Analyzing Hardware Based Malware Detectors

Nisarg Patel, Avesta Sasan, Houman Homayoun
{npatel33, asasan, hhomayou} @gmu.edu

Department of Electrical and Computer Engineering
George Mason University
Fairfax, VA 22030

ABSTRACT

Detection of malicious software at the hardware level is emerging as an effective solution to increasing security threats. Hardware based detectors rely on Machine Learning(ML) classifiers to detect malware-like execution pattern based on Hardware Performance Counters(HPC) information at runtime. The effectiveness of these learning methods mainly relies on the information provided by expensive-to-implement limited number of HPC. This paper is the first attempt to thoroughly analyze various robust machine learning methods to classify benign and malware applications. Given the limited availability of HPC the analysis results help guiding architectural decision on what hardware performance counters are needed most to effectively improve ML classification accuracy. For software implementation we fully implemented these classifier at OS Kernel to understand various software overheads. The software implementation of these classifiers are found to be relatively slow with the execution time in the range of milliseconds, order of magnitude higher than the latency needed to capture malware at runtime. This is calling for hardware accelerated implementation of these algorithms. For hardware implementation, we have synthesized the studied classifier models on FPGA to compare various design parameters including logic area, power, and latency. The results show that while complex ML classifier such as MultiLayerPerceptron and logistics are achieving close to 90% accuracy, after taking into consideration their implementation overheads, they perform worst in terms of PDP, accuracy/area and latency compared to simpler but slightly less accurate rule based and tree based classifiers. Our results further show OneR to be the most cost-effective classifier with more than 80% accuracy and fast execution time of less than 10ns, achieving highest accuracy per logic area, while mainly relying on only a single branch-instruction HPC information.

1. INTRODUCTION

Malware is a piece of software which is used by attacker to perform various malicious activities from stealing information and performing DoS attack to gaining root access. Such malware threats are increasing rapidly. According to McAfee threats report[16] published in March 2016, 42 million malware samples have been recorded in the last quarter of 2015 alone with the rate of 316 new threats every minute. Recent proliferation of computing devices in the form of mobile devices and Internet-Of-Things have made effective detection of malware a great urgent challenge to be addressed.

Anti-virus(AV) software can detect and remove such harmful programs, but it has several significant drawbacks. First, traditional AV software relies on static signature based detection in order to detect malware. Such detection mechanism search for suspicious byte patterns in the program. Attacker can deceive AV software by programming malware in such a way that its signature appears as a benign software. Second, AV software are prone to exploits like any other software which can ultimately compromise protection if exploited. Third, AV softwares are slow and resource hungry. Situation becomes worse for metamorphic viruses, as effective detection of them is an NP-complete problem[18]. Behavior based detection[11] method uses dynamic aspects such as system call traces, control flow graphs, and data flow graphs, which can overcome few disadvantages of static based detection. In spite of that, being a software based solution, behavior based detection is still vulnerable to exploitation.

Recently, security researchers have shifted their attention to hardware based solutions. Demme et al.[5] showed that using supervised ML classifiers on collected Hardware Performance Counters(HPC) traces of both malware and benign programs, the running applications can be classified with high level of accuracy. Hardware based detectors offer fast online detection, efficiency in resource utilization, and invulnerability from getting infected by attackers which make them suitable for mitigating newer threats. However, there are several design challenges with hardware based detectors including having the capability of online monitoring of HPC, low false positives, small logic area and power overhead for implementation on processor, and small detection latency which includes reading HPC and running ML classifiers.

This paper is the first attempt to thoroughly analyze and understand various machine learning methods to classify benign and malware applications. We test various design parameters of several learning methods in terms of accuracy, hardware implementation cost such as power, area, and latency, as well as software implementation cost. This helps the designer to understand and navigate the tradeoffs between several design parameters offered by each learning algorithms. Hardware performance counters are finding their way into commodity processor architectures such ARM based and Intel based processors. However, they are very limited in numbers and can only capture few processor runtime microarchitectural behaviors at a time. For instance, a high-end Intel Xeon server has only 6 registers to monitor microarchitectural behavior at run-time. For mobile Intel Atom processor, this is even lower and there are only 4 registers available. Given the high implementation cost of on-chip HPC and their limited availability, the results of this research will also help in making architectural decision on what HPC are needed to implement to most effectively

