Adversarial Attack on Microarchitectural Events based Malware Detectors

Sai Manoj Pudukotai Dinakarrao, Sairaj Amberkar, Sahil Bhat, Abhijitt Dhavlle, Hossein Sayadi, Avesta Sasan, Houman Homayoun and Setareh Rafatirad
George Mason University, Fairfax, VA, USA
{spudukot,samberka,sbhat6,adhavlle,hsayadi,asasan,hhomayou,srafatir}@gmu.edu

ABSTRACT

To overcome the performance overheads incurred by the traditional software-based malware detection techniques, Hardware-assisted Malware Detection (HMD) using machine learning (ML) classifiers has emerged as a panacea to detect malicious applications and secure the systems. To classify benign and malicious applications, HMD primarily relies on the generated low-level microarchitectural events captured through Hardware Performance Counters (HPCs). This work creates an adversarial attack on the HMD systems to tamper the security by introducing the perturbations in the HPC traces with the aid of an adversarial sample generator application. To craft the attack, we first deploy an adversarial sample predictor to predict the adversarial HPC pattern for a given application to be misclassified by the deployed ML classifier in the HMD. Further, as the attacker has no direct access to manipulate the HPCs generated during runtime, based on the output of the adversarial sample predictor, we devise an adversarial sample generator wrapped around a normal application to produce HPC patterns similar to the adversarial predictor HPC trace. As the crafted adversarial sample generator application does not have any malicious operations, it is not detectable with traditional signature-based malware detection solutions. With the proposed attack, malware detection accuracy has been reduced to 18.04% from 82.76%.

KEYWORDS

Malware detection, adversarial learning, adversarial malware, hardware security, hardware-assisted security, machine learning

ACM Reference Format:

Sai Manoj Pudukotai Dinakarrao, Sairaj Amberkar, Sahil Bhat, Abhijitt Dhavlle, Hossein Sayadi, Avesta Sasan, Houman Homayoun and Setareh Rafatirad. 2019. Adversarial Attack on Microarchitectural Events based Malware Detectors . In *The 56th Annual Design Automation Conference 2019 (DAC '19), June 2–6, 2019, Las Vegas, NV, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3316781.3317762

1 INTRODUCTION

The ever-increasing complexity of modern computing systems result in the growth of security vulnerabilities, making such systems an appealing target for sophisticated attacks. The attackers take the advantage of existing vulnerabilities to compromise the systems and deploy malware. Malware, also known as malicious software, is a program or application designed by the attackers to infect the computing systems without the user agreement for serving harmful purposes such as stealing sensitive information, unauthorized data access, destroying files, running intrusive programs on devices to perform Denial-of-Service (DoS) attack, and disrupting essential services to carry out financial fraud.

To overcome the shortcomings such as latency and computational complexity of traditional malware detection techniques including signature and semantics-based software-driven techniques [11, 19], hardware-assisted malware detection (HMD) approaches are proposed [5]. HMD refers to utilizing the low-level microarchitectural hardware events and logs for detecting and classifying the malware from benign applications. The HMD enjoys the benefit of reduced malware detection latency by orders of magnitude with smaller hardware cost [5]. Recent works [4–6, 20–22, 24, 25] have shown that by deploying Machine Learning (ML) techniques [12, 16] fed with the low-level microarchitectural events (features) captured by Hardware Performance Counters (HPCs) can aid in differentiating benign and malware applications. The HPCs are a set of special-purpose registers built into modern microprocessors to capture the trace of hardware-related events such as LLC load misses, branch instructions, branch misses, and executed instructions while executing an application (benign or malware).

The work in [5] was one of the preliminary works that has proposed to utilize the HPC data for malware detection and demonstrated the effectiveness of offline ML algorithms in malware classification. They showed high detection accuracy results for Android malware by applying multiple ML algorithms, namely Artificial Neural Network (ANN) and K-Nearest Neighbor (KNN). The researchers in [7] and [24] discussed the feasibility of employing unsupervised learning method on low-level features to detect Return-oriented programming (ROP) and buffer overflow attacks by finding an anomaly in the hardware performance counters' information. Although unsupervised algorithms are more effective in detecting new malware and attacker evolution, they are complex in nature demanding more sophisticated analysis and computational overheads. The work in [15] uses logistic regression to classify malware into multiple classes and train a specialized classifier for detecting malware class. They further used specialized ensemble learning to improve the accuracy of logistic regression. To enhance the performance, the work in [15, 21] proposes use of ensemble ML based solutions for effective malware detection using low-level microarchitectural features. These ML based malware detectors (HMD) can be implemented in microprocessor hardware with significantly low overhead as compared to the software-based methods, as detection inside the hardware is very fast (few clock cycles) [19, 25]. As a

DAC '19, June 2–6, 2019, Las Vegas, NV, USA

ACM ISBN 978-1-4503-6725-7/19/06…$15.00
https://doi.org/10.1145/3316781.3317762

whole, it can be seen that recently a large body of works have been dedicated to employ low-level microarchitectural events fed to ML classifiers to make the systems secure.

