Can Monitoring System State + Counting Custom Instruction Sequences aid Malware Detection?

Aditya Rohan
Department of Materials Science and Engineering
IIT Kanpur
raditya@iitk.ac.in

Kanad Basu
Department of Electrical and Computer Engineering
University of Texas at Dallas
kanad.basu@utdallas.edu

Ramesh Karri
Department of Electrical and Computer Engineering
New York University
rkarri@nyu.edu

***Abstract*—Signature and behavior-based anti-virus systems (AVS) are traditionally used to detect Malware. However, these AVS fail to catch metamorphic and polymorphic Malware - which can reconstruct themselves every generation or every instance. We introduce two Machine learning (ML) approaches on system state + instruction sequences –which use hardware debug data – to detect such challenging Malware. Our experiments on hundreds of Intel Malware samples show that the techniques either alone or jointly detect Malware with $\geq$ 99.5% accuracy.**

I. Introduction

Malware, a portmanteau for malicious software, is any software designed to disrupt, damage, or gain unauthorized access to a computer system. Malware can be classified based on their functionality as follows: Adware, Spyware, Virus, Worm, Trojan, Rootkit, and Keyloggers. Signature-based detection is one of the most popular commercial Malware detection techniques [1]. However, since signature-based detectors need the signature of a Malware apriori, they are ineffective against unknown zero-day Malware. Another class of detection techniques is based on Malware behavior. This class of techniques observe the behavior of a program to conclude whether it is malicious or not. The advantage of using a behavior-based system over a signature-based system is the ability to detect unknown and morphing Malware. However, behavior-based techniques incur processing time. Heuristic methods use data mining and ML to learn the behavior of Malware. An early attempt at heuristic Malware detection was employed by [2].

In this paper we introduce two new Malware detection algorithms, namely BSLA (monitoring instruction sequences) and GPR (processor system state), utilizing the hardware debug logic present in all computers. We propose to re-purpose debug logic for extracting features for the ML-based classifier, since it is easily accessible. Using pre-existing debug logic avoids the cost of manufacturing new hardware. Since the debug logic is not used in-field, re-purposing it does not incur hardware overhead. We extract salient features for the two Malware detection techniques and use them for binary classification of programs (Malware vs benign) by a ML-based classifier. The approach achieves accuracy as high as 99.65% with very basic classifiers without much tuning.

The paper is organized in the following manner: we discuss related work in Section II and describe the background on Malware morphing, Debug Hardware, Hardware Performance Counters, and ML terminology used in the paper, in Section III. The approaches are discussed in Section IV. We present the experimental setup in Section V and the results in Section VI. Section VII concludes the paper.

II. Related Work

Since software-based Malware detection is not effective, researchers are using trusted hardware features instead. Hardware performance counters (HPC) have been proposed for Malware detection. This approach uses dynamic analysis of programs using HPCs to detect Malware. HPC-based rootkit detection was developed by [3]. [4] using ML on HPC features to distinguish benign programs from Malware. In [5], hardware features were used by an unsupervised anomaly-based Malware detection scheme. [6] used HPCs to detect anomaly in multi-threaded processes. [7] developed an analytical model for malware detection using HPCs. Several researchers are using trusted HPC-based features to improve system security [8], [9], [10]. It is difficult for an attacker to bypass a hardware-based detector when compared to a software equivalent. The approach improves the existing HPC-based Malware detection by utilizing custom HPCs and data from General Purpose Registers to distinguish Malware from benign programs.

Micro-architectural characteristics are also able to distinguish benign programs and Malware using certain opcode categories as features for an ML-based classifier [11]. Starting from a feature space of 25 attributes, the authors created new feature spaces of branch, unigram and bigram opcodes using feature reduction techniques such as “Discriminant Feature Variance-based Approach” and “Markov Blanket”. The system detected Metamorphic Worm and Next Generation Virus Construction Kit viruses with 100% accuracy, precision, and recall. Statistical analysis of opcode distributions can also detect Malware [12]. The distribution of opcode frequency of Malware differs significantly from benign programs, which helped to distinguish the two.

Morphing Malware, a complex class of Malware, explained in Section III, are difficult to detect using traditional signature-based detectors. Researchers have developed behaviour-based techniques to detect these types of Malware [13], [14]. However, as explained in Section I, these methods incur high cost and timing overhead.