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

DAC '17, June 18-22, 2017, Austin, TX, USA
© 2017 ACM. ISBN 978-1-4503-4927-7/17/06…$15.00
DOI: http://dx.doi.org/10.1145/3061639.3062202

cpu-cycles	instructions	cache-references	cache-misses
branch-instructions	branch-misses	bus-cycles	ref-cycles
cpu-clock	task-clock	page-faults	context-switches
cpu-migrations	minor-faults	major-faults	alignment-faults
dummy	emulation-faults	L1-dcache-loads	L1-dcache-load-misses
L1-dcache-store-misses	L1-dcache-prefetch-misses	L1-icache-load-misses	LLC-loads
LLC-load-misses	LLC-stores	LLC-store-misses	LLC-prefetches
LLC-prefetch-misses	dTLB-loads	dTLB-load-misses	dTLB-stores
dTLB-store-misses	iTLB-loads	iTLB-load-misses	branch-loads
branch-load-misses	node-loads	node-load-misses	node-stores
node-store-misses	node-prefetches	node-prefetch-misses	L1-dcache-stores

Table 1: List of HPC events under PERF

improve ML classifiers accuracy.

In this paper, we collect HPC traces from execution of malware and benign applications in Linux. For software implementation, we fully implement several robust ML classifiers at OS Kernel level to understand various software overheads including the time to read the HPC and the time to execute the classifiers. We also report the accuracy of these ML classifiers for different number of architectural events. The results for software implementation of most accurate classifiers are found to be in the range of milliseconds, orders of magnitude higher than the response time needed to detect malware. Note that most malware are small pieces of codes with a execution time ranging from microseconds to milliseconds. It is therefore important to capture them within this execution range, to prevent them from corrupting the system. Such high overhead of software implementation of ML Classifiers is calling for a hardware accelerated implementation. In response, in this paper, we fully implement and evaluate hardware accelerated ML classifiers on FPGA and measure their power, latency, and area overhead. By comparing their area, accuracy, latency, and power, we show various implementation trade-offs to find out which ML classifiers are more suited for hardware implementation. The rest of this paper is organized as follows. We provide background on performance counters available under Linux in Section 2. We show procedure of collecting performance counters in detail in Section 3. We perform offline learning and testing of collected data in Section 4. We discuss software implementation of classifier models in Section 5. We discuss hardware implementation of classifier models in Section 6. Section 7 discusses the related work and Section 8 concludes the work.

2. BACKGROUND

Hardware Performance Counters(HPC) are special purpose registers available in modern microprocessors which keep track of different microarchitectural events. The main purpose of HPC is to analyze and tune architectural level performance of running applications [14, 13, 15]. While HPC are finding their ways in various processor platforms from high-performance to low power embedded, they are limited in the number of microarchitectural events that can be captured simultaneously. This is mainly due to limited number of physical registers on the processor chip which are expensive to implement. Recently, application areas of HPC are grown from mere performance analysis to detecting firmware modification in embedded systems [20], estimating complete system power [3], and detection of software malware [5] or even hardware trojans [21, 22]. Typical HPC events available under Linux *Perf* tool for Intel Haswell processors are shown in Table 1.

We are using HPC to collect execution traces for all available microarchitectural events by executing collected malware and benign applications in an isolated environment. If two different programs are executed on CPU, they generate different performance counter traces. HPC trace of branch-instructions is shown in Figure 1 for a normal and a malware

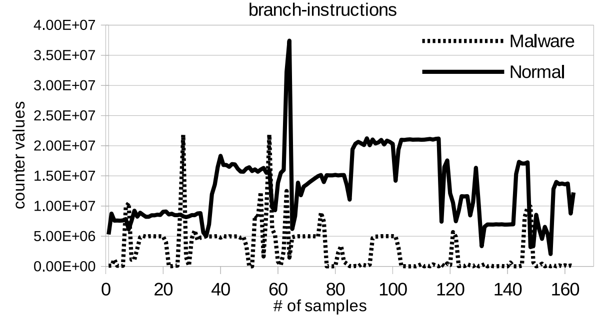

Figure 1: HPC comparison for branch-instructions

application. The results show substantial different traces for the branch instruction behavior for a malware compared to a normal application. Using this observation, malware can also be separated from normal applications by its different HPC values. Our goal is to learn malware behavior with collected HPC of various applications (including malware and normal) using supervised Machine Learning(ML) methods. Detailed performance counters collection procedure is discussed in the next section.

