blue box refers to the benignware. The dashed line is the mean of each distribution. The boxes represent 25% ∼ 75% of the distributions. The whiskers (the short, horizontal lines outside the boxes) represent the confidence interval equivalent to $\mu \pm 3\sigma$ of Gaussian Distribution (0.3% ∼ 99.7% of the total distributions).

From Figure 3(a), we can see overlapping of boxes and whiskers in all the columns. Figure 3(b) shows the results of the data matrix (A) multiplied with the eigenvector matrix (V'). We can clearly classify the malware or benignware, since there are gaps between the distributions of malware and benignware in both features. By multiplying the eigenvector matrix, different features contribute to classification with weights according to their abilities to discriminate data. Hence, we can achieve higher classification rates with lower dimensional data.

4.2 Selection of Events

As discussed in §4.1, we reduce the dimensions to extract features from the measured HPC values to form the machine learning models. At the same time, the hardware limitation on number of HPCs without time-multiplexing requires the selection of events from more than 100 available micro-architectural events. Hence, we designed a method to select our micro-architectural events, while reducing the dimensions of examples at the same time.

In our method, our selection of events is based on minimizing 3 sources of losses These 3 main sources of losses in the measured HPC values are:

- Jitter: the timing variations between identical measurements of the measured HPC values.
- Noise: the amplitude variations between identical measurements at the same time-stamp of the measured HPC values.
- Approximation error: the loss of the minor eigenvectors.

Jitter and noise are introduced due to the limitations in the measurements. As we will discuss in §6, noise and jitter cannot be eradicated. To minimize the impact from jitter, we divide the measured results into 32 equal time intervals, and sum the gathered values in each time interval to form 32 histogram bins (each bin corresponds to one feature). This is the same design choice as the one used by Demme et al. [3]. Histogram bins preserve the sampled information, while reducing the effects of jitter in the values of HPCs. In addition to jitter, we observe noise in the measured HPC values, as Weaver et al. do in their work [20, 33]. To minimize the noise for our selection of events, we repeat the measurements on the same program and the same events 32 times, and then we calculate the cumulative sum in each bin, in order to increase the Signal-to-Noise Ratio (SNR). Assuming the noise introduced during the measurement is Additive White Gaussian Noise (AWGN) [34], this approach increases the SNR by a factor of 32.

Approximation error is introduced by the elimination of minor eigenvectors in V when we transform V to V'. For each example, we multiply the measured HPC values to the major eigenvector matrix V' instead of V. In our method, by trading off the number of eigenvectors in the major eigenvector matrix, we reduce the dimensionality and increase the approximation error in Equation 2. We use the product of the standard matrix A and the eigenvector matrix V' as our input matrix in machine learning model as we described in Equation 1.

[4] SHA256 hash value: 3737 5106 291b ecca 8427 766e 24f5 4887

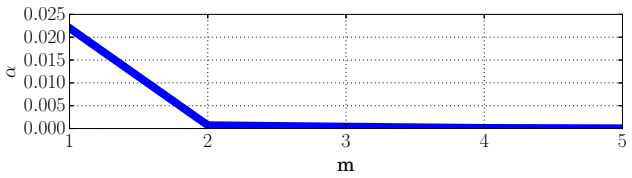

Figure 4: Error Bound vs the Number of Eigenvetors Plot: when choosing different number of eigenvectors for reduction in dimensions, the error bound α changes according to m eigenvectors.

$$AV = \sum_{i=1}^{m} v^{(i)}\lambda^{(i)} + \sum_{i=m+1}^{n} v^{(i)}\lambda^{(i)} \tag{2}$$

$$= \sum_{i=1}^{m} v^{(i)}\lambda^{(i)} + \epsilon(\alpha v\lambda) \tag{3}$$

In equation 2, $\lambda^{(i)}$ denotes the i^{th} largest eigenvalue with n eigenvalues (λ). $v^{(i)}$ is the corresponding eigenvector of $\lambda^{(i)}$, and m is the number of reduced dimensions. Equation 2 represents the separation of m major and $n-m$ minor eigenvectors. The first term in Equation 2 is AV'. The approximation error is the difference between AV and AV', which is the second term in Equation 2. In Equation 3, ϵ denotes the upper bound function. α denotes error coefficient, with the error term ($AV - AV'$) divided by the original input data (AV). Equation 3 expresses that with a given m value, we can estimate the approximation error using α. By having more eigenvectors in the eigenvector matrix V' (larger m), we can reduce α, which corresponds to a lower approximation error. As we observe from Equation 3, the approximation error depends on the choice of eigenvectors. We cannot determine the eigenvectors before we train and test our dataset. However, we can use a subset of programs to compute the eigenvectors and choose the parameters in Equation 3.

