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

Boyou Zhou, Anmol Gupta, Rasoul Jahanshahi, Manuel Egele, and Ajay Joshi
Electrical and Computer Engineering Department, Boston University, Boston, MA 02215 USA

Editor's notes:
This article revisits the literature around the use of hardware performance counters for malware detection, highlighting discrepancies, and providing a retrospection of challenges for future research.
—Rosario Cammarota, Intel Labs
—Francesco Regazzoni, University of Amsterdam and Università della Svizzera Italiana

Distinguishing between malicious and benign software has remained one of the biggest challenges facing computer security over recent decades. As signature-based antivirus (AV) scanners are easily thwarted by polymorphic malware, most commercial and academic antimalware 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 advanced micro devices (AMD) [2]) counter registers dictates that HPCs can monitor only 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. 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

Digital Object Identifier 10.1109/MDAT.2021.3063338
Date of publication: 19 March 2021; date of current version: 20 May 2021.

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 from 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 ransomware and benignware use cryptographic application programming interfaces (APIs), but the ransomware maliciously encrypt user files, while the benignware safeguards user information. One cannot distinguish between benignware and ransomware 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.

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 [3], [5], [6] test their machine learning models using measured HPC values from the same programs used during training (in the "Machine learning models" section, 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 AV every day. 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 1000 iterations of tenfold cross-validations on six 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 straightforward 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 ("Related work and motivation" section).
- We perform thorough experiments with a program count that exceeds prior works [3], [5]–[8] by a factor of 2× ~ 3×, and the number of experiments in cross-validations that is three orders of magnitude more than previous works.
- We train and test data set 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 ("Experimental results" section).
- 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 [9], many of the researchers have avoided the shortcomings that we

[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.

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.

Related work and motivation

Many previous works commonly utilize *subsemantic features* in malware detection [3], [5]–[8], [10]–[12]. Ozsoy et al. [10] defined the term *subsemantic features* as "micro-architectural information about an executing program that does not require modeling or detecting program semantics." All these previous works have several drawbacks to a great extent. We categorize the drawbacks that we observed into the following classes.

- Dynamic binary instrumentation (DBI).
- Virtual machines (VMs).
- Division of data by traces (TTA1 in the "Machine learning models" section).
- No cross-validations or insufficient validations.
- Few data samples.

DBI tools, such as Intel's Pin [4], [13], Quick EMUlator (QEMU) [14], or Valgrind [15], can also extract *subsemantic features*. Khasawneh et al. [11] and [12] use pin to monitor the instructions executed on VMs in their experimental setup [10]. Although DBI can extract subsemantic features that are not available from HPCs, it 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 antimalware 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 lead to inaccurate measurements. A plethora of previous works run the evaluated programs on VMs [3], [8], [10]–[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 [9]. Second, a regular user uses the bare-metal machines instead of the virtualized machines. These observations motivate our experimental setup ("Experimental setup" section) 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 [3], [5]–[7] choose to maximize the measuring granularity by using HPCs without time-multiplexing. Previous works [3], [5], [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 six micro-architectural events. After the selection of events, we use HPCs to track these six 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 data sets 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 the "Machine learning models" section) that the testing data set can have

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 diamonds (◇) indicate the nonuse 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.

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)	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
[3]	◇	◇	•	◇	•	◇	•	◇	◇	◇	•	•	•	•	◇	•	◇	3
[5]	◇	◇	•	◇	•	•	◇	◇	•	◇	•	•	•	◇	◇	•	◇	2
[6]	◇	•	◇	•	•	◇	•	◇	◇	◇	◇	◇	◇	◇	◇	•	◇	4
[7]	◇	•	◇	•	◇	•	•	◇	◇	◇	◇	◇	◇	◇	◇	•	◇	3
[8]	◇	•	◇	•	•	◇	◇	•	◇	◇	•	◇	◇	◇	◇	•	◇	4
[9]	•	•	◇	◇	◇	•	◇	•	◇	◇	◇	◇	◇	•	◇	◇	◇	2
[10]	•	•	◇	◇	◇	•	◇	•	◇	◇	◇	•	◇	◇	•	◇	◇	3
[11]	•	•	◇	◇	◇	•	◇	•	◇	◇	◇	•	◇	◇	•	◇	◇	3
★	◇	◇	•	•	•	•	◇	◇	◇	•	•	•	•	•	•	◇	•	-