3. DATA COLLECTION AND DATA SET

We perform all data collections on a machine with an Intel Haswell Core i5-4590 CPU running Ubuntu 14.04 with Linux 4.4 Kernel. There are different methods for capturing HPC data including 1) Reading Model-specific Registers (MSRs) directly, 2)Building kernel module for sampled collection, and 3) Using different utilities available under Linux such as *Perf*, *PAPI*, and *Perfctr*. For this work we use *Perf*. *Perf* uses *perf_event_open* function call behind the scene which can measure multiple events at a time. We are using 52 benign applications and 57 malware applications for performance counters data collection. Benign applications consist of mibench benchmark suite[7], Linux system programs, browsers, text editors, and word processor. For malware applications, Linux malware are collected from virustotal.com [1]. Malware applications include Linux ELFs, python scripts, perl scripts, and bash scripts, which are created to do malicious activities.

HPC data are collected by running both malware and benign applications in isolated environment called Linux Containers(LXC) [9]. LXC is an operating system level virtualization which shares same kernel with the host operating system. The reason of using Linux Containers over other commonly available virtualization platforms such as VMWare or VirtualBox is that it provides access to all actual HPC

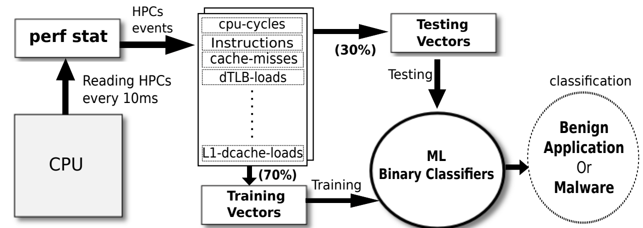

Figure 2: Offline training and detection procedure

{cpu-cycles, instructions, cache-references, ... , class}
{ 1870076 , 1661525 , 26132 ,..................., "No Malware"}
{ 5097510 , 5823364 , 59228 ,..................., "Malware" }
⋮ ⋮ ⋮ ⋮

Figure 3: Vector format with all features

rather than emulated ones. As shown in Table 1 our CPU has 44 events available under *perf* tool. As Intel Haswell has only 8 counter registers available[10], we can only measure 8 events at a time. We divide 44 events into 6 batches of 8 events and run each application 6 times at sampling time of 10ms to gather all micro-architectural events. Running malware inside container can contaminate the environment which may affect subsequent data collection. To make sure there is no contamination in collected data due to previous run, we are destroying container after each execution. We automate this data gathering using Python scripting. All collected events are transformed into vectors with events as columns as shown in Figure 3. Moreover, extra column of "class" is added to each vector in which collected data is manually labeled as "Malware" or "No Malware" depending on the application type. Manual labeling is used for both learning and testing of Machine Learning model for different classifiers.

4. OFFLINE LEARNING AND TESTING

Detecting malicious behavior is a binary classification problem which we are addressing using supervised Machine Learning(ML) techniques. Detailed steps of offline training and testing is shown in Figure 2. After collecting microarchitectural events using *perf*, we use WEKA tool[8] for evaluating accuracy of different ML classification algorithms. WEKA also generates ML models which we use later for synthesizing on FPGA using high level synthesis tools. We follow standard 70%-30% data set split for training and testing. To follow complete non-biased splitting, we separate 70% benign-70% malware application for training and 30% benign-30% malware application for testing. It is important to note that if instead we split the total vector data set then some identical vectors may end up in both sides creating biased results. We have a total of 44 events/features that needs to be captured, which can become a significant overhead both for software implementation as it requires multiple runs of the same applications due to limited number of HPC, and for hardware implementation due to area, power and latency cost. Therefore, it is important to reduce the number of necessary performance counters and eliminate the unnecessary events which are not significantly expressing malware behavior.

