A Cautionary Tale about Detecting Malware Using Hardware Performance Counters and Machine Learning

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

Recent works propose to use Hardware Performance Counter (HPC) values with machine learning (ML) classification models for malware detection. However, measured HPC values and ML models cannot reliably distinguish malware from benignware. This is because of the semantic gap between the high-level malicious behavior and the low-level micro-architectural events. We run 962 benignware and 962 malware on our experimental setup, and show 83.39%, 84.84%, 83.59%, 75.01%, 78.75%, and 14.32% F1-score for Decision Tree, Random Forest, K Nearest Neighbors, Adaboost, Neural Net, and Naive Bayes, respectively. We perform 10-fold cross-validation on our models 1,000 times and show variations of the cross-validation results. Our analysis cautions the community about the shortcomings of using HPC and ML for detecting malware. In fact, after reading our original publication [1], many of the researchers have avoided these shortcomings in their follow-up research in using HPCs for malware detection. We believe that our efforts have made a positive influence on the security research community.

KEYWORDS

Malware Detection, Hardware Performance Counters, Machine Learning

ACM Reference Format:
Boyou Zhou, Anmol Gupta, Rasoul Jahanshahi, Manuel Egele, Ajay Joshi. 2021. A Cautionary Tale about Detecting Malware Using Hardware Performance Counters and Machine Learning . In *Proceedings of ACM Conference (Conference'17)*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 [2]) and 6 (on AMD [3]) counter registers dictates that HPCs can only monitor a small subset of these events at one time.

Under these constraints, previous works [4–7] leverage the measured HPC values to classify an unknown program as either benign or malicious. Previous works record data of labeled programs in time-series with a fixed frequency, use the HPC values in time-series to train various supervised machine learning models, and yield classifiers to 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. For example, both ransomeware and benignware use cryptographic APIs, but the ransomeware maliciously encrypt user files, while the benignware safeguards user information. One cannot distinguish between benignware and ransomeware based on the measured HPC values, because no HPC event can indicate the ownership of the API 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. 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.

We survey the existing literature in this field, and identify common traits that exhibit impractical setups and mis-interpretation 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.

Conference'17, July 2017, Washington, DC, USA
2021. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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

In this work, we build an experimental setup close to the user environment, and evaluate the fidelity of machine learning models. We run all experiments in a bare-metal environment instead of relying on virtualization techniques, since the sampling of virtualized HPC values is different from the sampling HPCs on a bare-metal system of a regular user. Previous works [4, 6, 7] test their machine learning models using measured HPC values from the same programs used during training (In §4, we refer to this approach as **TTA1**). This scenario would reflect all programs (benign and malicious) are known and labeled for training, which is absolutely unlikely, as millions of new malware samples are reported to Anti-Virus (AV) everyday. Thus, we test our models with measured HPC values from programs that have not been observed during training, which reflects a scenario that programs in the same category are available for training, but not the same program sample.

We perform 1,000 iterations of 10-fold cross-validations on 6 classifiers 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 [4, 6–9] 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://bit.ly/2SwYwPN

Since our original publication [1], many of the researchers have avoided the shortcomings that we have identified and made improvements in using HPCs for malware detection. We hope that our work can guide future research in HPCs and machine learning for malware detection in the right direction.

2 RELATED WORK AND MOTIVATION

Many previous works commonly utilize *sub-semantic features* in malware detection [4, 6–12] . 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" [10]. All these previous works have several

