Detecting Malicious Attacks Exploiting Hardware Vulnerabilities Using Performance Counters

Congmiao Li, Jean-Luc Gaudiot
Electrical Engineering and Computer Science
University of California, Irvine
Irvine, USA
congmial@uci.edu, gaudiot@uci.edu

***Abstract*—Over the past decades, the major objectives of computer design have been to improve performance and to reduce cost, energy consumption, and size, while security has remained a secondary concern. Meanwhile, malicious attacks have rapidly grown as the number of Internet-connected devices, ranging from personal smart embedded systems to large cloud servers, have been increasing. Traditional antivirus software cannot keep up with the increasing incidence of these attacks, especially for exploits targeting hardware design vulnerabilities. For example, as DRAM process technology scales down, it becomes easier for DRAM cells to electrically interact with each other. For instance, in *Rowhammer* attacks, it is possible to corrupt data in nearby rows by reading the same row in DRAM. As *Rowhammer* exploits a computer hardware weakness, no software patch can completely fix the problem. Similarly, there is no efficient software mitigation to the recently reported attack *Spectre*. The attack exploits microarchitectural design vulnerabilities to leak protected data through side channels. In general, completely fixing hardware-level vulnerabilities would require a redesign of the hardware which cannot be backported. In this paper, we demonstrate that by monitoring deviations in microarchitectural events such as cache misses, branch mispredictions from existing CPU performance counters, hardware-level attacks such as *Rowhammer* and *Spectre* can be efficiently detected during runtime with promising accuracy and reasonable performance overhead using various machine learning classifiers.**

Keywords—security, machine learning, malware detection, microarchitectural features

I. Introduction

Traditional antivirus (AV) software is normally used to protect computer systems from cyber attacks targeting software vulnerabilities above the operating system (OS) level. However, more recent attacks targeting computer architecture and hardware design flaws cannot be detected by such antivirus tools since the attacks are not based in the software application or the OS and do not leave traces in system log files. Therefore, such attacks are serious security risk for all systems using these flawed hardware designs.

Common computer hardware components such as the memory may behave unexpectedly under specific conditions and can be exploited by malicious attackers. For example, the *Rowhammer* [35] vulnerability is a hardware design defect in DRAM chips that allows attackers to flip bits in memory at unauthorized locations by repeatedly accessing physically adjacent rows within a DRAM refresh interval rapidly. *Rowhammer* can be exploited to mount attacks and bypass existing software security and trust features including memory isolations. Increasingly advanced *Rowhammer* attacks could compromise various platforms including standalone personal computers [36], web browsers [37, 38], virtual machines in clouds [39, 40], and mobile devices [41]. Such attacks can be used to escalate privileges [36, 37, 41], break cryptographic keys [39], compromise remote systems over the network [42], or lock down a processor by denial-of-service attack [43].

Further, information can be leaked to unprivileged parties by analyzing the differences in unintended side channels. These unintentional side channels can use timing information [1-4], power consumption [5-7], electromagnetic radiation [8,9], light emission [10], or even sound [11]. In microarchitectural side-channel attacks, malicious processes attempt to interfere with the victim through shared microarchitectural resources. The interference pattern such as cache timing [12-15], branch prediction history [16,17], or Branch Target Buffers [18,19] can then be exploited to infer secrets.

Speculative execution reduces the processor idle time to improve performance by executing predicted path prematurely according to the execution history before the actual path is known definitively. *Spectre* attacks [20, 21] exploit speculative execution by tricking the processor into taking the wrong branch, instructions associated with the malicious branch execution path can be carefully crafted to leak the victim's memory or register contents through microarchitectural side channels. Kocher *et al.* [21] demonstrated the use of cache side channels for their attack implementation, although there could be other side channels. *Spectre* attacks works well on various Intel, AMD and ARM processors. Furthermore, the KAISER software patch [22] to mitigate *Meltdown* attacks cannot defend against *Spectre* attacks. To completely solve the problem, changes to the processor design and instruction set architectures (ISAs) are required.