III. Background

A. Malware Morphing

Attackers morph Malware to create variants so as to evade upgraded AVS. There are two classes of morphing Malware:

Polymorphic Malware modifies itself to make it difficult for signature-based detectors. Such Malware has a mutation engine that mutates its structure; this mutation engine typically changes the decryption routine used by the Malware. Behavior-based techniques [13] successfully detect polymorphic Malware as only the code structure of such Malware changes while the behavior-based structural model is not changed.

Metamorphic Malware rewrites itself in each iteration of propagation. The new version of the code is different from the previous version and thus evades detection by signature-based detectors. The longer a metamorphic Malware lives, the more complex its structure becomes and the harder it is to detect. Techniques used to morph Malware include (i) adding a variable number of NOP instructions or useless loops, (ii) switching the registers used, (iii) function re-ordering, and (iv) program flow modification. Metamorphic Malware succumbs against methods that use Opcodes [12] and control flow graphs [14], because these techniques also take program semantics into account, along with the code structure. Metamorphic Malware only change the program structure; however, the functionality remains same and hence, can be detected using ways that preserve program semantics.

B. Hardware-based Malware detector

Hardware Performance Counters (HPCs) are general purpose registers used to measure micro-architectural events like number of branches taken and number of instructions. HPCs were initially used for compiler improvement, but have been used since for Malware detection [15], [3], [4].

Metamorphic Malware are difficult to detect. As shown by [16], metamorphic Malware can undermine signature-based detection techniques. Monitoring HPC counts periodically is also not robust against metamorphic Malware, since a change in HPC values due to modification in program binary may alter the classification results. Therefore, it is essential to go beyond HPCs to other hardware features, to techniques that offer insights into the binary analysis of the Malware.

The first approach BSLA monitors instruction sequences to distinguish between Malware and benign programs. It is challenging to monitor a sequence of instructions using HPCs. Hence, we propose to add HPCs that count instruction sequences instead of simple instructions. The second approach uses register values obtained using debug hardware like ARM Coresight or Intel PAT.

C. Debug Hardware

Post-silicon validation and debug are used to identify undetected functional bugs in a fabricated chip [17]. Our second method, GPR, reuses the debug hardware to detect Malware. We will describe two popular debug architectures.

Important elements of the **ARM Coresight** debug architecture (as shown in Figure 1) include the trace buffer, the trace bus, the trace port, and the JTAG debug port. Each core has an embedded trace macrocell (ETM) that captures the trace data and transmits them to the trace bus.

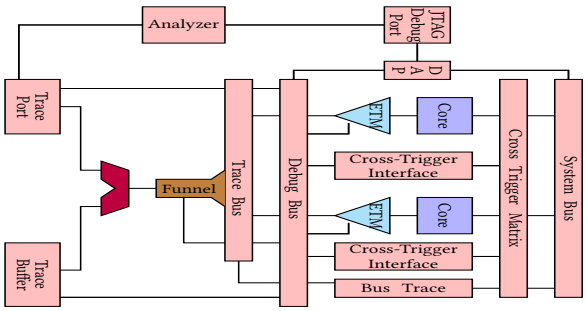

Fig. 1. ARM CoreSight debug architecture.

The trace bus duplicates the trace data using a funnel and directs it to either the trace buffer for on-chip analysis or via the trace port for off-chip analysis. The trace port can deliver the collected traces in real-time to an off-chip analyzer. The debug engineer uses hints from the analysis to add debug trigger points and inputs.

Intel Platform Analysis Technology (PAT) contains a set of tools for debugging software on Intel architecture, similar to ARM CoreSight. Intel PAT also provides functionality like setting breakpoints, run and stop execution control, as well as register and memory access and information. Intel Processor Trace and Intel Trace Hub are hardware and platform-level features respectively that log information about program execution from hardware, firmware, and software without significant overhead and interruption.

D. ML Preliminaries

We discuss ML terminology used in the study.

- *True Positives (TP)* are Malware classified as Malware.
- *True Negatives (TN)* are benign programs classified as benign programs.
- *False Positives (FP)* are the benign programs classified as Malware.
- *False Negatives (FN)* are the Malware classified as benign.