4.1 Feature Reduction

Feature reduction can be helpful in reducing offline learning time as well as reducing hardware design complexity. We follow two steps approach for feature reduction: First a manual approach and next an algorithmic on. For the manual approach, we analyze the events manually and exclude certain event which are obviously not related to the goal. Following the manual approach, for the algorithmic approach, we apply attribute selection algorithm available under WEKA to reduce features.

Out of 44 events shown in Table 1, there are certain events which are provided by Linux kernel and they are included as software events under *Perf*. We exclude such events. These events are alignment-faults, context-switches, cpu-clock, cpu-migrations, emulation-faults, major-faults, dummy, minor-faults, page-faults, and task-clock. Moreover, events like cpu-cycles, and ref-cycles don't represent uniqueness in terms of program phase so they are excluded too. A total of 12 events are removed using manual feature reduction.

branch-instructions	branch-loads
instructions	iTLB-load-misses
dTLB-load-misses	dTLB-store-misses
LLC-prefetch-misses	L1-dcache-stores

Table 2: Reduced HPC events

For algorithmic selection of features, we use "Correlation Attribute Evaluation" to rank 32 remaining features under WEKA. Correlation evaluation algorithm calculates pearson correlation between each attribute and class.

$$\rho(i) = \frac{cov(X_i, C)}{\sqrt{var(X_i)var(C)}} \quad (1)$$

where ρ is pearson correlation coefficient. X_i is an input dataset of any performance counter event i. C is an output dataset contains different classes, "Malware" or "No Malware" in our case. Value of i represents anyone feature out of 32 features. $cov(X_i, C)$ measures covariance between input dataset and output dataset. $var(X_i)$ and $var(C)$ measure variance of both input and output dataset respectively. Equation (1) can be elaborated further as shown below.

$$\rho(i) = \frac{\sum_{k=1}^{n}(x_{k,i} - \bar{x_i})(c_k - \bar{c})}{\sqrt{\sum_{k=1}^{n}(x_{k,i} - \bar{x_i})^2 \sum_{k=1}^{n}(c_k - \bar{c})^2}} \quad (2)$$

where, k is the total number of values in both dataset. $x_{k,i}$ is k^{th} value in input dataset for feature i. c_k is k^{th} value in output dataset. This algorithm finds correlation co-efficient for all 32 features as per above equation. We list top 8 features with the highest correlation co-efficient value. These short listed features are shown in Table 2. These events have mixture of branch related events representing core behavior and cache related events representing memory behavior of an application.

"branch-instruction" has highest value of ρ than other features. Features that are listed in Table 2 has following pearson correlation coefficient relation.

ρ(branch-instructions) $>$ ρ(branch-loads) $>$ ρ(instructions) $>$ ρ(iTLB-load-misses) $>$ ρ(dTLB-load-misses) $>$ ρ(dTLB-store-misses) $>$ ρ(LLC-prefetch-misses) $>$ ρ(L1-dcache-stores)

4.2 Result analysis

We apply 11 ML classifiers to our dataset to find their accuracy in classifying malware and benign applications. We calculate accuracy before algorithmic feature reduction with 32 features and after feature reduction with top 8, 4 and 1 feature(s) as shown in Table 2. Figure 4 shows accuracy of various ML classifiers for our dataset. Before feature reduction, most of the algorithms perform well, mostly providing above 80% accuracy. Certain algorithms have significant accuracy reduction due to reduction in features but few classifiers perform well even after feature reduction. Classifiers like JRIP, OneR, PART, J48, and MultiLayerPerceptron have small reduction in accuracy after feature reduction. The Reason of OneR's constant accuracy result is due to the nature of OneR algorithm. OneR algorithm chooses only one feature that can most accurately predict, which is "branch-instruction", which explains why OneR is not affected by feature reduction. The results show that a minimum of four features on average provide relatively high accuracy of more than 80% across most algorithms. Reducing the features below four significantly impact performance in most classifier.