the same examples produced by programs in training data set. In real-life, it is unlikely that the offline training data set 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 1000 repetitions of ten-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 [3], [6], [7], while other works [8], [10]–[12] present insufficient cross-validation, i.e., not every example in the data set 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. 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 [3], should require at least 10,000 experiments, which is equivalent to more than 1000 programs in the tenfold cross-validations. As a result, we consider the works with fewer than 1000 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. 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 represent 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 diamonds (⋄) indicate the nonuse 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 two drawbacks in each work.

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

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 [3].

Malware and benignware

For forming the set of malware, we downloaded 1000 malware from Virustotal [21], and performed a test run of those 1000 malware on worker nodes. After the test run, we identified 962 malware which could run for more than 1 min and used them in our malware experiments. According to *AVClass tool* [22], our data set consisted of 35 distinct malware families.

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 data set. We avoided the complication of reinstallation by excluding all the executable program files with "uninstall" in their names. We performed a test run of all these programs, and selected 1382 benignware that could run for 1 min.

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 data set, 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.

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.

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. 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.

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

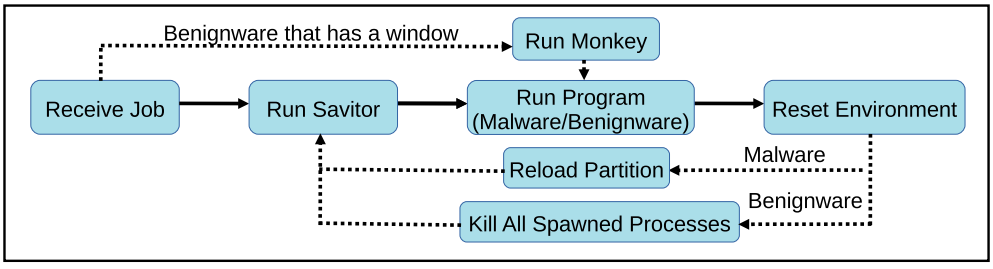

Figure 1. Our workflow of benignware/malware experiments: the worker node receives the dispatched jobs of experiments from the master node. The worker node spawns a sampling process, and then the sampling process runs the target process (benignware/malware). The dotted arrow (- - →) means that the action 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.

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.

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 PCA to reduce the feature vector in our system. We ran each of the seven programs from Futuremark Benchmarks on 130 micro-architectural events 32 times (**130 × 32 × 7**). From the results of these seven programs, we selected six events with two eigenvectors that represent the binned results as our selected features (events listed in Table 2), and generated the eigenvector matrix, denoted as $v192 \times 12$, 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 [3], [5], [10]–[12]. We monitored the six 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 our extended conference paper [9].

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

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 $(2 \times 962 \times 32)$[4] measured HPC values (1026 CPU hours). We sum the measured HPC values into 32 histogram bins for each of six events. Each example of histogram binned HPC values has 192 $(6\ events \times 32\ bins)$ features. By multiplying each example with the v eigenvector matrix, 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, 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 data sets (training-and-testing split). In our experiments, we consider two training-and-testing approaches (TTA) to divide our data set 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 data set and 3078 traces (10% of 30,784 traces) as testing data set.

TTA2: Divide 962 programs with a split of 90:10 ratio, resulting in traces of 866 programs (90% of programs) as training data set and traces of 96 programs (10% of programs) as testing data set.

In **TTA1**, the traces resulting from the same program sample can appear in both training and testing data sets. 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 every day, 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

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

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

models, we perform tenfold cross-validations 1000 times. For each tenfold cross-validation, we randomly shuffle the data set to ensure difference across 1000 rounds. In each tenfold cross-validation, each example in the data set is used in training nine 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 1000 tenfold 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 [9].