Accuracy is the most direct way to assess the performance of a model, it's simply the ratio of total correct observations over the number of the total observations.

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN}$$

Precision is the ratio of correctly predicted Malware to the total predicted Malware. Precision tells us the fraction of samples tagged Malware are actually Malware.

$$Precision = \frac{TP}{TP+FP}$$

Recall is the ratio of number of samples classified as Malware and number that are actually Malware. It tells us the fraction of Malware that were correctly identified as such.

$$Recall = \frac{TP}{TP+FN}$$

IV. Proposed Malware Detection Methodology

Figure 2 outlines the proposed approaches. Both approaches operate in two steps. In the first step, a binary analysis of the program (benign or Malware) is performed to infer program properties. These properties are used to train a ML-based classifier to distinguish between benign programs and Malware. In the second step, the corresponding properties are extracted from programs at run time and fed as inputs to the classifier which classifies them as benign or malicious. Similar classifier-based methods have been used for Malware detection before in [2],[18]. However, our methods are different from [2],[18] as we use instruction sequencing and system state for detection, with a significantly lower number of features providing similar or better results.

A. Monitoring Custom HPCs of Branch-Store-Load-Arithmetic Instruction Sequences (BSLA)

BSLA uses custom HPCs to measure the ordering of certain types of instructions within the binary. The instruction types we chose cover a large number of instructions: branch/jump (b), load (l), store (s), and arithmetic(a). The feature vector we use for training has 20-elements as follows: $bb, bl, bs, ba, lb, ll, ls, la, sb, sl, ss, sa, ab, al, as, aa, b, l, s, a$. Here 'bb' counts the number of "branch instruction $\rightarrow$ branch instruction" and 'bl' counts the number of " branch instruction $\rightarrow$ load instruction". HPCs can be designed to compute the values of these custom instruction sequences. Two data sets are created using these HPC values – one for Malware and the other for benign programs. The Malware data are labeled with **1**, while the benign data are labeled with **0**. A combination of HPC values make up the training set for the ML-based classifier. Our experimental data set contains **472** Malware and an equal number of benign programs. For classification, we split the data randomly into 70% training data and 30% testing data.

Furthermore, we assess the effect of each type of instruction on the classification accuracy. We remove each of branch, load, store and arithmetic instructions in turn from the data set to create four new data sets, comprising of reduced number of HPC values. For example, removing only arithmetic instructions from the data set results in the training vector with the following HPCs: $bb, bl, bs, lb, ll, ls, sb, sl, ss, b, l, s$. Similarly, we remove two instruction types to monitor the effect of the combination of these on the original data set. For example, excluding store and arithmetic instructions from the data sets yields the vector with only six HPC values: bb, bl, lb, ll, b, l.

1) Hardware Implementation: The primary hurdle in implementing BSLA is obtaining the feature vectors, i.e., count of the sequence of instructions, which are not supported by existing HPCs. One can develop custom HPCs for each of these features, like bl and ba.

In order to generate and verify the feasibility of custom HPCs, we used Xilinx Vivado High Level Synthesis (HLS) on a MIPS processor [19]. The HLS started with a high level software description of the processor in C and converted it into a RTL. We modeled specialized HPCs like bl, sb and sa in the C description of the MIPS processor and synthesized it using Vivado HLS. The hardware overhead for each counter is 1.5% and has no performance penalty.

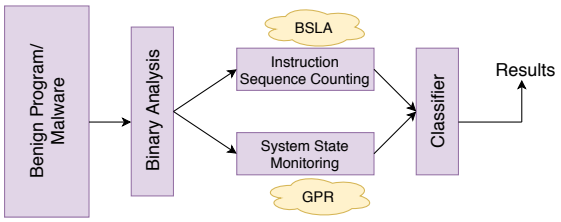

Fig. 2. Outline of the two Malware detection approaches.