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

	Tool Choice	Experimental Setups		Event Choice	Data Division		Cross Validation				Machine Learning Models					# of Programs	Opensource	
Paper	**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)	Data Divided By Samples (TTA2 in § 4)	**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
[4]	⋄	⋄	●	⋄	●	⋄	●	⋄	⋄	⋄	●	●	●	●	⋄	●	⋄	3
[6]	⋄	⋄	●	⋄	●	●	⋄	⋄	●	⋄	●	●	●	⋄	⋄	●	⋄	2
[7]	⋄	●	⋄	●	●	⋄	●	⋄	⋄	⋄	⋄	⋄	⋄	⋄	⋄	●	⋄	4
[8]	⋄	●	⋄	●	⋄	●	●	⋄	⋄	⋄	⋄	⋄	⋄	⋄	⋄	●	⋄	3
[9]	⋄	●	⋄	●	●	⋄	⋄	●	⋄	⋄	●	⋄	⋄	⋄	⋄	●	⋄	4
[10]	●	●	⋄	⋄	⋄	●	⋄	●	⋄	⋄	⋄	⋄	⋄	●	⋄	⋄	⋄	2
[11]	●	●	⋄	⋄	⋄	●	⋄	●	⋄	⋄	⋄	●	⋄	⋄	●	⋄	⋄	3
[12]	●	●	⋄	⋄	⋄	●	⋄	●	⋄	⋄	⋄	●	⋄	⋄	●	⋄	⋄	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.

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)**
IV **No Cross-Validations or Insufficient Validations**
V **Few Data Samples**

Dynamic Binary Instrumentation (DBI) tools such as Intel's Pin [5, 13], QEMU [14], or Valgrind [15], can also extract *sub-semantic features*. Khasawneh et al. use Pin to monitor the instructions executed on virtual machines in their experimental setup [10–12]. 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 the default use-case for current anti-malware suites. We denote the drawbacks of DBI as **Drawback I** in Table 1.

While DBI is infeasible in an online detection system, other methods in sampling HPCs can also incur inaccurate measurements. A plethora of previous works run the evaluated programs on VMs [4, 9–12]. We chose to use bare-metal machine based on two observations. First, virtualizing HPCs is a challenge [16], as the measured virtualized HPC values are different from measured bare-metal HPC values [1]. Second, a regular user uses the bare-metal machines instead of the virtualized machines. 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 [17], previous works [4, 6–8] choose to maximize the measuring granularity by using HPCs without time-multiplexing. Previous works [4, 6, 10–12] have used empirical study to select 4 (Intel) or 6 (AMD) events for monitor, without providing a numerical analysis on how micro-architectural events are selected. In our experiments, we perform a Principal Component Analysis (PCA) based approach to select 6 micro-architectural events. After the selection of events, we use HPCs to track these 6 events, and transform the measured HPC values to examples in machine learning models, i.e. feature extraction. We then divide examples into training and testing datasets for machine learning models (training-and-testing split). Previous works [4, 6, 7, 9] have training-and-testing split based on the examples (TTA1 in § 4) that the testing dataset can have the same examples produced by programs in training dataset. 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 prevents machine learning models from overfitting[3]. We observe that there is no cross-validation in some of the previous works [4, 7, 8], while other works [9–12] present insufficient cross-validation, i.e. not every example in the dataset is validated, and none of these works reported the variations of the cross-validation results. 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 and compare against the results from previous works: Decision Tree (DT), Random Forest (RF), K Nearest Neighbors (KNN), Neural Nets (NN), Naive Bayes and AdaBoost.

A double decimal precision result, reported previously [4], should require at least 10,000 experiments, which is equivalent to more than 1,000 programs in the 10-fold cross-validations. 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) [4, 6–9]. 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. 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 the table, rows are various works on 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 text highlights drawbacks, and black text expresses 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.

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 [18]. We collected the data back from the worker nodes using a Samba [19] server on the master node. We used Bindfs [20] 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 [10–12]. We applied fixed-frequency time-based HPC sampling as the previous works [4].

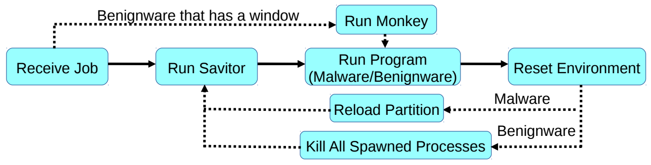
3.1 Malware and Benignware

For forming the set of malware, we downloaded 1,000 malware from Virustotal [21], 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* [22], our dataset consisted of 35 distinct malware families.