On the other hand, despite the ML classifiers being deployed in numerous applications and shown robustness against random noises, the exposed vulnerabilities have shown that the outcome of ML classifiers can be modified or controlled by adding specially crafted perturbations to the input data [8, 17, 18, 23], often referred as *Adversarial samples*. A plethora of works on adversarial attacks exist, focusing specifically on computer vision applications [8, 17, 18, 23], where the number of features are often large. Recently, a few works on crafting adversarial malware are as well proposed in [10]. However, the works such as [10] consider the application features in a binary format (feature exist or not) for showcasing the attack and defense. Though the application features (in binary format) are manipulated, traditional techniques such as semantic and signature analysis based methods can detect these adversaries [14]. Similarly, in [26], authors evaluate the efficiency of detecting malware through HPCs. Though the presented experimental results in [26] are in-favor of efficient malware detection through HPCs, they claim that if HPC traces of malware and benign applications are similar, it is hard to detect malware. However, no details on crafting nor feasibility to create such malware is provided, which limits the efficacy. In contrast to the existing works, *this work proposes an adversarial attack on HMDs in which the adversarial samples are generated through a benign code that is wrapped around a benign or malware application to produce a desired output class from the embedded ML-based malware detector*. One of the main challenges to address is that the attacker or user has no direct access to modify the HPC and furthermore, manipulation of HPCs is highly complex to perform despite employing techniques like code obfuscation for executing malware [5, 13].

Firstly, we assume the victim's defense system to be a blackbox and perform reverse engineering to mimic the behavior of the embedded HMD or other security system and build a ML classifier. In order to determine the required number of HPCs to be generated through the application to be misclassified, we employ an '*adversarial sample predictor*' which predicts the number of HPCs to be generated to misclassify an application by the HMD. As aforementioned, the HPCs cannot be modified directly by the attacker, as such we craft an '*adversarial HPC generator*' application (code) that generates the required number of HPCs. The crafting of adversarial HPC generator is performed by employing a linear model that relates the HPC events and the parameters of the adversarial generator code. This adversarial HPC generator application is wrapped around the application that needs to be misclassified. To the best of our knowledge, *this is the first work that is capable of generating adversarial HPCs through a benign application and proposes a methodology how to craft such an application and obtain adversarial behavior*. The main focus of this work is create false alarms (malware classified as benign and benign classified as malware) in order to weaken the trust on the embedded defenses, which increases the scope for attacks. The proposed work benefits from the following: a) no need to tamper or modify the source code of the application around which the proposed adversarial sample generator code will be wrapped (i.e., executed in parallel); b) the crafted application has no malicious features embedded, thus not detectable by ML malware detectors;

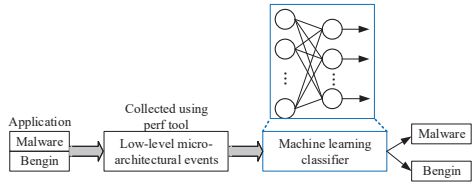


Figure 1: Process of detecting malware by employing low-level microarchitectural events

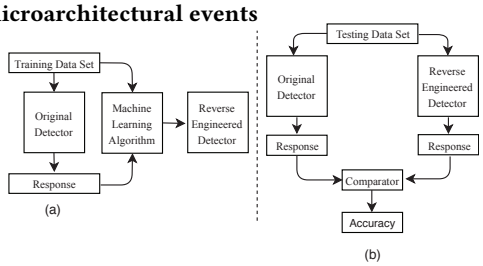


Figure 2: (a) Process of reverse engineering HMD; (b) Testing Performance of Reverse-Engineered Detector

and c) scalable and flexible i.e., the crafted application can generate events as required to generate powerful adversary. We outsource the code on github (https://github.com/saimanojpd/AMC).