2) Drawbacks: An attacker may create Malware that can evade detection. Obfuscation techniques change the structure of the program without changing its functionality. Popular obfuscation methods include Garbage-code-insertion, Register Usage exchange, Subroutine Permutation, Instruction Reordering and Instruction Replacement [20]. Garbage-code-insertion morphs the program by inserting instructions like NOP and XCHG, that doesn't affect the functionality but alter the structure of the program. Another method is Instruction Replacement. Certain Malware replaces some instructions with other instructions that have identical functionality; e.g., replacing "sub eax, eax" with "xor eax, eax". BSLA can not detect a morphed Malware, since the order of instructions as well as the type change. Thus we introduce another method that fixes this drawback of BSLA.

B. GPR: Monitoring the General-Purpose-Registers' States

In order to overcome the shortcomings of BSLA, we use a second method known as GPR, that traces the values stored in the general purpose registers at different points during program execution. The GPR in a processor store footprints of various stages of a program execution. Even if a Malware modifies itself using metamorphic or polymorphic methods, its footprints in the registers will still be the same [21]. Tracking the evolution of GPRs during the program execution can distinguish Malware from benign programs.

We use an ML classifier for GPRs. The values stored in the registers with a code designating the binary classification, is used as a vector for training. This can yield multiple data sets. We use two of them that cover the rest in terms of additional value they provide. In Data set 1, the values of GPRs were collected once after execution of every binary and in Data set 2, the GPR values were collected every 1000 instructions[1].

To store the GPR values, we simulated the assembly instructions with an interpreter in python, with simulated memory and GPRs. Data set 1 contains 134 vectors each for benign and Malware programs. data set 2 contains 3036 vectors each for benign and Malware programs.

1) Hardware Implementation: GPR monitoring can be implemented using the debug infrastructure in Section III-C. ARM Coresight and Intel PAT provide a detailed view of the GPRs. These GPR values can be read via a JTAG interface. Hence, the GPR monitoring has no hardware overhead.

[1]We call it the per file approach, i.e. once for every file.

V. Experimental Setup

A. Intel 80386 Instruction Set Architecture

The general purpose Intel registers that we traced are: *eax, ebx, ecx, edx, edi, esi, ebp, esp.* We use **134** 32-bit Malware samples and **134** 32-bit benign programs for training. We choose to perform the experiments on 32-bit Malware as they are more abundantly available, as shown by the VirusTotal Malware database.

B. Data sets

We create the data sets using program and malware binaries. We use Malware binaries from VirusTotal for Malware data set and benign program binaries[2] from **/usr/bin** directory of the Linux systems for benign program data set. These raw binaries are not ready for the classifier. We processed them using the Linux shell utility objdump which can be used to study various properties of an object file. We used objdump to disassemble binaries and store the output of all programs for further processing.

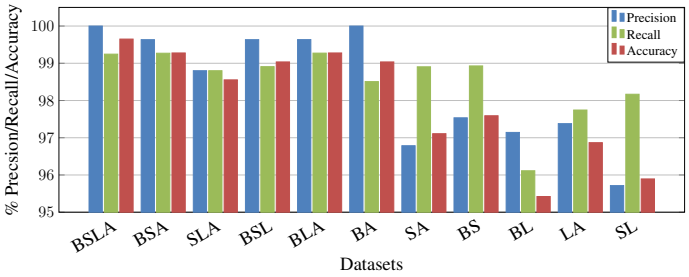

Fig. 3. Results for all datasets for BSLA approach with RF classifier.

VI. Results

A. BSLA

Figure 3 reports precision, recall and accuracy of BSLA on the data sets. The first plot refers to the data set when all instructions are considered. The next four plots indicate data set when one out of four instruction types are not included. For example, BSA indicates the scenario when only load instructions are excluded. The last six plots use data sets when two instruction types are excluded. For example, BA is a data set with no load or store instructions.

Figure 3 shows that the classifier with all instruction types has a precision of 100%, and an accuracy of 99.65%. The accuracy decreases on reducing an instruction type. The effect is most pronounced when branch instructions are removed. Malware and benign programs have noticeable differences in control flow [4]. Hence, not considering branch instructions fails to capture the differences. There is a trade-off between Malware detection accuracy and hardware overhead. The accuracy decreases with an increase in the number of instruction types being excluded. Although the accuracy improves when all instruction types (BSLA as shown in Figure 3) are considered, several custom HPCs have to be measured, which results in a hardware overhead as explained in Section IV-A1.