As in real-life, it is impossible to use the entire dataset for the selection of events. Here, we chose a subset of programs, **7** programs from the Futuremark [28] benchmark for the selection of events. The choice of programs from Futuremark benchmark suite is driven by the fact that Futuremark has analyzed user behavior and automated this behavior in the benchmarks. All the programs of Futuremark benchmark are real-world applications commonly used in office. We measured the programs at at the frequency of 1 kHz for 1 minute, as we described in §3.1. Our experimental hardware (AMD Bulldozer micro-architecture) enables us to monitor **130** events 6 at a time. We accumulated the measured HPC values into **32** bins, with each measurement summed into 32-dimension vector. Thus we ran each of the 7 programs from Futuremark Benchmarks on 130 micro-architectural events 32 times (**130×32×7**).

$$A_{e_k}V_{e_k} = \sum_{i=1}^{m} v_{e_k}^{(i)}\lambda_{e_k}^{(i)} + \epsilon(\alpha v_{e_k}\lambda e_k) \tag{4}$$

$$\alpha(m) = \min_{e_j} \frac{\sum_{i=m+1}^{n} v_{e_j}^{(i)}\lambda_{e_j}^{(i)}}{v_{e_j}\lambda e_j} \tag{5}$$

Table 2: Description of the Selected Events [2]

Events	Definition
0x04000	The number of accesses to the data cache for load and store references
0x03000	The number of CLFLUSH instructions executed
0x02B00	The number of System Management Interrupts (SMIs) received
0x02904	The number of Load operations dispatched to the Load-Store unit
0x02902	The number of Store operations dispatched to the Load-Store unit
0x02700	The number of CPUID instructions retired

With the results (**130×32×7**) from the experiments, we denote the k^{th} event as e_k, its i^{th} eigenvalue as $\lambda_{e_k}^{(i)}$, and the corresponding eigenvector as $v_{e_k}^{(i)}$ for $k = 1, 2, \ldots 130$ in Equation 3, in order to re-write Equation 3 into Equation 4. In Equation 5, e_j corresponds to the 6 events with the minimum α when $j = 1, 2, \ldots 6$, excluding the events whose measured HPC values are all zeros. We apply Equation 1 to compute v_{e_k} and λ_{e_k}. We calculate the eigenvalues for 130 events and find out that there is no event among 130 events with more than 10 eigenvectors ($n \leq 10$). We exclude all the events that only have zero values in the measured HPC values, since these events provide no signal in the measured HPC values. By changing the number of eigenvectors (m), we can calculate the error coefficient (α) in Equation 5. We plotted the error coefficients for $m = 1, 2 \ldots 5$ in Figure 4. The gradient of α decreases when m is more than 2. Subsequently, we consider the optimal trade-off between m and α when $m = 2$ and $\alpha(2) = 0.072\%$, which corresponds to the upper bound of error as 0.072% AV, with the linear combination of 2 components from v_{e_j}.

The 6 events, which we selected in our experiments, are listed in Table 2. We assemble the eigenvectors of 6 events, 2 for each event, and we get the v matrix in Equation 6.

$$v_{192\times12} = \begin{bmatrix} v_{e_{k_1}}^{(1)}, v_{e_{k_1}}^{(2)}, & & & \\ & v_{e_{k_2}}^{(1)}, v_{e_{k_2}}^{(2)}, & & \\ & & \ddots & \\ & & & v_{e_{k_6}}^{(1)}, v_{e_{k_6}}^{(2)} \end{bmatrix} \tag{6}$$

In Equation 6, $v_{e_{k_j}}^{(i)}$ represents the i^{th} largest eigenvector in j^{th} event. Each $v_{e_{k_j}}^{(i)}$ is a 32×1 eigenvector. By multiplying each example with the v eigenvector, we reduce the dimensions from 192 (6 *events* × 32 *bins*) to 12 (6 *events* × 2 *components*).

In summary, we list following steps to select the events:

- Run 7 programs 32 times and measure 130 micro-architectural events.
- Divide the total run time of each program into 32 intervals and sum the measured HPC values in each interval into a separate bin.
- Sum the bins across different runs of the identical measurements.
- Apply PCA on 130 events with 7 programs.
- Compute the approximation errors for 7 programs.
- Find 6 events with the least approximation errors.

Four of the selected events in our experiments align with other works that do not provide any analysis of their selection of events [3, 5, 9–11]. We observe that one event is related to data cache load and

store references. Two other micro-architectural events are related to load and store operations, which have also been used in other works. It is not clear how load and store operations deterministically contain the information of malicious behavior. Any statistics of memory behavior should be legitimate in program execution, since the memory accesses inheretly exist in every program. In our selection of events, we include the events in kernel mode to capture complete program behavior. The remaining 3 selected hardware events related to cache flush behavior, system management interrupts and CPUID instructions. However, we cannot infer any reasons why these instructions/operations by the kernel can be mapped to any malicious user-level behavior.

4.3 Classification Models

In §4.2, we selected the 6 events to monitor and formulate the eigenvector matrix in Equation 6. With this method, we can extract features from the measured HPC values to get examples for machine learning models, i.e. traces in our datasets of benignware and malware.