The rest of this paper is organized as follows. Section 2 provides an introduction to HMDs. Section 3 describes the approach adopted in this work to perform reverse engineering of HMDs. Section 4 presents the adversarial HPC sample prediction, followed by the details of how application needs to be crafted in order to generate the adversarial HPC samples as predicted in Section 5. The experimental evaluation and setup is presented in Section 6 with conclusions drawn in Section 7.

2 HARDWARE-ASSISTED MALWARE DETECTORS: BACKGROUND

Here, we present the background of HMDs and its functionality. In HMD, when an application is executed, the low-level microarchitectural events are captured with the aid of HPCs. These low-level microarchitectural events are utilized to train ML classifiers to classify the malware from benign applications. For detecting malware during runtime, the HPCs are collected and provided to the ML classifier to determine whether the executing application is malware or benign [5]. Similarly, [19] proposed a single-stage ML-based HMD and analyzed impact of different ML classifiers on area and power overheads. The work in [3] employed HPC values to construct support vector machine (SVM) detectors to identify malicious programs. Similar works are reported in [15, 21]. Figure 1 illustrates the process of using low-level microarchitectural events for malware detection and classification from benign applications.

3 REVERSE ENGINEERING OF HMD

Considering the worst case scenario, where the victim malware detector (defense) is unknown, we perform a reverse engineering to mimic the functionality of the victim HMD. Thus, as a first step to craft adversarial malware, we perform reverse engineering of the victim's HMD similar to that proposed in [14]. The performed reverse engineering is described in Figure 2.

In order to reverse engineer the victim's HMD, we first create a training dataset that comprises of benign and malware applications. Nearly 12,000 benign and 12,000 malware applications are used in the reverse engineering process. The victim's HMD (Original HMD)

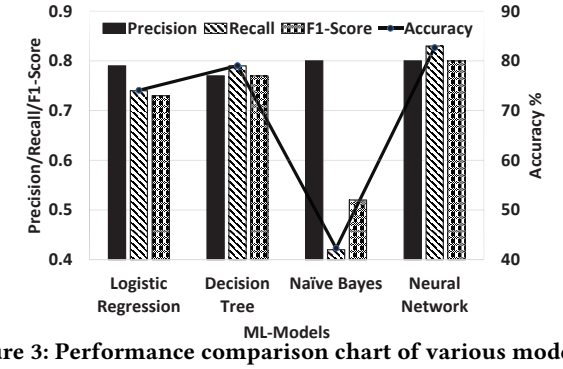


Figure 3: Performance comparison chart of various models

is fed with all the applications and the responses are recorded. These responses are utilized to train different ML classifiers in order to mimic the functionality of the victim's HMD, as shown in Figure 2(a). Further, it is tested by comparing the outputs from victim's HMD response and the reverse engineered ML classifier's response, as shown in Figure 2(b). Reverse engineering is non-trivial as the adversaries generated on a closely functional model will be highly effective compared to a weakly generated adversary. To ensure the reverse engineering is performed in an efficient way, we train multiple ML classifiers and choose the classifier that yields high performance i.e., mimics the victim's HMD with high accuracy.

Performance of Reverse Engineered HMD

We choose Logistic Regression, Decision Tree, Naive Bayes and Neural Network classifiers to mimic the victim's HMD. The rationale for choosing these classifiers is that they represent a wide-range of classification techniques and are popularly employed for robust classification. Figure 3 shows the performance parameters of the experimented four different classifiers used to functionally mimic the victim's malware detector (referred to as Original Detector in Figure 2). One can observe that neural networks perform better compared to other three above mentioned classifiers with an accuracy of 82.7%, precision of 0.8, recall 0.83 and F1-score of 0.80. Higher accuracy and other performance metrics indicate that the employed ML classifier (neural network in this case) mimics the victim's HMD with high precision and is robust. Hence, we consider the neural network as the representation of the victim's HMD in the rest of this work.

4 ADVERSARIAL HPC SAMPLE PREDICTION

Once the reverse engineered HMD is built i.e., neural network's hyper parameters are determined, to launch and craft an adversarial malware, it is non-trivial to determine the level of perturbations that need to be injected into HPC patterns in order to get the applications misclassified. To determine the number of such HPC events to be generated, we deploy (offline) an adversarial sample predictor. As the ML classifiers are robust to random noises, one needs to perturb the HPC patterns in more sophisticated manner. To perturb the HPC patterns, we employ a low-complex gradient loss based approach, similar to Fast-Gradient Sign Method (FGSM) which is widely employed in image processing. The advantage of such an approach is its low complexity and low computational overheads. Additionally, it has been observed from our experiments that the HPC samples follow a continuous distribution, and as such a gradient loss based approach is feasible and beneficial to determine the required perturbation in HPC features to be misclassified.