A large recall for all our data sets signifies that very few Malware were incorrectly identified as benign programs. A high precision indicates that very few benign programs were marked as Malware. Compared to similar approaches in [2], [18], our approach has better accuracy and recall, while using only 10% of the features used in [18].

B. GPR

1) Experiments on simple Malware: We report the results on Malware detection using three ML-based classifiers – K-nearest-neighbors (kNN), random forest (RF), and decision trees (DT), on both data sets (traced every execution and every 1000 cycles). We chose these classifiers since they work very well without any fine tuning and have been demonstrated to provide high Malware detection accuracy in prior research [4]. The results of the RF, kNN and DT classifiers are shown in Figure 4, Figure 5 and Figure 6, respectively. Part (a) of all figures is for Data Set 1 and Part (b) is for Data Set 2. The plot corresponding to "data set 1" or "data set 2" show the precision, recall, and accuracy of a model trained on all 8 GPRs. The rest of the columns represent results created by removing the registers mentioned in the plot axis. For example, "woEAX" refers to the case when data from register *eax* is not considered. In this experiment, we have eliminated only one register at a time.

For Data set 1, we see that the highest accuracy, 98.77% along with a recall of 100%, was achieved when *eax* register is excluded, in Figure 4(a). Therefore, using all register traces for classification is not always the best practice. Another trend is the high precision compared to the relatively lower recall. Results from kNN classifier for data set 1, in Figure 5, when compared to that of the results from RF classifier show an opposite trend: higher recall and lower precision. The maximum accuracy obtained with the kNN classifier is 90.12%, when the *esp* register is excluded. The results from the DT classifier, shown in Figure 6(a), are relatively well balanced when compared to kNN or RF classifiers since it has very similar values for precision, recall, and accuracy. In this case, highest accuracy (95.06%), recall (95.12%) and precision (95.12%) is obtained by excluding *edi* register. For Data set 2, a more pronounced trend of higher precision is obtained using RF classifier (see Figure 4(b)). The overall accuracy of the RF classifier for Data set 2 is lower than that of Data set 1 (94.32%) for all registers. The precision obtained using RF classifier is 97.73%. For Figure 5(b), when kNN classifiers are used, we see the dominance of recall over precision. Unlike the the RF classifier, kNN classifier shows an increase in overall accuracy in Data set 2 compared to Data set 1. The same is true for DT classifier as well. However for both , the highest accuracy achieved is lower than that for data set 1. While the removal of registers doesn't have major effect on accuracy for all classifiers, a closer look suggests that there is a drop in accuracy when *ebx* register is removed in all training sets.

[2]64-bit and 32-bit binaries.

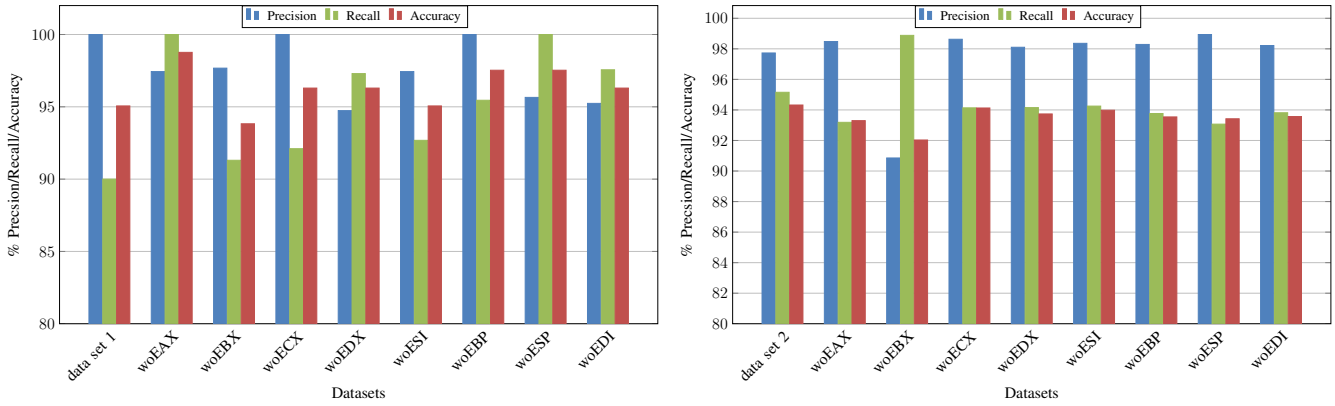

Fig. 4. (a) Results from RF for data set 1 for GPR-Intel approach. (b) Results from RF for data set 2 for GPR-Intel approach.