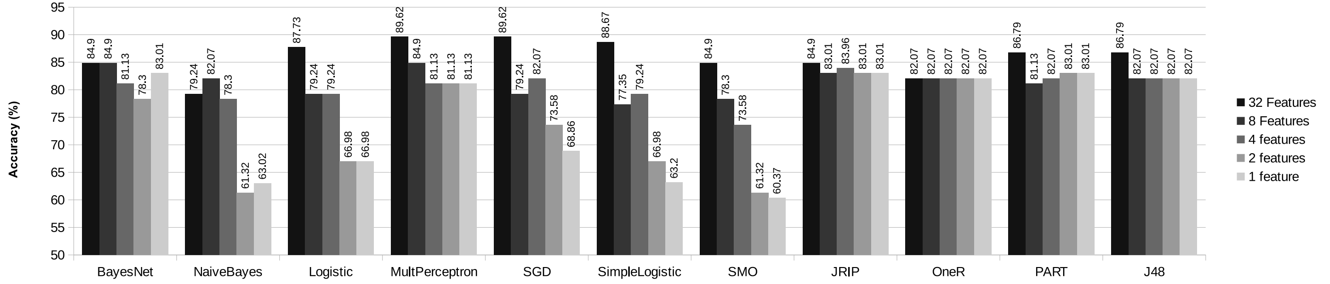

Figure 4: Accuracy comparison of ML classifiers

The results suggest that at least four hardware performance counter registers are required to highly accurately detect malware.

Also it is important to note that performance counters data are collected at 10ms interval rate because of *perf*'s maximum sampling rate limitation. There are patches available to boost *perf* sampling rate to 1ms interval. However, increasing sampling rate doesn't increase the detection capacity noticeably for common Linux malware such as buffer overflow and Return-Oriented Programming(ROP) attacks [6]. While higher sampling rate can bring slightly higher accuracy for other attack types, it generates noticeable measurement overhead.

5. SOFTWARE IMPLEMENTATION

In this section, we present details of software implementation of ML classifiers. For software implementation, we fully implement the classifiers listed in Table 3 at OS Kernel level to understand various software overheads. Software overhead includes the time to read from the HPC and the time to execute the classifiers. Intel CPU comes with Turbo Boost technology which may generate error in execution time measurement. We disabled Turbo Boost and also set CPU governor to operate at constant frequency of 800 MHz to avoid possible error in measurement. Performance counter reading overhead is negligible in kernelspace when monitoring single core but overall system overhead increases for monitoring processes running on multiple cores. Overhead can be as high as 12% [4] depending on the number events that are being sampled. The results for software implementation overhead of these classifiers show them to be slow with the execution time in the range of milliseconds, which is order of magnitude higher than the latency needed to capture malware at run-time. It is important to note that several studied malwares have execution time in the range of microseconds or less which require fast detection to prevent them from corrupting the system.

Classifier	Latency(ms)	Classifier	Latency(ms)
BayesNet	0.624	NaiveBayes	0.802
SMO	0.652	SimpleLogistic	0.648
SGD	0.652	PART	0.642
OneR	0.653	MultiLayerPercep	0.87
Logistic	0.844	JRIP	0.653
J48	0.663		

Table 3: ML classifier execution overhead

6. HARDWARE IMPLEMENTATION

The latency overhead of software implementation of classifiers is calling for hardware accelerated implementation of these algorithms to enhance their performance in analyzing malware related microarchitectural events. When it comes to choosing Machine Learning(ML) classifiers for hardware implementation, accuracy of any algorithm is not the only important parameter. Area, power and latency of logic (performance) implementation are also key factors in deciding a cost-efficient ML classifier. Some complex ML classifier algorithms such as Ensemble Learning [12] or Convolution Neural Networks can boost up accuracy of baseline classifiers but they will also add significant overhead for hardware implementation in terms of logic area, power consumption and detection latency. We are interested in analyzing these overheads when implementing commonly available baseline ML classifiers. Classifiers with high accuracy, low area, low power consumption, and low latency are the ideal choices for hardware based detection techniques. That doesn't mean accuracy boost techniques are discouraged but techniques which are going to make classifier model more complex on hardware at a benefit of only small incremental accuracy improvement are discouraged. For hardware implementation, we use Xilinx High-Level Synthesis (HLS) compiler which translates high level code into HDL code.