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 AUC in receiver operating characteristic (ROC) plots. We use the positive label to denote malware and the negative label to denote benignware.

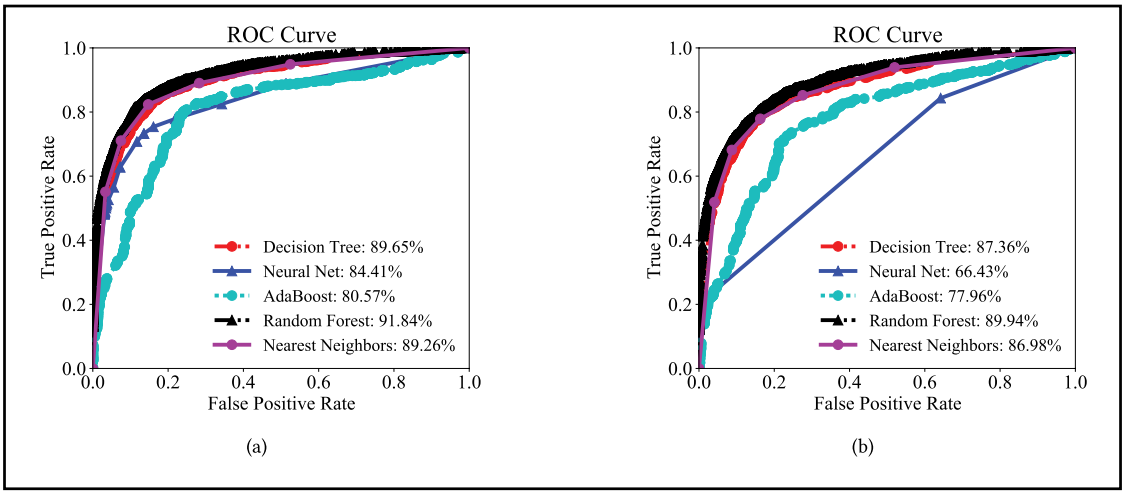

Figure 2. ROC curve of five models. (a) AUC of DT, NN, AdaBoost, RF, and KNN using (TTA1) is 89.65%, 84.41%, 80.57%, 91.84%, and 89.26%, respectively. (b) AUC of DT, NN, AdaBoost, RF, and KNN using (TTA2) is 87.36%, 66.43%, 77.96%, 89.94%, and 86.98%, respectively.

Malware detection

In this section, we report the detection rates (precision, recall, and F1-score) with two 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 the "Machine learning models" section. 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 2b 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 2b, compared to 89.65%, 91.84%, 89.26%, 58.11%, 80.57%, and 84.41% using **TTA1** in Figure 2a.

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. 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 2b. 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.

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

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

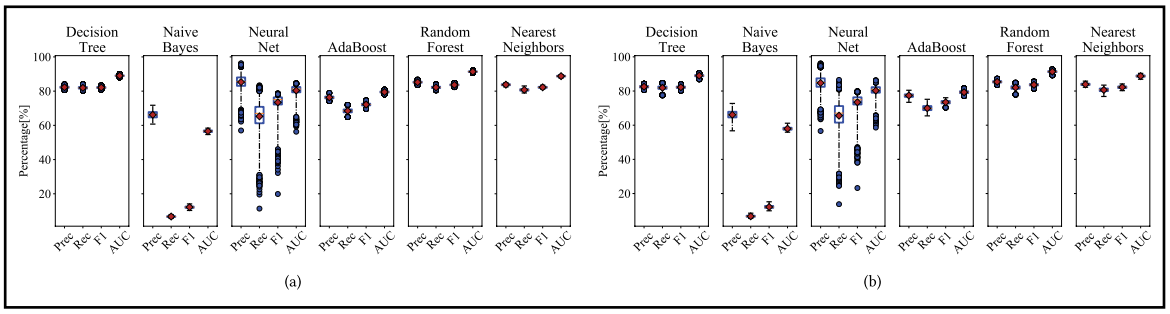

Figure 3. Box plots of distributions of tenfold cross-validation experiments using (a) TTA1 and (b) TTA2. Red diamonds are means, and blue box corresponds to cross-validation experiment results that lie between 25 and 75 percentiles. The whiskers (the short, horizontal lines outside the blue box) represent 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 AUC in ROC. These tenfold cross-validation experiments show that we cannot achieve 100% malware detection accuracy.