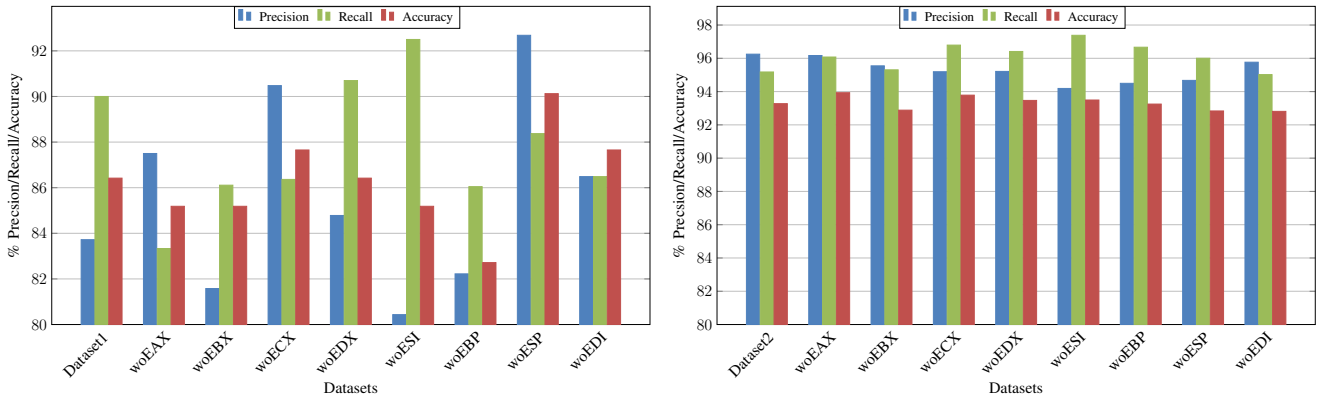

Fig. 5. (a) Results from kNN for Dataset1 for GPR-Intel approach. (b) Results from kNN for Dataset2 for GPR-Intel approach.

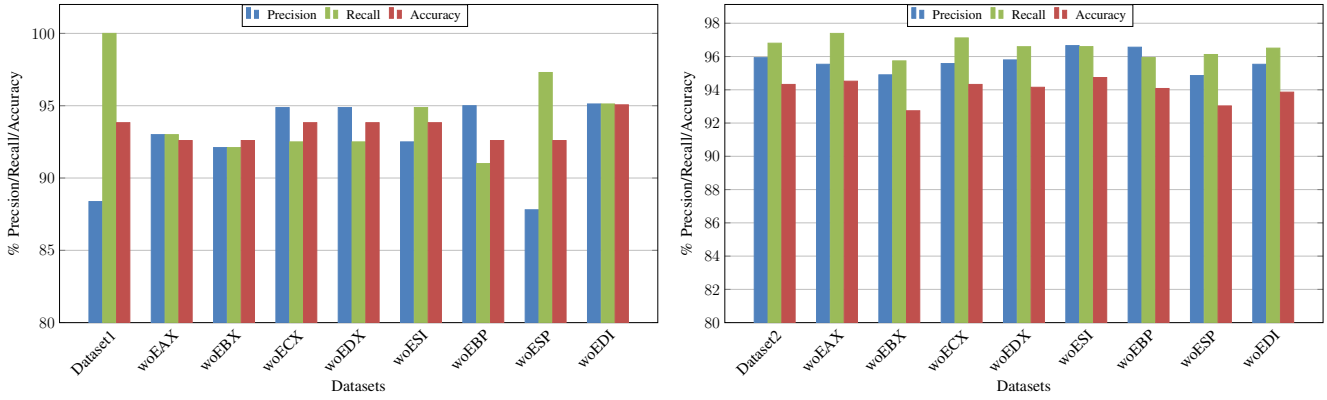

Fig. 6. (a) Results from DT for Dataset1 for GPR-Intel approach. (b) Results from DT for Dataset2 for GPR-Intel approach.