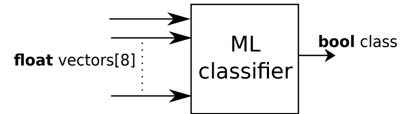

Figure 5: ML classifier block

6.1 High Level Synthesis

We are using Xilinx Vivado Design suite to synthesize ML classifiers for Xilinx Virtex 7 FPGA. Latency and power estimation are collected at 10ns clock cycle time. Accuracy of ML classifiers are based on data collected at 10ms interval using *perf*.

Offline generated classifier models from WEKA are implemented in C code. The generated models from WEKA have information pertaining to corresponding ML classifiers. For example, generated BayesNet model have table of probabilities associated with each feature and J48 classifier model have whole decision tree printed. We manually interpret the models and implement the logic in C code. For synthesizing ML classifier models, we assume that the logic of fetching counter values periodically from CPU is already implemented and is the same for all studied classifiers. Hence, we are excluding data fetching logic for latency, area, and power estimation. Moreover, we assume that vector with all HPC events are already available at the input to classifier. Every model is treated as a black box which accepts HPC event vector of size 8 and outputs binary value "mal-

ware" or "not malware". The logical view of the ML classifier hardware is shown in Figure 5. Core logic of black box is implemented from WEKA generated models. Xilinx Vivado HLS compiler is used to convert C to VHDL IP core. During high level synthesis latency and area/utilization of all ML classifiers are collected. To collect total power estimation of the implemented model, IP core is synthesized in Vivado. Power estimation is collected for 100 MHz clock attached to the IP core. Power estimation contains both static power and dynamic power consumption of digital logic.

Results of estimated latency, area, and power are shown in Table 4. Latency unit is in terms of number of clock cycles required to classify input vector. Area unit is the total number of utilized LUTs, FFs, and DSP units inside Virtex 7 FPGA. The unit for power consumption is Watt.

Classifier	Latency (cycles@10ns)	Power (W)	Area (LUTs+FFs+DSPs)
BayesNet	14	0.445	6794
NaiveBayes	233	1.34	58177
SMO	34	0.443	2556
SimpleLogistic	22	0.454	4721
SGD	34	0.444	2556
PART	6	0.436	2131
OneR	1	0.324	1258
MultiLayerPercep	302	1.03	36252
Logistic	68	0.63	13041
JRIP	4	0.436	1504
J48	9	0.436	1801

Table 4: Hardware Synthesis Result

6.2 Ranking Classifiers

In this section we present the ranking of various ML classifiers implementation in hardware in terms of latency, area, power, and accuracy. An ideal classifier should have high accuracy, small logic footprint, fast detection response time (delay), and low power consumption. Out of these parameters logic area and power consumption are important during hardware design phase due to silicon and power budget. Latency and accuracy are both Quality-of-Service(QoS) parameters for malware detectors. Different computing system have different budget requirements. For example, data centers or servers have more aggressive requirements on QoS than power and area whereas battery powered devices have more aggressive requirement on power and area budget than QoS. As accurate detection of malware is the highest priority for every type of computing device, accuracy of malware detection is the most important parameter to decide suitable ML classification algorithm. Hence, we analyze different combination of parameters along with the accuracy of the algorithms.

To account for power and latency together, we calculate Power Delay Product(PDP) of all ML classifiers using latency and power data collected from hardware synthesis. This comparison is helpful in shortlisting classifiers which are more suitable for battery powered embedded devices. Comparison of accuracy and PDP is shown in Figure 6. ML classifiers with lower PDP and higher accuracy is preferred. Classifiers OneR, JRIP, PART and J48 performs better than the rest of the classifiers. OneR, JRIP ,and PART are rule based classifiers, and generate rules for the features which involves comparisons rather than computation, therefore they can run faster. This is also the case for tree based classifier, J48. Classifiers such as BayesNet and Logisitc regression involves computation like probabilities and sigmoid functions respectively, resulting in higher execution latency.