In order to craft the adversarial perturbations, we consider the reverse engineered ML classifier i.e., neural network with θ as the hyper parameters, x being the input to the model (HPC trace), and y is the output for a given input x, and $L(\theta, x, y)$ be the cost function used to train the neural network. Then the perturbation required to misclassify the HPC trace is determined based on the cost function gradient of the neural network (in this case). The adversarial perturbation generated based on the gradient loss, similar to the FGSM [8] is given by

$$x^{adv} = x + \epsilon sign(\nabla_x L(\theta, x, y)) \quad (1)$$

where ϵ is a scaling constant ranging between 0.0 to 1.0 is set to be very small such that the variation in x (δx) is undetectable. In case of FGSM the input x is perturbed along each dimension in the direction of gradient by a perturbation magnitude of ϵ.Considering a small ϵ leads to well-disguised adversarial samples that successfully fool the machine learning model. In contrast to the images where the number of features are large, the number of features i.e., HPCs are limited, thus the perturbations need to be crafted carefully and also be made sure it can be generated during runtime by the applications. For instance, a HPC of value '-1' cannot be generated by an application. Hence, we provided lower bound on the adversary values that can be predicted.

In contrast to works that assume the application features to be binary such as [10], this work aims to predict and determine the adversaries for the low-level microarchitectural event patterns i.e., HPC patterns to generate during runtime with the aid of a benign code, which is one of the primary distinctions from existing works. It needs to be noted that determining the required perturbation for a given application is done offline. The process of crafting the adversarial application to generate the perturbations in the HPC trace during runtime is presented in the following section.

5 ADVERSARIAL HPC GENERATOR

In order to generate the required number of HPCs, we craft an application (benign) that spawns as a separate thread and generates the additional number of HPC events that makes the overall HPC count similar to the predicted HPC count by the adversarial HPC predictor discussed previously.

5.1 Adversarial HPC Generation

A pseudocode depicting the process of creating adversarial HPC is shown in Algorithm 1.

In Algorithm 1, we show the pseudo code to create adversarial LLC load misses and branch misses. The LLC load misses and branch misses are some of the pivotal microarchitectural events that malicious applications [19] or even side-channel attacks affect.Hence, we showcase a simple example of perturbing those in Algorithm 1, however, other events can also be perturbed.

In order to generate LLC load misses, an array of size n is initially loaded from the memory and flushed to generate LLC load misses. This is outlined in Line 2-12 of Algorithm 1. The experiments are repeated multiple times with different array sizes (n) and different number of elements flushed (k) to determine the number of LLC load misses generated. Further, a linear model is built to find the dependency of n and k on number of LLC load misses. As such,

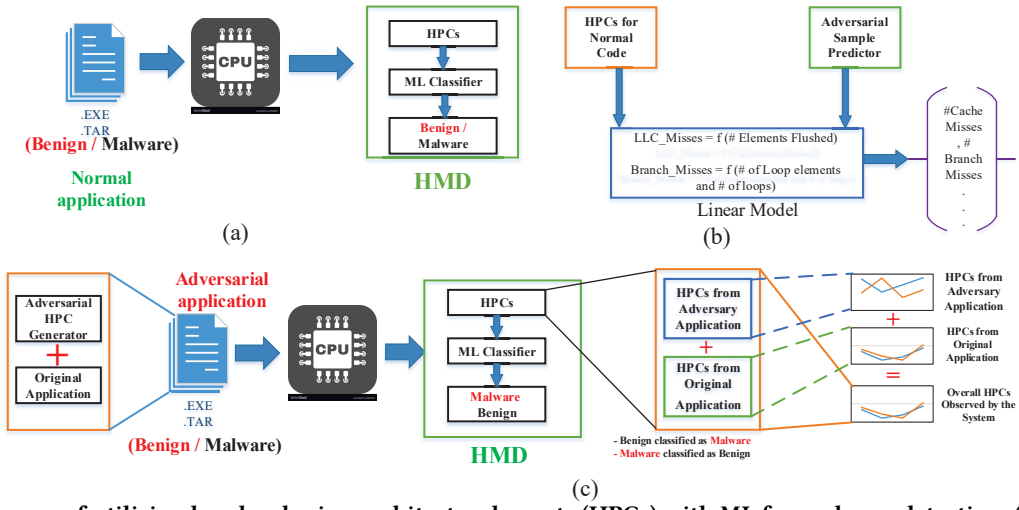


Figure 4: (a) Process of utilizing low-level microarchitectural events (HPCs) with ML for malware detection; (b) Determining parameter of adversarial code generator with the aid of adversarial HPC predictor; (c) process of adversarial HPC generator embedded into original application yet spawned as separate thread leading to adversarial output (misclassification)