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%, and 12.21%, compared to 82.17%, 83.75%, 82.28%, 74%, 72.27%, and 12.15% using **TTA1**, respectively. In DT, RF, KNN, NN, Ad aBoost, and Naive Bayes models, the mean of distributions of F1-scores using **TTA2** are 2.145%, 2.336%, 2.248%, 14.88%, 3.29%, and 2.611%, compared to 1.416%, 1.326%, 1.388%, 13.2%, 2.365%, and 2.392% using **TTA1**, respectively. Comparing the results using **TTA1** and **TTA2**, the standard deviations of DT, RF, KNN, NN, AdaBoost, and Naive Bayes models increased by 1.515×, 1.762×, 1.62×, 1.127×, 1.391×, and 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 3a and b is due to the unrealistic assumption that the programs in the training set appear in the testing data set. Figure 3b presents the results where the malicious program is not included in the training data set. 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**. 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.

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-s delay* between each encryption. We measure the values of HPCs for modified Notepad++ in our experimental setup ("Experimental setup" section). 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, 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.

Discussion

For our experiments, we run Windows 7 32-bit operating system on AMD 15h family Bulldozer micro-architecture machine. Weaver et al. [30] performed extensive studies investigating the determinism of the measured HPC values in various micro-architectures. 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 [9]. 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 predetermined interval. Brasser et al. [33] discussed multiple hardware-assisted security solutions and their respective limitations used by third party developers. Dinakarrao et al. [34] introduced adversarial attacks on HPC-based malware detection systems. All these research works essentially reiterate 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 the "Related work and motivation" section, 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] and [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.

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 2–3 and the experiment numbers in cross-validation to three 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 1323 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] *Intel Itanium Architecture Software Developer's Manual,* Intel Corp., Mountain View, CA, USA, 2010.

[2] *BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 10h-1Fh Processors,* Adv. Micro Devices, Santa Clara, CA, USA, 2015.

[3] J. Demme et al., "On the feasibility of online malware detection with performance counters," in *Proc. 40th Annu. Int. Symp. Comput. Archit. (ISCA)*, 2013, p. 559.

[4] H. Patil et al., "Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation," in *Proc. 37th Int. Symp. Microarchit. (MICRO)*, 2004, pp. 81–92.

[5] M. Kazdagli, V. J. Reddi, and M. Tiwari, "Quantifying and improving the efficiency of hardware-based mobile malware detectors," in *Proc. 49th Int. Symp. Microarchit. (MICRO)*, 2016, pp. 1–13.

[6] X. Wang et al., "Hardware performance counter-based malware identification and detection with adaptive compressive sensing," *Trans. Archit. Code Optim.*, vol. 13, no. 1, pp. 1–23, 2016.

[7] A. Tang, S. Sethumadhavan, and S. J. Stolfo, "Unsupervised anomaly-based malware detection using hardware features," in *Proc. Int. Workshop Recent Adv. Intrusion Detection (RAID)*, 2014, pp. 109–129.

[8] B. Singh et al., "On the detection of kernel-level rootkits using hardware performance counters," in *Proc. 17th Asia Conf. Comput. Commun. Secur. (AsiaCCS)*, 2017, pp. 483–493.

[9] B. Zhou et al., "Hardware performance counters can detect malware: Myth or fact?" in *Proc. Asia Conf. Comput. Commun. Secur.,* 2018, pp. 457–468.