To have the same number of measurements on the same program samples as in §4.2, we run each benignware program and each malware program 32 times, and collect 61,568 ($2 \times 962 \times 32$), 30,784 for benignware and 30,784 for malware, measured HPC values (1,026 CPU hours). We sum the measured HPC values into 32 histogram bins (as described in §4.2) for each of 6 events. Each example of histogram binned HPC values has 192 (6 $events \times 32$ $bins$) features. By multiplying each example with the v eigenvector in Equation 6, we reduce the dimensions from 192 (6 $events \times 32$ $bins$) to 12 (6 $events \times 2$ $components$). To this end, we convert the **measured HPC values** into histogram bins, and then transform them into **traces**.

Using the reduction of dimensions (§4.2), the input matrix $A_{30,784\times192}$ (30,784 examples and 192 features) of benignware or malware is transformed to lower-dimensional space as $A'_{30,784\times12}$ (30,784 examples and 12 features). For training and testing of the machine learning models, we are going to separate the examples in matrix A' into training and testing datasets (training-and-testing split). In our experiments, we consider 2 Training-and-Testing Approaches (TTA) to divide our dataset into training set and testing set. The two approaches are as follows:

TTA1 Dividing 30,784 traces with a split of 90:10 ratio, resulting in 27,704 traces (90% of 30,784 traces) as training dataset and 3,078 traces (10% of 30,784 traces) as testing dataset both in benignware and malware experiments.

TTA2 Dividing 962 programs with a split of 90:10 ratio, resulting in traces of 866 programs (90% of programs) as training dataset and traces of 96 programs (10% of programs) as testing dataset both in benignware and malware experiments.

In the first training-and-testing approach (TTA1), we randomly choose 27,704 traces as training dataset and 3,078 traces as testing dataset both in benignware and malware experiments. In this approach, the traces resulting from the same program sample can appear in both training and testing datasets. As a result, such an approach corresponds to a highly optimistic and unrealistic scenario where the testing programs (benignware or malware) are available during training. Given that thousands of new malware appearing everyday, it is impossible to include all the malware that user may encounter. Hence, TTA1 should not be applied in training machine learning models for malware detection.

In the second training-and-testing approach (TTA2), we randomly choose traces of 866 programs as training dataset and traces of 96 programs as testing dataset both in benignware and malware experiments. TTA2 corresponds to a realistic case where during training model, we do not have access to the exact programs, benign or malicious, that users run in the real life. To validate across our models, we perform 10-fold cross-validations 1,000 times. For each 10-fold cross-validation, we randomly shuffle the dataset to ensure difference across 1,000 rounds. In each 10-fold cross-validation, each example in the dataset is used in training 9 times and testing once. This ensures the identical times of training and testing for every single example, compared to randomly shuffling the data and validating the machine learning models. With 1,000 10-fold cross-validations, we can ensure that the standard deviations of detection rates increase no more with more rounds of validations.

In our experiments, we perform training and testing with both TTA1 and TTA2. We compare the detection results in terms of precision, recall, F1-score, and Area Under Curve (AUC) in both approaches. We use the implementations of machine learning models in scikit-learn package [35]: DT, RF, NN, KNN, AdaBoost, and Naive Bayes. The seed for randomness in machine learning initialization and division of data comes from the random number generator "`/dev/urandom`". During training, we set the parameters of the machine learning models as described below to prevent the machine learning models from underfitting due to default limitations in computational resources set by scikit-learn. We used default values for the remaining parameters in scikit-learn.

- DT: We set the maximum depth as 100 to classify the malware and benignware. The number of levels is sufficient to avoid any underfitting of the model.
- RF: We set the maximum depth to be the same as in the DT. We enable a maximum 200 estimators in the RF. The number of estimators is sufficient to avoid any underfitting of the model.
- NN: The network we use here is a Multilayer Perceptron (MLP) neural network having 4 layers with 100 neurons in each layer. We apply "tanh" function as the activation function. We use $L2$ regularization on the parameters in the NN.
- KNN: We choose 5 as the number of nearest neighbors, and perform experiments with K value varying from 1 to 20. When K equals to 5, the F1-score reaches the highest detection rates.
- AdaBoost: Adaboost is an Ensemble classifier, which utilizes a collection of estimators. Adaboost fits a sequence of classifiers on the training data. The predictions are decided based on a majority vote [36]. The default value of the number of estimators is 50. We use 200 estimators instead of 50, since our test experiments show that 200 estimators have a higher detection rate for AdaBoost.
- Naive Bayes: We use the same number of malware and benignware traces in the model. The prior probability is 50%.