In order to collect benignware programs, we first installed all the packages and software from Futuremark [23], python performance module [24], ninite.com [25], and Npackd [26] on the worker nodes. 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

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



Benignware that has a window
Run Monkey
Receive Job
Run Savitor
Run Program (Malware/Benignware)
Reset Environment
Reload Partition
Malware
Benignware
Kill All Spawned Processes

Figure 1: Our workflow of benignware/malware experiments: The worker node receives the dispatched jobs of experiments from the master node. The worker node spawns a sampling process, and then the sampling process 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.

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.

3.2 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 1. 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.

3.2.1 Malware Experiment. We follow the steps in Figure 1 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, the sampling process 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.

Table 2: Description of the Selected Events [3]

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

3.2.2 Benignware Experiment. Similar to the malware experiments, benignware experiments also follow the workflow in Figure 1. 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, the sampling process 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 [27], 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 the sampling process finishes recording the measured HPC values, the system resets by killing any processes spawned during the experiments. Given that 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. To avoid *Curse of Dimensionality* [28], we applied Principal Components Analysis (PCA) to reduce the feature vector in our system. We ran each of the 7 programs from Futuremark Benchmarks on 130 micro-architectural events 32 times (**130×32×7**). From the results of these 7 programs, we selected 6 events with 2 eigenvectors that represent the binned results as our selected features (events listed in the Table 2), and generated the eigenvector matrix, denoted as $v_{192\times12}$, from the PCA. Four of the selected events in our experiments align with other works that do not provide any analysis of their selection of events [4, 6, 10–12]. We monitored the 6 events from Table 2 for our 962 benignware and 962 malware program samples. Due to page limit here, we do not provide our quantitative analysis to extract features from the measured HPC values of our selected micro-architectural events. One can find the detailed analysis in the our extended conference paper [1].

To have the same number of measurements on the same program samples, we run each benignware program and each malware program 32 times, and collect 61,568 (2×962×32)[4] measured HPC values (1,026 CPU hours). We sum the measured HPC values into 32 histogram bins for each of 6 events. Each example of histogram binned HPC values has 192 (6 *events* × 32 *bins*) features. By multiplying

[4]30,784 for benignware and 30,784 for malware

each example with the v eigenvector matrix, we reduce the dimensions from 192 (6 *events* × 32 *bins*) to 12 (6 *events* × 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, the input matrix $A_{30,784 \times 192}$ (30,784 examples and 192 features) of benignware or malware is transformed to lower-dimensional space as $A'_{30,784 \times 12}$ (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 for both benignware and malware experiments are as follows:

TTA1 Divide 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.

TTA2 Divide 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.

In **TTA1**, 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.

The 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: 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. The details related to the model configurations can be found in our extended conference paper [1].

5 EXPERIMENTAL RESULTS

In this section, we show the results of our experiments to detect malware using HPCs and contrast them with 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.

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. We train and test various machine learning models and determine the detection rates (precision, recall, and F1-score) with **TTA1** and **TTA2**. 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 2 shows the ROC curves and the AUCs of ROC for different machine learning models.

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. The detection rates are lower when using **TTA1** as compared to the scenario using **TTA1**.

Figure 2(b) shows the ROC curves and the AUCs of ROC for different machine learning models. 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 2(b), compared to 89.65%, 91.84%, 89.26%, 58.11%, 80.57%, and 84.41% using **TTA1** in Figure 2(a).

Demme et al. showed precision varying from 25% ∼ 100% [4] 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. 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 2(b). These results show that high detection rates and robustness in detection are over-estimated by some prior works due to division of data during training. In the next subsection, we show that the results presented in this subsection are not an exception.