[10] M. Ozsoy et al., "Malware-aware processors: A framework for efficient online malware detection," in *Proc. 21st Int. Symp. High Perform. Comput. Archit. (HPCA)*, 2015, pp. 651–661.

[11] K. N. Khasawneh et al., "Ensemble learning for low-level hardware-supported malware detection," in *Proc. Int. Workshop Recent Adv. Intrusion Detection (RAID)*, 2015, pp. 3–25.

[12] K. N. Khasawneh et al., "RHMD: Evasion-resilient hardware malware detectors," in *Proc. 50th Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, Oct. 2017, pp. 315–327.

[13] C.-K. Luk et al., "Pin: Building customized program analysis tools with dynamic instrumentation," *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 190–200, 2005.

[14] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proc. USENIX Annu. Tech. Conf., FREENIX Track,* 2005, pp. 41–46.

[15] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," *ACM SIGPLAN Notices,* vol. 42, no. 6, pp. 89–100, 2007.

[16] B. Serebrin and D. Hecht, "Virtualizing performance counters," in *Proc. Eur. Conf. Parallel Process., Bordeaux*, France, Aug. 2011, pp. 223–233.

[17] V. M. Weaver and S. A. McKee, "Can hardware performance counters be trusted?" in *Proc. Int. Symp. Workload Characterization (IISWC),* 2008, pp. 141–150.

[18] Pivotal Software Inc. (2017). *Rabbitmq*. Accessed: Nov. 12, 2017. [Online]. Available: http://www.rabbitmq.com/

[19] (2017). *Samba Opening Windows to a Wider World*. Accessed: Dec. 5, 2017. [Online]. Available: https://www.samba.org/

[20] (2017). *Bindfs*. Accessed: Dec. 5, 2017. [Online]. Available: https://bindfs.org/

[21] Virustotal. (2017). *Virustotal.* Accessed: Jul. 12, 2017. [Online]. Available: https://www.virustotal.com/

[22] M.Sebastián et al., "AVCLASS: A tool for massive malware labeling," in *Proc. Int. Symp. Res. Attacks, Intrusions, Defenses,* 2016, pp. 230–253.

[23] (2017). *Futuremark*. Accessed: Nov. 15, 2017. [Online]. Available: https://www.futuremark.com/

[24] (2017). *Performance: Python Package Index*. Accessed: Nov. 30, 2017. [Online]. Available: https://pypi.python.org/pypi/performance/0.5.1

[25] (2017). *Ninite*. Accessed: Nov. 15, 2017. [Online]. Available: https://ninite.com/

[26] (2017). *Npackd*. Accessed: Nov. 15, 2017. [Online]. Available: https://npackd.appspot.com/

[27] (2017). *Android Debug Bridge.* Accessed: Nov. 12, 2017. [Online]. Available: https://developer.android.com/studio/command-line/adb.html

[28] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016.

[29] A. Young and M. Yung, "Cryptovirology: Extortion-based security threats and countermeasures," in *Proc. Secur. Privacy,* 1996, pp. 129–140.

[30] V. M. Weaver, D. Terpstra, and S. Moore, "Non-determinism and overcount on modern hardware performance counter implementations," in *Proc. Int. Symp. Perform. Anal. Syst. Softw. (ISPASS),* 2013, pp. 215–224.

[31] S. Das et al., "SoK: The challenges, pitfalls, and perils of using hardware performance counters for security," in *Proc. 40th Symp. Secur. Privacy (SP)*, 2019, pp. 20–38.

[32] K. Basu et al., "A theoretical study of hardware performance counters-based malware detection," *IEEE Trans. Inf. Forensics Security*, vol. 15, no. 15, pp. 512–525, Jun. 2019.

[33] F. Brasser et al., "Special session: Advances and throwbacks in hardware-assisted security," in *Proc. 3rd Int. Conf. Compil., Archit. Synth. Embedded Syst. (CASES)*, 2018, pp. 1–10.