Table 3: Detection Rates with TTA1 and TTA2: Red means the value is less than 50% and bold means that the value is more than 90%

Models	TTA1				TTA2			
	Precision[%]	Recall[%]	F1-Score[%]	AUC[%]	Precision[%]	Recall[%]	F1-Score[%]	AUC[%]
Decision Tree	83.04	83.75	83.39	89.65	83.21	77.44	80.22	87.36
Naive Bayes	70.36	7.97	14.32	58.11	56.72	5.425	9.903	58.38
Neural Net	82.41	75.4	78.75	84.41	**91.34**	22.16	35.66	66.43
AdaBoost	78.61	71.73	75.01	80.57	75.78	65.6	70.32	77.96
Random Forest	86.4	83.34	84.84	**91.84**	84.36	78.44	81.29	89.94
Nearest Neighbors	84.84	82.37	83.59	89.26	82.7	77.88	80.22	86.98

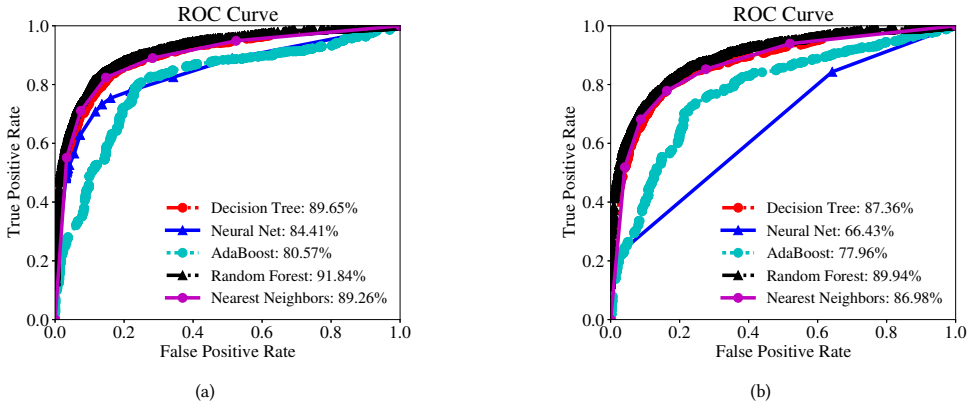

Figure 5: Receiver Operating Characteristic (ROC) curve of 5 models. (a) The AUC of DT, NN, AdaBoost, RF, and KNN using (TTA1) is 89.65%, 84.41%, 80.57%, 91.84%, and 89.26%, respectively. (b) The AUC of DT, NN, AdaBoost, RF, and KNN using (TTA2) is 87.36%, 66.43%, 77.96%, 89.94%, and 86.98%, respectively.

5 EXPERIMENTAL RESULTS

In this section, we show our results with the experiments to detect malware using HPCs and contrast the ones obtained in previous works. We report malware detection rates in terms of precision, recall, F1-score, and Area Under Curve (AUC) in Receiver Operating Characteristic (ROC) plots. We use the positive label to denote malware and the negative label to denote benignware. True positive samples (T_+) are malware programs that are classified as malware. False positive samples (F_+) are benign programs that are classified as malware. False negative samples (F_-) are malware programs that are classified as benignware. Precision is defined as the number of true positive samples (T_+) divided by the number of all the positive samples, ($T_+ + F_+$) in Equation 7. Recall is defined as the number of true positive samples (T_+) divided by the sum of the number of true positive (T_+) and the number of false negative (F_-) samples in Equation 7. The F1-score is the harmonic mean of precision and recall in Equation 8.

$$Precision = \frac{T_+}{T_+ + F_+} \qquad Recall = \frac{T_+}{T_+ + F_-} \tag{7}$$

$$F1 - score = \frac{2 \times Recall \times Precision}{Precision + Recall} = \frac{2T_+}{2T_+ + T_- + F_-} \tag{8}$$

The ROC curve represents how the true positive rate varies with different thresholds for the false positive rate. We can reach 100% true positive rate only if we accept a false positive rate of 100%. Conversely, if we want to achieve a 0% false positive rate, then that leads to 0% true positive rate. By changing the false positive rate threshold, we can trade-off the false positive rate with the true positive rate. Thus we use AUC of ROC curve to measure how effective classifiers are at various false positive rate thresholds.

5.1 Malware Detection

In this section, we report the detection rates (precision, recall, and F1-score) with 2 different data divisions, **TTA1** and **TTA2**. **TTA1** is the division of data according to the *traces*; while **TTA2** is the division of data according to the *programs*, as defined in §4.3.

5.1.1 Results from TTA1 Experiments. We train and test traces using various machine learning models and determine the detection rates (precision, recall, and F1-score) with **TTA1**. Then we plot the ROC curves and compute the AUCs. Table 3 shows the precision, recall, F1-score, and the AUCs of ROC curves. Any results with a value larger than 90% and smaller than 50% are set in **bold** and red, respectively. Figure 5 shows the ROC curves and the AUCs of ROC for different machine learning models.