Algorithm 1 Pseudocode for generating adversarial HPCs

Input: Application 'App()'
Output: Adversarial microarchitectural events

1: **cache_miss_function()** {Sample pseudo code that generates required number of adversarial LLC misses}
2: #define array[n] % Size of array and loop define amount of variation in HPCs
3: load i #0
4: *Loop 1:* cmp i #n {Compare i with n}
5: array[i]=i
6: jump Loop1
7: end
8: load i #0
9: *Loop 2:* cmp i #k {$k <= n$}
10: *ld rax &array[i] # load array address in register rax*
11: *cflush (rax)* {Clflush as a function of array and loop size}
12: *jump* Loop2
13: end
14: **branch_misses_function()** {Code that generates required number of adversarial branch instructions and branch misses}
15: #define int a, b, c, d
16: a<b<c<d<n
17: *Loop 3:* cmp i #a { ··· function ··· }
18: *Loop 4:* cmp i #b { ··· function ··· }
19: *Loop 5:* cmp i #c { ··· function ··· }
20: *Loop 6:* cmp i #d { ··· function ··· }
21: *Loop 7:* cmp i #n { ··· function ··· }
22: *jump* Loop 3; end ;
23: {Similar functions to generate other HPCs as predicted by adversarial sample predictor}
24: **APP()** {User/Attacker's application to be executed}

once the adversarial sample predictor predicts the number of LLC load misses to be generated to craft an adversarial sample, the n and k are accordingly determined. The rationale to employ a linear model is its low complexity, yet yielding high accuracy (<3% error) to determine the dependency between n and k for our experiments. It needs to be noted that as the LLC misses are dependent on the system, and random in nature, hence, we execute the application multiple times (100) with same n, and k and average the obtained LLC load misses to alleviate any errors caused.

Example: For instance, the crafted application similar to that depicted in Line 2-12 of Algorithm 1 with n and k set to 100K leads to an LLC load miss of 73K, whereas when n and k is set to 500K, around 287K LLC load misses are generated. The experiment is performed on Intel Core i7-8700K running Ubuntu 18.4, having GCC 7.3 version. The *Perf* tool available on Linux is utilized to obtain the HPC events. The flushing of the data has been verified by executing the attack code with and without flushing the cache lines - the execution time is around 1.5× when the data is flushed compared to the case when data is not flushed.

In similar manner, branch misses and branch instructions are generated as shown in Line 15-22 of Algorithm 1. To increase the branch misses, a set of conditional statements i.e., comparison statements are embedded into the application to create branch misses, as the number of branch instructions depend on the number of conditions to be checked. In the presented pseudo code, we have five conditional statements for generating branch-misses (Line 15-22).

For the attack code on branch miss events, with a loop size of 20K and integer values assigned to a, b, c and d based on the number of loops, as in Line 15-22 of Algorithm 1, the number of branch misses is around 255K. An increase in number of branch misses is observed with the addition of dummy loops that are designed to not satisfy the condition.

All these adversarial sample generators are spawned as separate threads along with the user or attacker's application that needs to be misclassified. In this manner, the adversarial HPC generator does not interfere with the original application's source code, yet is able to mislead the embedded defense mechanism. Figure 4(a) shows the HPC trace of a normal application, and the HPC trace

predicted by the adversarial sample predictor to misclassify the ML classifier is depicted in Figure 4(b). The process of adversarial HPC generation during runtime is depicted in Figure 4(c). If the predicted HPC values are smaller than that generated by original applications, we insert the delay elements to smoothen the HPC trace and reduce the HPC values. It needs to be noted using this process, we generate adversaries to classify benign as malware as well as malware as benign applications.