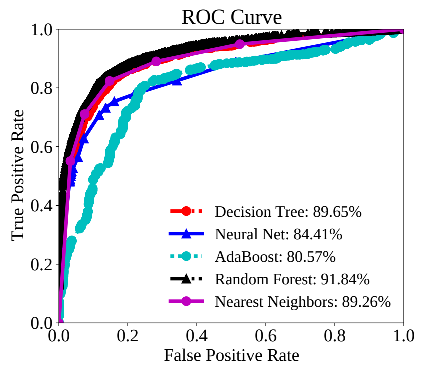
5.2 Cross-Validation

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, different training and testing splits [28]. 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

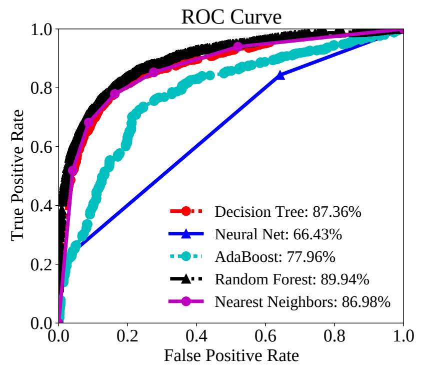
[5]False Discovery Rate $(F_+/(F_+ + T_+)$

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%

	TTA1				TTA2			
Models	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

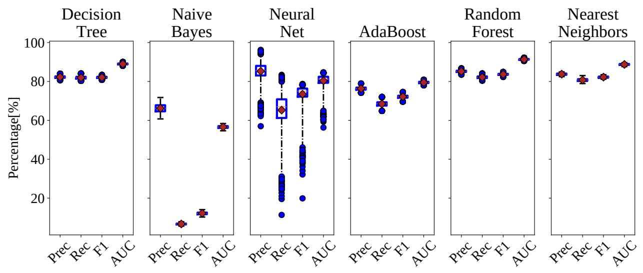


(a)

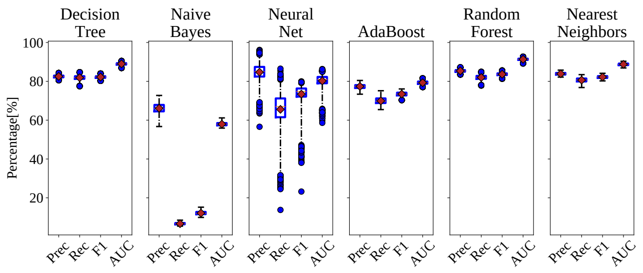


(b)

Figure 2: 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.



(a)



(b)

Figure 3: 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.

from cross-validations. The lack of proper cross-validation motivates us to further evaluate the machine learning models using cross-validation.

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. The difference between standard deviations in Figure 3(a) and Figure 3(b) is due to the unrealistic assumption that the programs in the training set appear in the testing dataset. Figure 3(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. The HPC measurements can be helpful for other security applications, such as the detection of low-level hardware attack, but the results from **TTA2** clearly show that using the results from **TTA1** can be misleading and prematurely draw the conclusion using HPC measurements and machine learning to differentiate between benignware and malware.

5.3 Ransomware

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

Ransomware is a malware that maliciously encrypts files and extorts users in exchange for the decryption keys [29]. We craft the malware by *simply* infusing Notepad++ with a ransomware. We modify the constructor of Notepad++ to iterate over a hardcoded directory, encrypt each file with a hardcoded password and a session key, and dump the content to another file in the same directory, with 5 seconds delay between each encryption. 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, and 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. In our malware example, the changes of HPC values caused by ransomware are overshadowed in the sampled values of HPCs when running Notepad++. The variation tolerance results in classifying the modified Notepad++ as benignware.

6 DISCUSSION

For our experiments, 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 [30]. By comparing the HPC values across different micro-architectures, Weaver et al. showed 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, because both benignware and malware function properly during experiments. For the reduction of dimensions, many other approaches can serve the same purpose as PCA. We used PCA in our designs as PCA is one of the most popular methods for reduction of dimensions.