DT uses the different features to classify examples at different tree branches. RF uses a collection of DTs to perform classifications. KNN determines the classes of each examples by comparing the number of examples within predefined distances. DT, RF, and KNN

models target classifying outliers in the dataset [37], which fit our malware detection problem. According to our results, the detection rates of precision, recall, and F1-score are higher in DT, RF, and KNN models than any other models. The F1-scores in DT, RF, and KNN models are 83.39%, 84.84%, and 83.59%. Figure 5 shows the higher true positive rates of RF and DT with different thresholds, compared to other models. Accordingly, the AUCs in DT, RF, and KNN are 89.65%, 91.84%, and 89.26%. Figure 5(a) shows that the AUCs of ROC curves in DT and RF are the highest in various thresholds of false positive rates.

AdaBoost model leverages a collection of machine learning models. AdaBoost and NN model are designed to classifying clusters of examples. They perform worse in terms of detection rates compared to DT, RF, and KNN, as these models are designed to classify outliers. The F1-scores in AdaBoost and NN are 75.01% and 78.75%. In Figure 5(a), AdaBoost and NN models are also worse than DT and RF models. The AUC values for AdaBoost and NN are 80.57% and 84.41%.

The classification of Naive Bayes is only based on the probabilities of the occurrences of malware and benignware, which is a poor assumption to design classifiers [38]. In our design, we use the prior probability (50%) to design the Naive Bayes classifier. Naive Bayes model has many false negatives, which causes the F1-score value to be as low as 14.32%. The AUC of Naive Bayes is 58.11%. As a result, Naive Bayes classifies examples between malware and benignware with low detection rates.

5.1.2 Results from TTA2 Experiments. We perform another experiment of training and testing using various machine learning models to show the detection rates (precision, recall, and F1-score) with **TTA2**. The F1-scores of DT, RF, KNN, Naive Bayes, AdaBoost, and NN models are 80.22%, 81.29%, 80.22%, 9.903%, 70.32%, and 35.66% using **TTA2**, compared to 83.39%, 84.84%, 83.59%, 14.32%, 75.01%, and 78.75% using **TTA1** in Table 3. By using **TTA2**, the detection rates are lower compared to the scenario using **TTA1**.

Figure 5(b) shows the ROC curves and the AUCs of ROC for different machine learning models using . The AUCs of ROC of DT, RF, KNN, Naive Bayes, AdaBoost, and NN models are 87.36%, 89.94%, 86.98%, 58.38% 77.96%, and 66.43% using **TTA2** in Figure 5(b), compared to 89.65%, 91.84%, 89.26%, 58.11%, 80.57%, and 84.41% using **TTA1** in Figure 5(a).

Demme et al. showed precision varying from 25% ∼ 100% [3] among different families of malware, without any recall values reported using **TTA1**. The median precision among all the families of malware is around 80%, with **TTA1**. Precision value of 80% corresponds to the False Discovery Rate[5] of 20%. Consider that a default Windows 7 installation has 1,323 executable files, an AV system with a 20% False Discovery Rate would flag 264 of these files incorrectly as malware – clearly such a detection system would not be practical. As a result, such a malware detection method is not usable in real-life systems. With thousands of malware reported everyday, the offline training of malware detection cannot capture the same malware program that a user may encounter. In real-life cases, the malware detection rates of HPC-based malware detection would be those in columns of **TTA2** of Table 3 and Figure 5(b). These results show that high detection rates and robustness in detection are over-estimated due to division of data during training. Our comparison using **TTA1** and **TTA2** shows that using **TTA2** can cause the precisions to be even lower. Thus, prior works could have even worse precisions by using **TTA2**. In the next subsection, we will show that the results presented in this subsection are not an exception.

[5]False Discovery Rate $(F_+/(F_+ + T_+)$

5.2 Cross-Validation

Cross-validation is a common practice in machine learning for avoiding the overfitting of machine learning models. Cross-validation is used to validate whether the detection rates are consistent with repeated training and testing [39]. If the detection rates fluctuate during cross-validation, we can infer that the machine learning models are not trained properly. We observe that previous works either have no cross-validation or report no results from cross-validations. The lack of proper cross-validation motivates us to further evaluate the machine learning models using cross-validation. We use 3 times standard deviation (3σ) to quantify the fluctuations in detection rates. 3σ refers to 0.3% ∼ 99.7% of random instances distributed within the range of 3σ. In the context of malware detection, a high value of 3σ in detection rates means that the performance of the model is not stable across different datasets.

5.2.1 Cross-validations for TTA1 Experiments. A common practice of cross-validation is using 10-fold cross-validation [39]. 10-fold cross-validation divides the dataset into 10 subsets with equal number of examples. It then performs training on 9 subsets and testing on the remaining one, with each subset as a testing subset. The standard deviations of detection rates in 10 experiments show whether the detection rates of the model are stable across 10 experiments. We consider that either a split of $60 - 20 - 20$ training-testing-validation or 10-fold cross-validation is not sufficient cross-validation, since the standard deviations of the detection rates increase with more examples in the dataset. We repeated the 10-fold cross-validations until the standard deviations of detection rates do not increase with more cross-validations. In this work, we perform cross-validation 1,000 times (randomly shuffling the examples before each training-and-testing split), which is 3 orders of magnitude more than previous works.