5.2 Summary

The proposed adversarial attack on microarchitectural events comprises of three phases. Firstly, we perform reverse engineering to build a ML classifier that mimics the functionality of the victim's HMD or malware detectors. Further, with the aid of adversarial sample predictor, the required number of HPC events to misclassify the applications is determined. To determine the parameters of adversarial generator application, a linear model relating different features of the application and the HPC events is built. Thus, based on the derived linear model and the required number of adversarial HPCs, the parameters of the adversarial HPC generator application (for instance variables *i*,*k*,*n* in Line 4 and Line 2 of Algorithm 1) are determined. Lastly, this crafted HPC generator application is spawned as separate thread together with normal (malware or benign) application, leading to overall HPCs generated by the modified application close to those predicted by the adversarial sample predictor, eventually leading to misclassification.

6 EXPERIMENTAL RESULTS

6.1 Experimental Setup and Data Collection

This section provides the details of the experimental setup and data collection process. The applications (both malware and benign) are executed on an Intel Xeon X5550 machine running Ubuntu 14.04 with Linux 4.4 Kernel. In order to extract the HPC information, we used *Perf* tool available under Linux. *Perf* provides rich generalized abstractions over hardware specific capabilities. It exploits *perf-event-open* function call in the background which can measure multiple events simultaneously. We executed more than 3000 benign and malware applications for HPC data collection. Benign applications include MiBench benchmark suite [9], Linux system programs, browsers, text editors, and word processor. For malware applications, Linux malware is collected from virustotal.com [2] and virusshare.com [1]. Malware applications include five classes of malware comprising 607 Backdoor, 532 Rootkit, 2739 Virus, 1264 Worm and 7221 Trojan samples. The adversarial sample predictor is implemented in Python using the Cleverhans library. The linear model is derived using the traditional statistical curve fitting technique. The adversarial sample generator is implemented using C and executed on a Linux terminal as a shell script that facilitates to execute the user/attacker's application in parallel. The hyper parameters of the neural network mimicing the victim's HMD or security defense and the parameters used for adversarial sample predictor are outlined in Table 1.

6.2 Impact of Adversarial Attack on HPCs

We depict the impact of adversarial sample generator (application) on the generated HPC events in Figure 5 and 6. Figure 5 shows the LLC load misses of a benign application (notepad++). The Figure 5(a) shows the LLC load misses in normal case. For this HPC pattern, the adversarial HPC pattern predicted by the adversarial sample

Table 1: Architectural details of HMD

Parameters of ML classifier in HMD			
Input	16 features	Optimization	ADAM
# hidden layers	1	Batch size	128
Hidden layer 1 (ReLu)	250 neurons	Epochs	100
Dropout	0.2	Learning rate	0.001
Adversarial Sample Predictor Parameters			
Attack type		FGSM	
Adversarial perturbation		0.3	

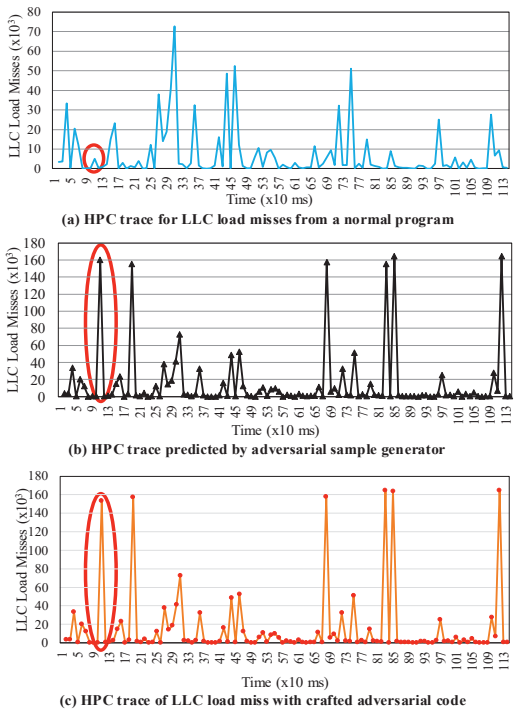
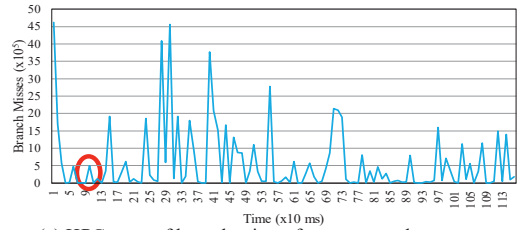


Figure 5: (a)LLC load miss HPC trace of an application; (b) LLC load miss HPC trace of the application predicted by adversarial sample predictor; and (c) LLC load miss HPC trace of the application predicted by adversarial sample generator