In contrast to the methods in [2], [18], GPR method gives a similar or better performance with a small number of features.

2) Experiments on Metamorphic Malware: To test the method against obfuscation, we create obfuscated samples by inserting some of *NOP* instructions in the assembly dump of the samples. This method was also used in [20]. Table I shows a Malware sample before and after obfuscation. For the Data set 1 (Perfile approach), the precision, recall and accuracy values using GPR remain the same, which means that the obfuscation failed to evade our detector.

The *NOP* instruction does not alter the register values. However, the Data set 2 (data gathered every 1000 cycles) fails against this obfuscation as it collects the register states once every 1000 instructions and introducing a variable number of *NOP* alters the set of instructions that belong to a batch of 1000 instructions after which we store the register state. Our experiments showed that one can change the register values after every 1000 instructions. Meticulous crafting of binaries can allow an attacker to mask GPR values of Malware to mimic a benign program. BSLA does not detect such

Original Malware	After obfuscation
add $0x1,%ecx	add $0x1,%ecx
mov %ecx,-0xa0(%ebp)	nop
	nop
	mov %ecx,-0xa0(%ebp)

TABLE I
OBFUSCATED MALWARE CODE THAT DEFEATS BSLA.

metamorphic Malware, since the *NOPs* break the sequence of instructions. Hence, only GPR with Perfile data sets can detect metamorphic Malware.

C. BSLA+GPR

To evaluate the combined performance of the methods, we select a subset of eight features out of the 20 features from *BSLA* and the eight features from *GPR*. We use two feature selection techniques: *Univariate Selection* selects features with the strongest correlation with the output variables. *Feature Importance* ranks features in terms of importance with respect to the remaining features. The features for out data set are: $la, rbp, rdi, al, ll, bl, as, sb$. The feature vector has two registers (*rbp,rdi*) and six custom HPCs (*la,al,ll,bl,as,sb*). Our experiments are performed on the Intel platform. The results on the three classifiers are shown in Figure 7. For all classifiers, a slight increase in accuracy is obtained compared to *GPR*. BSLA has a better accuracy than BSLA+GPR, but due to the register features, BSLA+GPR doesn't fail against morphing malware unlike BSLA. Compared to GPR, the accuracy improves by about 1-2% for all classifiers, due to the addition of instruction sequencing features.

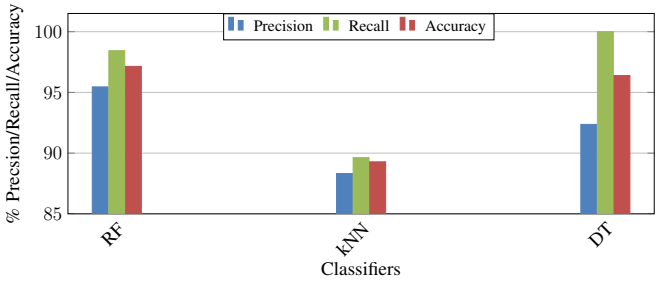

Fig. 7. Performance of classifiers by joining BSLA+GPR features.

VII. CONCLUSIONS

We propose two hardware-based fast, high accuracy techniques for Malware detection, with model building times as low as 0.006 seconds and accuracy upto 97.14%. These detectors are superior to traditional software-based AVS. We use instruction sequencing for our first approach (BSLA) and monitor the state of the GPRs for the second one (GPR), with both methods providing high precision and recall. Monitoring the state of the GPRs is effective even when the malware obfuscates. Combining the two techniques (BSLA+GPR) gives us a superior malware detection capability.

VIII. ACKNOWLEDGEMENTS

This work supported by National Science Foundation (A#: 1526405).