Figure 6 shows the distributions of detection rates (precision, recall, and F1-scores) with both **TTA1** and **TTA2** for various machine learning models. In Figure 6, the red diamonds are the means, and the blue boxes correspond to distributions of detection rates (Precisions, Recalls, and F1-scores) lying between 25 and 75 percentiles. The whiskers (the short, horizontal lines outside the blue box) represent the distributions of detection rates lying between 0.3% and 99.7%, which is equivalent to $\mu \pm 3\sigma$ of a Gaussian Distribution. The blue dots are outliers that are outside the $\mu \pm 3\sigma$ regime. A wide spread of distributions in detection rates means that the detection rates fluctuate across different training datasets. Conversely, a narrow spread of distributions means that the detection rates are stable across different training datasets. In DT, RF, KNN, NN, AdaBoost, and Naive Bayes models, the mean of distributions of F1-scores are 82.17%, 83.75%, 82.28%, 74%, 72.27%, 12.15%, with 3 standard deviations (3σ) of 1.416%, 1.326%, 1.388%, 13.2%, 2.365%, 2.392%,

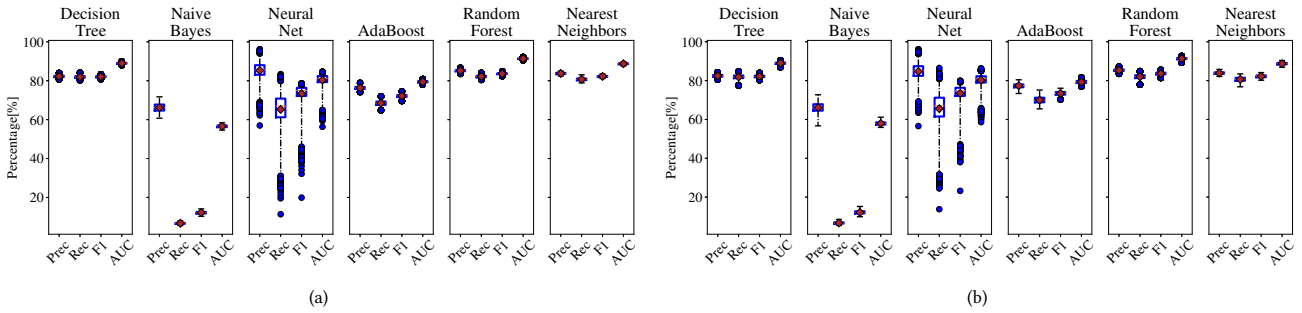

Figure 6: Box plots of distributions of 10-fold cross-validation experiments using (a) TTA1 and (b) TTA2. Red diamonds are means, and blue box corresponds to cross-validation experiment results that lie between 25 and 75 percentiles. The whiskers (the short, horizontal lines outside the blue box) represent confidence interval equivalent to $\mu \pm 3\sigma$ of a Gaussian Distribution. The blue dots are outliers that are outside the $\mu \pm 3\sigma$ regime. On the X-axis, Prec is precision, Rec is recall, and F1 is F1 score. AUC is area under curve in ROC. These 10-fold cross-validation experiments show that we cannot achieve 100% malware detection accuracy.

5.2.2 Cross-validations for TTA2 Experiments. In DT, RF, KNN, NN, AdaBoost, Naive Bayes models, the mean of distributions of F1-scores using **TTA2** are 82.13%, 83.61%, 82.2%, 73.69%, 73.43%, 12.21%, compared to 82.17%, 83.75%, 82.28%, 74%, 72.27%, 12.15% using **TTA1**, respectively. In DT, RF, KNN, NN, AdaBoost, Naive Bayes models, the mean of distributions of F1-scores using **TTA2** are 2.145%, 2.336%, 2.248%, 14.88%, 3.29%, 2.611%, compared to 1.416%, 1.326%, 1.388%, 13.2%, 2.365%, 2.392% using **TTA1**, respectively. Comparing the results using **TTA1** and **TTA2**, the standard deviations of DT, RF, KNN, NN, AdaBoost, Naive Bayes models increased by 1.515×, 1.762×, 1.62×, 1.127×, 1.391×, 1.092×, respectively. The overall detection rates using **TTA2** have much higher variations compared to ones using **TTA1**.

As previous works did not report standard deviations of their cross-validations, we cannot compare these results. From the Figure 6, we can conclude that reporting results of one training-and-testing experiment does not provide sufficient information in performance of machine learning models. We can only evaluate the performance of these models by providing a distribution of detection rates.