[34] S. M. P. Dinakarrao et al., "Adversarial attack on microarchitectural events based malware detectors," in *Proc. 56th Annu. Design Autom. Conf. (DAC)*, 2019, pp. 1–6.

[35] H. Wang et al., "Dreal: Detecting side-channel attacks at real-time using low-level hardware features," Univ. California at Davis, Davis, CA, USA, Tech. Rep. AD1101633, 2020.

[36] H. Wang et al., "SCARF: Detecting side-channel attacks at real-time using low-level hardware features," in Proc. *IEEE 26th Int. Symp. On-Line Test. Robust Syst. Design (IOLTS)*, Jul. 2020, pp. 1–6.

[37] H. Wang et al., "Phased-guard: Multi-phase machine learning framework for detection and identification of zero-day microarchitectural side-channel attacks," in *Proc. 38th Int. Conf. Comput. Design (CCD),* 2020, pp. 648–655.

[38] R. Tahir et al., "The browsers strike back: Countering cryptojacking and parasitic miners on the Web," in *Proc. IEEE Conf. Comput. Commun. (CCC INFOCOM),* Apr./May 2019, pp. 703–711.

[39] L. Delshadtehrani et al., "PHMon: A programmable hardware monitor and its security use cases," in Proc. 29th *USENIX Secur. Symp. (USENIX Security)*, 2020, pp. 807–824.

[40] L. Delshadtehrani et al., "Nile: A programmable monitoring coprocessor," *IEEE Comput. Archit. Lett.,* vol. 17, no. 1, pp. 92–95, Jan./Jun. 2018.

[41] S. Canakci et al., "Efficient context-sensitive CFI enforcement through a hardware monitor," in *Proc. 17th Int. Conf. Detection Intrusions Malware, Vulnerability Assessment (DIMVA)*, 2020, pp. 259–279.

[42] Y. Dai et al., "SMASH: A malware detection method based on multi-feature ensemble learning," *IEEE Access,* vol. 7, pp. 112588–112597, 2019.

[43] K. Basu et al., "PREEMPT: PReempting malware by examining embedded processor traces," in *Proc. 56th Design Autom. Conf. (DAC),* Jun. 2019, pp. 1–6.

[44] V. Ramos et al., "An accurate tool for modeling, fingerprinting, comparison, and clustering of parallel applications based on performance counters," in *Proc. 33rd Int. Parallel Distrib. Process. Symp. Workshops (IPDPS),* 2019, pp. 797–804.

Boyou Zhou is with Amazon.com Inc. His research interest includes hardware-assisted software security. Zhou has a BS from Southeast University, Nanjing, China, and a PhD from the Electrical and Computer Engineering Department, Boston University, Boston, MA, USA, in software security.

Anmol Gupta is with Analog Devices Inc. His research interests include computer architecture, embedded system, AI/ML on edge, low-power mixed-signal SoC, DSP, audio processing, cybersecurity, hardware modulation using system Verilog and FPGAs. Gupta has an MS in computer engineering from Boston University, Boston, MA, USA (2017).

Rasoul Jahanshahi is a Graduate Student Research Assistant and currently pursuing a PhD with the Electrical and Computer Engineering Department, Boston University, Boston, MA, USA. His current research interest includes security aspects of PHP web applications. He is a member of the Boston University Security Lab.

Manuel Egele is an Assistant Professor with the Electrical and Computer Engineering Department, Boston University, Boston, MA, USA. He is a Co-Director of the Boston University Security Lab, where his current research interests include practical security aspects of commodity and mobile systems. He is a member of IEEE and ACM.

Ajay Joshi is an Associate Professor with the Electrical and Computer Engineering Department, Boston University, Boston, MA, USA. His research interests include VLSI design and emerging device technologies including silicon photonics and memristors. Joshi has a PhD in electrical and computer engineering from Georgia Institute of Technology, Atlanta, GA, USA.

■ Direct questions and comments about this article to Boyou Zhou, Electrical and Computer Engineering Department, Boston University, Boston, MA 02215 USA; bobzhou@bu.edu.