In addition to latency and power, we also compare accu-

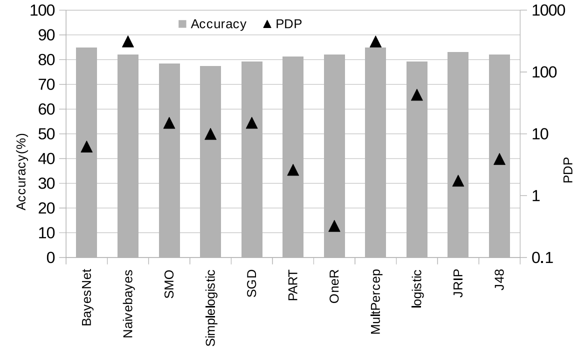

Figure 6: PDP and Accuracy Comparison

racy over unit of hardware area for various ML classifiers. Area and accuracy comparison is putting more emphasis on silicon area budget. We use ratio of Accuracy over Area to list down ML classifiers which requires small area and yet can predict with high accuracy. We show the results of Accuracy/Area in Figure 7. Classifier with higher ratio is considered better than with lower ratio. Again, rule based and tree based classifiers are performing significantly better in terms of accuracy per area compared to highly accurate but complex Bayesnet, MultiLayerPerceptron and logistic classifier.

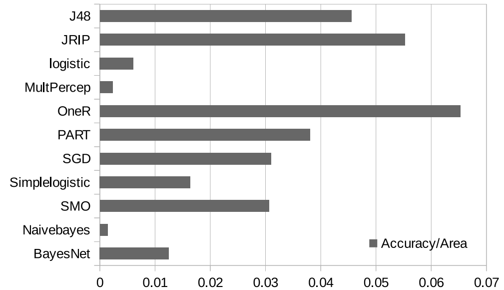

Figure 7: Accuracy/Area Ratio Comparison

Comparing the accuracy, PDP and accuracy/area results show that rule based and tree based classifiers are significantly better compared to complex and more accurate classifiers like MultiLayerPerceptron and Logistic. It is important to note that for this study we have mainly considered primary classifiers only. Certain tree based algorithms such as Random forest will be complex to implement too but will give better Accuracy to Area ratio than complex Artificial Neural Network(ANN) algorithms.

7. RELATED WORK

In this section, we discuss related work in the field of hardware based malware detection. Demme et al[5] first showed that offline Machine Learning(ML) can classify malware by learning performance counter event traces. They showed high detection accuracy result for Android malware by applying complex ML algorithms like Artifical Neural Network(ANN) and K-Nearest Neighbour(KNN). Although, they have discussed implementing classifiers on hardware, they didn't present any hardware overhead analysis results. The hardware implementation overhead, in particular power, area and latency are important as they decide which ML classifier performs most cost-efficient. As our results showed,

with ignoring hardware overheads, complex ML classifier such as MultiLayerPerceptron and logistics are the winners given their high accuracy, however after taking into consideration the overhead, they perform worst in terms of PDP, accuracy/area and latency compared to significantly simpler but slightly less accurate rule based and tree based classifiers such as JRIP, J48 and OneR. Tang et al[19] and Garcia[6] discussed feasibility of unsupervised learning method to detect ROP and buffer overflow attacks by finding anomaly in HPC during execution of such attacks. Although unsupervised algorithms are effective, they are complex in nature which can increase complexity when implementing in hardware. Also the software implementation is not an effective solution to detect malware at run-time, due to large latency to compute the complex algorithms. In a different work Ozsoy et al[17] used sub-semantic features rather than performance counters to detect malware. Moreover, they suggested changes in microprocessor pipeline to detect malware in truly real-time nature. They discussed estimated latency and area utilization of Logistic and ANN algorithm implementation for their architecture. However, Our work is different as it does not require any change in processor pipeline. Bahador et al[2] used Singular Value Decomposition(SVD) technique to detect malicious software in real-time. They used similar ML classifiers as ours but they haven't discussed hardware implementation of those classifiers.

8. CONCLUSION

This paper is the first effort in thoroughly analyzing various machine learning methods that uses hardware performance counters to classify benign and malware applications. We first present full software (Linux Kernel) and hardware (FPGA) implementation of various ML classifiers. While the accuracy of an ML classifier to detect malware is dependent on the number of feature captured with hardware performance counters, we found a minimum of four hardware performance counter to be sufficient to provide almost 80% accuracy across all studied ML classifiers. The results further show that the software solution is not responding as fast as it requires to capture run-time behavior of malware. In response, the hardware accelerated solution shows promising results, reducing the latency by order of magnitude with small hardware cost. The results show that without hardware overheads consideration, complex ML classifiers such as MultiLayerPerceptron and logistics are the winners given their higher accuracy, however after taking into consideration their implementation overheads, they perform worst in terms of PDP, accuracy/area and latency compared to significantly simpler but slightly less accurate rule based and tree based classifiers such as JRIP, J48 and OneR. Among all studied classifiers OneR, mainly relying on single branch-instruction hardware performance counter information, achieves over 80% accuracy, takes only 10ns for computation, consumes only 0.3 watts of power, and occupies just more than 1K gates for physical implementation found to be the most cost-effective solution for hardware based malware detection.