The research regarding the use of HPC measurements for malware detection as well as debugging and profiling tools has grown rapidly since our original publication [1]. Similar to our research, multiple researchers studied the limits of HPC measurements [31–34] in line with the spirit of our research. Among these published works, Das et. al. [31] evaluated the ways how 41 prior works used HPC values and showed how various challenges can undermine the effectiveness of security applications. Basu et. al. [32] developed a framework to determine the probability of malware detection systems while monitoring HPC values at a pre-determined interval. Brasser et. al. [33] discussed multiple hardware-assisted security solutions and their respective limitations used by third party developers. Dinakarrao [34] introduced adversarial attacks on HPC-based malware detection systems. All these research works essentially re-iterate our cautionary tales of using HPC for malware detection. We also observed that researchers have acknowledged these shortcomings and have chosen to improve their respective analysis by adapting their systems to tackle the limits of HPC measurements. Researchers overcame the drawbacks we discussed in section 2, by utilizing bare-metal environments [35–38], implementing customized hardware [39–41] instead of HPCs, proper cross-validation [38], using ensemble models [42], and using other than HPC values to detect malicious behavior [43]. Wang et al. [35, 36] proposed a customized tool to overcome the problem of contaminating the HPC values from other processes. Basu et al. [43] developed embedded trace buffer (ETB) based malware detector to identify malicious behaviors. Ramos et al. [44] proposed a post-processing method to mitigate the effect of HPC drawbacks for modeling parallel applications. All these efforts in the security community make us believe that our work has had positive and profound influence on the security research in the last couple of years. We hoped that our work will guide future work in the area of using HPCs and machine learning for malware detection in the right direction of research.

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. By identifying the shortcomings in the prior approaches of using HPCs and machine learning for malware detection, we have guided the community in the right direction. Publications on malware detection using HPCs and ML after our original paper have shown that our paper has had positive and profound influence on security research. We hope that our efforts will continue to help the research community in the coming years.

REFERENCES

[1] Boyou Zhou, Anmol Gupta, Rasoul Jahanshahi, Manuel Egele, and Ajay Joshi. Hardware performance counters can detect malware: Myth or fact? In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 457–468, 2018.

[2] *Intel Itanium Architecture Software Developer's Manual.* Intel Corporation, 2010.

[3] *BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 10h-1Fh Processors.* Advanced Micro Devices, Inc., 2015.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[15] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[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] 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.

[18] Pivotal Software Inc. Rabbitmq. http://www.rabbitmq.com/, 2017. (Accessed on 11/12/2017).

[19] Samba - opening windows to a wider world. https://www.samba.org/, 2017. (Accessed on 12/05/2017).

[20] bindfs. https://bindfs.org/, 2017. (Accessed on 12/05/2017).

[21] Virustotal. Virustotal. https://www.virustotal.com/, 2017. (Accessed on 07/12/2017).

[22] 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.

[23] Futuremark. https://www.futuremark.com/, 2017. (Accessed on 11/15/2017).

[24] Performance: Python package index. https://pypi.python.org/pypi/performance/0.5.1, 2017. (Accessed on 11/30/2017).

[25] Ninite. https://ninite.com/, 2017. (Accessed on 11/15/2017).

[26] Npackd. https://npackd.appspot.com/, 2017. (Accessed on 11/15/2017).

[27] Android debug bridge. https://developer.android.com/studio/command-line/adb.html, 2017. (Accessed on 11/12/2017).

[28] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning.* MIT press, 2016.

[29] Adam Young and Moti Yung. Cryptovirology: Extortion-based security threats and countermeasures. In *Proceedings of Security and Privacy*, pages 129–140. IEEE, 1996.

[30] 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.

[31] Sanjeev Das, Jan Werner, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. Sok: The challenges, pitfalls, and perils of using hardware performance counters for security. In *Proceedings of the 40th Symposium on Security and Privacy (SP)*, pages 20–38. IEEE, 2019.