The difference between standard deviations in Figure 6(a) and Figure 6(b) is due to the unrealistic assumption that the programs in the training set appear in the testing dataset. Figure 6(b) presents the results where the malicious program is not included in the training dataset. In conclusion, the mean of the distribution using **TTA2** is lower than that using **TTA1**, while the standard deviation of distribution using **TTA2** is higher than that using **TTA1**. In order to have a full evaluation on the machine learning models, it is imperative to use **TTA2** and exhibit a distribution of precision, recall, F1-score, and AUC of ROC curves.

5.3 Ransomware

In previous sections, the machine learning models are trained over the traces of HPCs to discriminate malware from benignware. We build a malware embedded in benignware and then show that this malware can evade HPC-based malware detection.

We craft the malware *simply* by infusing Notepad++ with a ransomware. Ransomware is malware that maliciously encrypts files and extorts users in exchange for the decryption keys [40]. By 2016, ransomware has become one of the most popular malware, as Kaspersky Security Bulletin 2016 has shown that at least one business is attacked by ransomware every 40 seconds [41]. We implement our ransomware to encrypt files when Notepad++ launches. The embedded ransomware traverses all the files in the "Pictures" folder and encrypts each file every 5 seconds with Microsoft Cryptography APIs [42]. We measure the values of HPCs for modified Notepad++ in our experimental setup (§ 3). We randomly select 90% of the benignware and malware samples as the training set, while we test on Notepad++ and modified Notepad++. The precision of DT, Naive Bayes, NN, AdaBoost, RF and KNN is 0%, 0%, 0%, 50.85%, 0%, and 0%, respectively.

These results are not surprising, as machine learning models tolerate the noise and jitters during training on sampled HPCs, in order to extract the malicious behavior in the programs. These tolerance necessitates the machine learning algorithms to have errors even with the training datasets. In our malware example, the changes of HPC values caused by ransomware is overshadowed in the sampled values of HPCs from running Notepad++. The variation tolerance results in classifying the modified Notepad++ as benignware.

6 DISCUSSION

We run Windows 7 32-bit operating system on AMD 15h family Bulldozer micro-architecture machine. Weaver et al. performed extensive studies investigating the determinism of the measured HPC values in various micro-architectures [33]. By comparing the HPC values across different micro-architectures, Weaver et al. show that the HPCs in various architectures have similar levels of variations during sampling. Hence, our conclusions from Bulldozer micro-architecture are applicable to other micro-architectures. In our benignware and malware experiments, we chose to allow the access to the network for benignware and prevent malware from accessing network. This design choice does not affect the results of HPC measurements, since benignware and malware both function properly during experiments. For the reduction of dimensions, many other approaches can server the same purpose as PCA. We

use PCA in our designs as PCA is one of the most popular methods for reduction of dimensions.

7 CONCLUSION

HPCs are hardware units that are designed to count low-level, micro-architectural events. Many works have investigated malware detection using HPC profiles. However, we believe that there is no causation between low-level micro-architectural events and high-level software behavior. The strong positive results in the previous works are due to a series of optimistic assumptions and unrealistic experimental setups. In this work, we rigorously evaluate the idea of malware detection using HPCs through realistic assumptions and experimental setups. We observe the low fidelity in HPC-based malware detection when we increase number of programs by a factor of $2 \sim 3$ and the experiment numbers in cross-validation to 3 orders of magnitude higher than previous works. Our best result shows an F1-score of 80.78%. The corresponding False Discovery Rate ($F_+/(F_+ + T_+)$ is 15%. This means that among 1,323 executable files in the Windows operating system files, 198 files will be flagged as malware. We also demonstrate the infeasibility in HPC-based malware detection with Notepad++ infused with a ransomware, which cannot be detected in our HPC-based malware detection system.