References

[1] Virustotal intelligence service. http://www.virustotal.com/intelligence/. Accessed: November 2016.

[2] Bahador et al. Hpcmalhunter: Behavioral malware detection using hardware performance counters and singular value decomposition. In *(ICCKE), 2014*. IEEE.

[3] Bircher et al. Complete system power estimation: A trickle-down approach based on performance events. In *2007 IEEE International Symposium on Performance Analysis of Systems & Software*, 2007.

[4] Bitzes et al. The overhead of profiling using pmu hardware counters. *CERN openlab report*, 2014.

[5] Demme et al. On the feasibility of online malware detection with performance counters. In *ACM SIGARCH Computer Architecture News*, volume 41, 2013.

[6] A. Garcia-Serrano. Anomaly detection for malware identification using hardware performance counters. *arXiv preprint arXiv:1508.07482*, 2015.

[7] Guthaus et al. Mibench: A free, commercially representative embedded benchmark suite. In *2001 IEEE International Workshop on Workload Characterization*.

[8] Hall et al. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 2009.

[9] M. Helsley. Lxc: Linux container tools. *IBM devloperWorks Technical Library*, 2009.

[10] Intel. Intel 64 and ia-32 architectures software developer's manual, volume 3b: System programming guide. *Part*, 2:18–65, 2016.

[11] Jacob et al. Behavioral detection of malware: from a survey towards an established taxonomy. *Journal in computer Virology*, 4(3):251–266, 2008.

[12] Khasawneh et al. Ensemble learning for low-level hardware-supported malware detection. In *International Workshop on Recent Advances in Intrusion Detection*, pages 3–25. Springer, 2015.

[13] M. Malik and H. Homayoun. Big data on low power cores: Are low power embedded processors a good fit for the big data workloads? In *2015 33rd IEEE International Conference on Computer Design (ICCD)*.

[14] M. Malik, S. Rafatirah, A. Sasan, and H. Homayoun. System and architecture level characterization of big data applications on big and little core server architectures. In *2015 IEEE International Conference on Big Data*.

[15] M. Malik, A. Sasan, R. Joshi, S. Rafatirah, and H. Homayoun. Characterizing hadoop applications on microservers for performance and energy efficiency optimizations. In *Performance Analysis of Systems and Software (ISPASS), 2016 IEEE International Symposium on*, pages 153–154. IEEE.

[16] McAfee Labs. Infographic: Mcafee labs threats report. March 2016.

[17] Ozsoy et al. Malware-aware processors: A framework for efficient online malware detection. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*.

[18] Spinellis et al. Reliable identification of bounded-length viruses is np-complete. *IEEE Transactions on Information Theory*, 49(1):280–284, 2003.

[19] Tang et al. Unsupervised anomaly-based malware detection using hardware features. In *International Workshop on Recent Advances in Intrusion Detection*, pages 109–129. Springer, 2014.

[20] Wang et al. Confirm: Detecting firmware modifications in embedded systems using hardware performance counters. In *Computer-Aided Design (ICCAD), 2015 IEEE/ACM International Conference on*.

[21] T. Winograd, H. Salmani, H. Mahmoodi, K. Gaj, and H. Homayoun. Hybrid stt-cmos designs for reverse-engineering prevention. In *Proceedings of the 53rd Annual Design Automation Conference*. ACM, 2016.

[22] T. Winograd, H. Salmani, H. Mahmoodi, and H. Homayoun. Preventing design reverse engineering with reconfigurable spin transfer torque lut gates. In *Quality Electronic Design (ISQED), 2016 17th International Symposium on*, pages 242–247. IEEE.