[32] Kanad Basu, Prashanth Krishnamurthy, Farshad Khorrami, and Ramesh Karri. A theoretical study of hardware performance counters-based malware detection. *Proceddings of the IEEE Transactions on Information Forensics and Security (TIFS)*, 15:512–525, 2019.

[33] Ferdinand Brasser, Lucas Davi, Abhijitt Dhavlle, Tommaso Frassetto, Sai Manoj Pudukotai Dinakarrao, Setareh Rafatirad, Ahmad-Reza Sadeghi, Avesta Sasan, Hossein Sayadi, Shaza Zeitouni, et al. Special session: Advances and throwbacks in hardware-assisted security. In *Proceedings of the 3rd International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pages 1–10. IEEE, 2018.

[34] Sai Manoj Pudukotai Dinakarrao, Sairaj Amberkar, Sahil Bhat, Abhijitt Dhavlle, Hossein Sayadi, Avesta Sasan, Houman Homayoun, and Setareh Rafatirad. Adversarial attack on microarchitectural events based malware detectors. In *Proceedings of the 56th Annual Design Automation Conference (DAC)*, pages 1–6, 2019.

[35] Han Wang, Hossein Sayadi, Avesta Sasan, Setareh Rafatirad, and Houman Homayoun. Dreal: Detecting side-channel attacks at real-time using low-level hardware features. Technical report, University of California Davis United States, 2020.

[36] Han Wang, Hossein Sayadi, Setareh Rafatirad, Avesta Sasan, and Houman Homayoun. Scarf: Detecting side-channel attacks at real-time using low-level hardware features. In *2020 IEEE 26th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pages 1–6. IEEE, 2020.

[37] H. Wang, H. Sayadi, G. Kolhe, A. Sasan, S. Rafatirad, and H. Homayoun. Phased-guard: Multi-phase machine learning framework for detection and identification of zero-day microarchitectural side-channel attacks. In *Proceedings of the 38th International Conference on Computer Design (CCD)*, 2020.

[38] Rashid Tahir, Sultan Durrani, Faizan Ahmed, Hammas Saeed, Fareed Zaffar, and Saqib Ilyas. The browsers strike back: Countering cryptojacking and parasitic miners on the web. In *Proceedings of the IEEE INFOCOM Conference on Computer Communications (CCC)*, pages 703–711, 2019.

[39] Leila Delshadtehrani, Sadullah Canakci, Boyou Zhou, Schuyler Eldridge, Ajay Joshi, and Manuel Egele. Phmon: A programmable hardware monitor and its security use cases. In *Proceedings of the 29th USENIX Security Symposium USENIX Security*, 2020.

[40] Leila Delshadtehrani, Schuyler Eldridge, Sadullah Canakci, Manuel Egele, and Ajay Joshi. Nile: A programmable monitoring coprocessor. *Proceedings of the IEEE Computer Architecture Letters (CAL)*, 2017.

[41] Sadullah Canakci, Leila Delshadtehrani, Boyou Zhou, Ajay Joshi, and Manuel Egele. Efficient context-sensitive cfi enforcement through a hardware monitor. In *Proceedings of the 17th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 259–279. Springer, 2020.

[42] Yusheng Dai, Hui Li, Yekui Qian, Ruipeng Yang, and Min Zheng. Smash: a malware detection method based on multi-feature ensemble learning. *Proceddings of the IEEE Access*, 7:112588–112597, 2019.

[43] Kanad Basu, Rana Elnaggar, Krishnendu Chakrabarty, and Ramesh Karri. Preempt: Preempting malware by examining embedded processor traces. In *Proceedings of the 56th Design Automation Conference (DAC)*, pages 1–6. IEEE, 2019.

[44] Vitor Ramos, Carlos Valderrama, Samuel Xavier de Souza, and Pierre Manneback. An accurate tool for modeling, fingerprinting, comparison, and clustering of

parallel applications based on performance counters. In *Proceedings of the 33th International Parallel and Distributed Processing Symposium Workshops (IPDPS)*, pages 797–804, 2019.