REFERENCES

[1] *Intel Itanium Architecture Software Developer's Manual.* Intel Corporation, 2010.
[2] *BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 10h-1Fh Processors.* Advanced Micro Devices, Inc., 2015.
[3] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. On the feasibility of online malware detection with performance counters. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, page 559, 2013.
[4] Harish Patil, Robert Cohn, Mark Charney, Rajiv Kapoor, Andrew Sun, and Anand Karunanidhi. Pinpointing representative portions of large intel itanium programs with dynamic instrumentation. In *Proceedings of 37th International Symposium on Microarchitecture (MICRO)*, pages 81–92, 2004.
[5] Mikhail Kazdagli, Vijay Janapa Reddi, and Mohit Tiwari. Quantifying and improving the efficiency of hardware-based mobile malware detectors. In *Proceedings of the 49th International Symposium on Microarchitecture (MICRO)*, pages 1–13, 2016.
[6] Xueyang Wang, Sek Chai, Michael Isnardi, Sehoon Lim, and Ramesh Karri. Hardware performance counter-based malware identification and detection with adaptive compressive sensing. *Transactions on Architecture and Code Optimization (TACO)*, 2016.
[7] Adrian Tang, Simha Sethumadhavan, and Salvatore J Stolfo. Unsupervised anomaly-based malware detection using hardware features. In *International Workshop on Recent Advances in Intrusion Detection (RAID)*, pages 109–129, 2014.
[8] Baljit Singh, Dmitry Evtyushkin, Jesse Elwell, Ryan Riley, and Iliano Cervesato. On the detection of kernel-level rootkits using hardware performance counters. In *Proceedings of the 17th Asia Conference on Computer and Communications Security (AsiaCCS)*, pages 483–493. ACM, 2017.
[9] Meltem Ozsoy, Caleb Donovick, Iakov Gorelik, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Malware-aware processors: A framework for efficient online malware detection. In *Proceedings of the 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 651–661, 2015.
[10] Khaled N Khasawneh, Meltem Ozsoy, Caleb Donovick, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Ensemble learning for low-level hardware-supported malware detection. In *International Workshop on Recent Advances in Intrusion Detection (RAID)*, pages 3–25, 2015.
[11] Khaled N Khasawneh, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Lei Yu. Rhmd: evasion-resilient hardware malware detectors. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 315–327, 2017.
[12] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, volume 40, pages 190–200. ACM, 2005. extras:luk05:pin.
[13] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.
[14] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.
[15] Dynamorio dynamic instrumentation tool platform. http://www.dynamorio.org/, 2017. (Accessed on 12/02/2017).
[16] Benjamin Serebrin and Daniel Hecht. Virtualizing performance counters. In *Proceedings of the European Conference on Parallel Processing*, pages 223–233, Bordeaux, France, August 2011.
[17] John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.
[18] Linux perf. http://www.brendangregg.com/perf.html, 2017. (Accessed on 11/19/2017).
[19] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. Barecloud: Bare-metal analysis-based evasive malware detection. In *USENIX Security Symposium (SP)*, pages 287–301, 2014.
[20] Vincent M Weaver and Sally A McKee. Can hardware performance counters be trusted? In *Proceedings of International Symposium on Workload Characterization (IISWC)*, pages 141–150. IEEE, 2008.
[21] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning.* MIT press, 2016.
[22] Pivotal Software Inc. Rabbitmq. http://www.rabbitmq.com/, 2017. (Accessed on 11/12/2017).
[23] Samba - opening windows to a wider world. https://www.samba.org/, 2017. (Accessed on 12/05/2017).
[24] bindfs. https://bindfs.org/, 2017. (Accessed on 12/05/2017).
[25] Paul J. Drongowski. An introduction to analysis and optimization with amd codeanalyst performance analyzer, 2008.
[26] Virustotal. Virustotal. https://www.virustotal.com/, 2017. (Accessed on 07/12/2017).
[27] Marcos Sebastián, Richard Rivera, Platon Kotzias, and Juan Caballero. Avclass: A tool for massive malware labeling. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 230–253. Springer, 2016.
[28] Futuremark. https://www.futuremark.com/, 2017. (Accessed on 11/15/2017).
[29] Performance: Python package index. https://pypi.python.org/pypi/performance/0.5.1, 2017. (Accessed on 11/30/2017).
[30] Ninite. https://ninite.com/, 2017. (Accessed on 11/15/2017).
[31] Npackd. https://npackd.appspot.com/, 2017. (Accessed on 11/15/2017).
[32] Android debug bridge. https://developer.android.com/studio/command-line/adb.html, 2017. (Accessed on 11/12/2017).
[33] Vincent M Weaver, Dan Terpstra, and Shirley Moore. Non-determinism and overcount on modern hardware performance counter implementations. In *Proceedings of International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 215–224. IEEE, 2013.
[34] S.S. Haykin. *Communication System.* Wiley Series in Management Series. John Wiley & Sons, 1983.
[35] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
[36] Yoav Freund and Robert E Schapire. A desicion-theoretic generalization of on-line learning and an application to boosting. In *European conference on computational learning theory*, pages 23–37. Springer, 1995.
[37] George H John. Robust decision trees: Removing outliers from databases. In *KDD*, pages 174–179, 1995.
[38] J Rennie, L Shih, J Teevan, and D Karger. Tackling the poor assumptions of naive bayes classifiers (pdf). ICML, 2003.
[39] Ron Kohavi et al. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, volume 14, pages 1137–1145. Stanford, CA, 1995.
[40] Adam Young and Moti Yung. Cryptovirology: Extortion-based security threats and countermeasures. In *Proceedings of Security and Privacy*, pages 129–140. IEEE, 1996.
[41] Kaspersky security bulletin 2016. review of the year. overall statistics for 2016. https://securelist.com/kaspersky-security-bulletin-2016-executive-summary/76858/. (Accessed on 12/10/2017).
[42] Cryptography reference (windows). https://msdn.microsoft.com/en-us/library/aa380256.aspx, 2017. (Accessed on 11/18/2017).