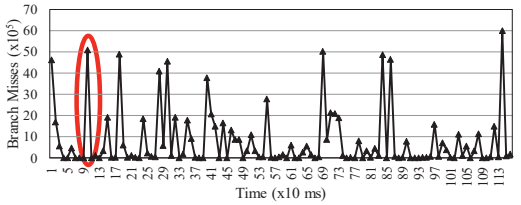
predictor (implemented in Python) is shown in Figure 5(b). One can observe that there exist some spikes in the pattern compared to the normal HPC pattern, as marked by circle. Figure 5(c) shows the HPC pattern generated when the application is integrated with the adversarial HPC generator. On an average, there is an error of 2.23% between the trace predicted by the adversarial sample predictor and the trace generated by the adversarial sample generator.

In a similar manner, we depict the branch misses in Figure 6. Figure 6(a) shows the HPC pattern of branch misses for a normal application (notepad++). The adversarial pattern predicted and generated by adversarial sample generator for branch misses is shown in Figure 6(a), and 6(b) respectively. One can observe that pattern predicted by the adversarial sample predictor and generator are similar. An average error of 2.15% is observed for branch misses, and 0.91% for branch instructions. A 2.23% error is observed for branch miss instruction. This indicates that adversarial generator can efficiently generate the required number of HPCs without being detected by the malware detectors.

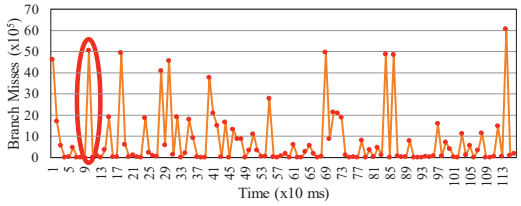
The neural network based HMD achieves an accuracy of 82.76% with normal samples. However, when the applications are integrated with the proposed adversarial sample generator application,



(a) HPC trace of branch misses from a normal program



(b) HPC trace predicted by adversarial sample generator



(c) HPC trace from crafted adversarial code

Figure 6: (a) Branch miss HPC trace of an application; (b) Branch miss HPC trace of the application predicted by adversarial sample predictor; and (c) Branch miss HPC trace of the application predicted by adversarial sample generator

the accuracy reduces to 18.04%. Similarly, a drastic reduction in precision, F1-score and recall are observed with the proposed attack on different applications. This is outlined in Table 2.

Table 2: Impact of adversarial attack on HMD

	Accuracy	Precision	F1-score	Recall
Before	82.7%	80.0%	80.0%	83.0%
After	18.3%	45.0%	10.0%	18.0%

6.3 Transferability Analysis

Though the reverse engineering results in building ML classifier that mimics the victim's HMD, they might not be same. For instance, victim's HMD might be using a logistic regression (LR) and the reverse engineered solution is a neural network. To showcase the robustness of the proposed adversarial malware crafting, we perform a transerability analysis. As stated in [14], LR and neural network achieves good performance and robust. Hence, we perform the transferability analysis of the generated adversarial malware on the LR based HMD. Thus, the adversarial malware generated is applied to a HMD using logistic regression, whose functionality is mimicked through reverse engineering. The results show that the malware detection accuracy falls to 5.10% with precision, F1-score, and recall to 16.0%, 7.0% and 5.0% respectively with the adversarial malware. This indicates that the ML classifier used to craft the adversarial malware is transferable to other systems as long as we can mimic the victim's malware detector functionality.

7 CONCLUSION

In this work, we propose an adversarial attack on microarchitectural event based malware detection systems i.e., HMD systems. These HMD systems utilize the underlying hardware performance counters to capture the microarchitectural events and provide them to ML classifier for detecting and classifying malware. This work employs an adversarial sample predictor to determine the HPC count required to get misclassified. Post determining the required number of HPC count, using the proposed adversarial sample generator the required number of additional HPC count is generated without intervening with the original application and eventually leading to misclassification. An error of $< 3\%$ in predicted and generated HPC events to create adversary is observed. Furthermore, the malware detection accuracy is reduced from 82.7% to 18.04%.