REFERENCES

[1] Z. Bazrafshan, H. Hashemi, S. M. H. Fard, and A. Hamzeh, "A survey on heuristic malware detection techniques," in *IEEE Conference on Information and Knowledge Technology*, pp. 113–120, 2013.
[2] M. G. Schultz, E. Eskin, F. Zadok, and S. J. Stolfo, "Data mining methods for detection of new malicious executables," in *Proceedings of IEEE Symposium on Security and Privacy*, pp. 38–49, 2001.
[3] X. Wang and R. Karri, "Numchecker: Detecting kernel control-flow modifying rootkits by using hardware performance counters," in *Proceedings of ACM Design Automation Conference*, pp. 1–7, 2013.
[4] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo, "On the feasibility of online malware detection with performance counters," in *ACM SIGARCH Comp. Arch. News*, pp. 559–570, 2013.
[5] A. Tang, S. Sethumadhavan, and S. J. Stolfo, "Unsupervised anomaly-based malware detection using hardware features," in *Proceedings of International Workshop on Recent Advances in Intrusion Detection*, pp. 109–129, 2014.
[6] P. Krishnamurthy, R. Karri, and F. Khorrami, "Anomaly detection in real-time multi-threaded processes using Hardware Performance Counters," *IEEE Transactions on Information Forensics and Security*, 2019.
[7] K. Basu, P. Krishnamurthy, R. Karri, and F. Khorrami, "A theoretical study of hardware performance counters-based malware detection," *IEEE Transactions on Information Forensics and Security*, 2019. To appear.
[8] X. Wang, S. Chai, M. Isnardi, S. Lim, and R. Karri, "Hardware performance counter-based malware identification and detection with adaptive compressive sensing," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 13, no. 1, p. 3, 2016.
[9] V. Jyothi, X. Wang, S. K. Addepalli, and R. Karri, "Brain: Behavior based adaptive intrusion detection in networks: Using hardware performance counters to detect ddos attacks," in *Proceedings of the IEEE International Conference on VLSI Design*, pp. 587–588, 2016.
[10] M. Alam, S. Bhattacharya, D. Mukhopadhyay, and S. Bhattacharya, "Performance counters to rescue: A machine learning based safeguard against micro-architectural side-channel-attacks.," *IACR Cryptology ePrint Archive*, vol. 2017, p. 564, 2017.
[11] J. Raphel and P. Vinod, "Heterogeneous opcode space for metamorphic malware detection," *Arabian Journal for Sci. and Eng.*, vol. 42, no. 2, pp. 537–558, 2017.
[12] D. Bilar, "Opcodes as predictor for malware," *Intl journal of electronic security and digital forensics*, vol. 1, no. 2, pp. 156–168, 2007.
[13] G. R. Thompson and L. A. Flynn, "Polymorphic malware detection and identification via context-free grammar homomorphism," *Bell Labs Technical Journal*, vol. 12, no. 3, pp. 139–147, 2007.
[14] D. Bruschi, L. Martignoni, and M. Monga, "Detecting self-mutating malware using control-flow graph matching," in *Proceedings of Springer International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 129–143, 2006.
[15] C. Malone, M. Zahran, and R. Karri, "Are hardware performance counters a cost effective way for integrity checking of programs," in *Proceedings of the ACM workshop on Scalable trusted computing*, pp. 71–76, ACM, 2011.
[16] M. Christodorescu and S. Jha, "Testing malware detectors," *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 4, pp. 34–44, 2004.
[17] K. Basu and P. Mishra, "Rats: Restoration-aware trace signal selection for post-silicon validation," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, no. 4, pp. 605–613, 2012.
[18] P. Khodamoradi, M. Fazlali, F. Mardukhi, and M. Nosrati, "Heuristic metamorphic malware detection based on statistics of assembly instructions using classification algorithms," in *Proceedings of IEEE International Symposium on Computer Architecture and Digital Systems*, pp. 1–6, 2015.
[19] P. Bruel, A. Goldman, S. R. Chalamalasetti, and D. Milojicic, "Autotuning high-level synthesis for fpgas using opentuner and legup," in *Proceedings of IEEE International Conference on ReConFigurable Computing and FPGAs*, pp. 1–6, 2017.
[20] K. Kaushal, P. Swadas, and N. Prajapati, "Metamorphic malware detection using statistical analysis," *International Journal of Soft Computing and Engineering*, vol. 2, no. 3, pp. 49–53, 2012.
[21] S.-B. Park and S. Mitra, "Ifra: Instruction footprint recording and analysis for post-silicon bug localization in processors," in *Proceedings of ACM Design Automation Conference*, pp. 373–378, 2008.