REFERENCES

[1] 2019. VirusShare Team. www.virusshare.com Last accessed: 04-May-2019.
[2] 2019. Virustotal intelligence service. www.virustotal.com/intelligence Last accessed: 04-May-2019.
[3] M. B. Bahador, M. Abadi, and A. Tajoddin. 2014. HPCMalHunter: Behavioral malware detection using hardware performance counters and singular value decomposition. In *Int. Conf. on Computer and Knowledge Engineering.*
[4] F. Brasser and et al. 2018. Advances and Throwbacks in Hardware-assisted Security: Special Session. In *Int. Conf. on CASES.*
[5] J. Demme and et al. 2013. On the Feasibility of Online Malware Detection with Performance Counters. *SIGARCH Comput. Archit. News* 41, 3 (Jun 2013), 559–570.
[6] S. Dinakarrao and et al. 2019. Lightweight Node-level Malware Detection and Network-level Malware Confinement in IoT Networks. In *Design Automation and Test Con. in Europe.*
[7] A. Garcia-Serrano. 2015. Anomaly Detection for Malware Identification using Hardware Performance Counters. *CoRR* abs/1508.07482 (2015).
[8] I. Goodfellow, J. Shlens, and C. Szegedy. 2015. Explaining and Harnessing Adversarial Examples. In *International Conference on Learning Representations.*
[9] M. R. Guthaus and et al. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *IEEE Int. W. on Workload Characterization.*
[10] A. Huang and et al. 2018. Adversarial Deep Learning for Robust Detection of Binary Encoded Malware. *CoRR* abs/1801.02950 (2018).
[11] G. Jacob, H. Debar, and E. Filiol. 2008. Behavioral detection of malware: from a survey towards an established taxonomy. *Journal in Computer Virology* 4, 3 (Aug 2008), 251–266.
[12] A. Jafari and et al. 2019. SensorNet: A Scalable and Low-Power Deep Convolutional Neural Network for Multimodal Data Classification. *IEEE Tran. on Circuits and Systems I* 66, 1 (Jan 2019), 274–287.
[13] Kaspersky. 2017. Advanced Threat Defense and Targeted Attack Risk Migration. *White Paper* (2017), 1–12. https://media.kaspersky.com/en/business-security/enterprise/KL_KATA_Whitepaper_OG.pdf.
[14] K. Khasawneh and et al. 2017. RHMD: Evasion-resilient Hardware Malware Detectors. In *IEEE/ACM Int. Symp. on Microarchitecture.*
[15] K. Khasawneh and et al. 2018. EnsembleHMD: Accurate Hardware Malware Detectors with Specialized Ensemble Classifiers. *IEEE Trans. on Dependable and Secure Computing* (2018).
[16] M. Khatwani and et al. 2018. Energy Efficient Convolutional Neural Networks for EEG Artifact Detection. In *IEEE Biomedical Circuits and Systems Conf.*
[17] Y. Liu, X. Chen, C. Liu, and D. Song. 2017. Delving into Transferable Adversarial Examples and Black-box Attacks. In *Int. Conf. on Learning Representations.*
[18] N. Papernot and et al. 2016. The Limitations of Deep Learning in Adversarial Settings. In *IEEE European Symp. on Security and Privacy.*
[19] N. Patel, A. Sasan, and H. Homayoun. 2017. Analyzing Hardware Based Malware Detectors. In *Design Automation Conf.*
[20] H. Sayadi and et al. 2018. Comprehensive Assessment of Run-Time Hardware-Supported Malware Detection Using General and Ensemble Learning. In *ACM Computing Frontiers.*
[21] H. Sayadi and et al. 2018. Ensemble Learning for Effective Run-time Hardware-based Malware Detection: A Comprehensive Analysis and Classification. In *Design Automation Conference.*
[22] H. Sayadi and et al. 2019. 2SMaRT: A Two-Stage Machine Learning-Based Approach for Run-Time Specialized Hardware-Assisted Malware Detection. In *Design Automation and Test Con. in Europe.*
[23] C. Szegedy and et al. 2014. Intriguing Properties of Neural Networks. In *Int. Conf. on Learning Representations.*
[24] A. Tang, S. Sethumadhavan, and S. Stolfo. 2014. Unsupervised Anomaly-Based Malware Detection Using Hardware Features. In *RAID Conference.*
[25] X. Wang and et al. 2015. ConFirm: Detecting Firmware Modifications in Embedded Systems Using Hardware Performance Counters. In *IEEE/ACM Int. Conf. on Computer-Aided Design.*
[26] B. Zhou and et al. 2018. Hardware Performance Counters Can Detect Malware: Myth or Fact?. In *ACM Asia Conf. on Computer and Communications Security.*