Unlike antivirus software which scans suspicious instructions in binaries and system logs files in a static fashion, recent research has shown that malicious software and attacks can be detected in modern processors through the use of dynamic microarchitectural execution patterns gleaned from widely available hardware performance counters (HPC). Demme *et al.* [23] adopted offline analysis based on various supervised machining learning algorithms to show that malware classification is possible using a complete trace of the

Acknowledgments -This work is partly supported by the National Science Foundation (NSF) under Grant No. CCF-1763793/3654. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

program behavior after execution. Compared with higher-level features in the OS and application, these microarchitectural features have less complexity and are easier to collect with low overhead. These features include events such as cache miss rate, branch misprediction rate, data reference patterns *etc* and could be highly effective in detecting attacks targeting hardware vulnerabilities as such attacks happen below the OS level.

In this paper, we demonstrate the use of low-level microarchitectural features to detect *Rowhammer* and *Spectre* during the attacks. As opposed to traditional static signature-based AV, this hardware performance counter based detection approach monitors the dynamic behavior of the system in the hardware level with low overhead and allows the system administrator to, effectively and in real-time, catch malicious code execution especially attacks exploiting hardware vulnerabilities such as *Rowhammer* and *Spectre*. To this end, we collected microarchitectural traces from a Linux system based on an Intel x86 processor in a "clean" environment and in a system under attack, respectively. In particular, because of the nature of *Rowhammer* and *Spectre* attacks which may potentially alter the branch prediction or cache miss behaviors of the victim's processor, we monitor those related features from performance counters and then use machine learning methods to train and test classifiers to detect the attack. Unlike previous offline detection method using the entire program traces after attack [23], we propose to use the Weighted Moving Average (WMA) of the time series data from the output of the classifier to make real-time decisions successively over sliding windows of program execution. The online detector can catch 100% of both attacks with 0% false positives for *Rowhammer* and 0.77% false positives for *Spectre* in the best-case scenario.

The rest of the paper is organized as follows. We first introduce *Rowhammer* and *Spectre* attacks in section II. Understanding how the attacks work helps us to select the hardware performance counter features for effective detection. In section III, we compare different machine learning algorithms to train the base classifier and introduce the metrics to evaluate the performance of classifier. Then an online detection approach that uses the output from the trained base classifier is proposed. In section IV, detailed experimental setup to collect the selected microarchitectural events in clean environment and in system under attack is described. The detection performance results for *Rowhammer* and *Spectre* attacks are presented in section V. Finally, we conclude in section VI.

II. Background

Exploits targeting hardware-based vulnerabilities constitute serious risks for wide range of computer systems as they can bypass most of the existing defense mechanisms. For example, with increased density and reduced reliability of DRAM, *Rowhammer* attacks allow data corruption in memory by repeatedly accessing the same row in DRAM. *Spectre* exploits critical modern processor design flaws caused by speculative execution to allow attackers to steal sensitive information from users' devices through microarchitectural side channels. In this section, we discuss in detail the above-mentioned attacks.

A. *Rowhammer*

To increase capacity and reduce energy consumption, DRAM chips have to be physically denser. However, smaller cells can only hold a lower charge which reduces the noise margin. Higher proximity of cells also introduces electromagnetic coupling effects. Therefore, every time a DRAM row is read from a memory bank, the memory cells in adjacent rows leak a small amount of charge. If this happens frequently within one refresh cycle, the affected cells can potentially leak enough charge to cause the stored bit value to flip, a phenomenon known as *Rowhammer* [35].

In singled-sided *Rowhammer* attacks [35], only one side of the neighboring row to the victim row is rapidly accessed. A newer version, namely double-sided *Rowhammer* [36], hammers both sides of the neighboring rows to increase the possibility of bit flip. To successfully trigger bit flip from CPU, memory accesses have to bypass CPU caches and reach the target row in DRAM as frequently as possible. This could generate abnormally large number of cache misses during a short time period. To avoid caches, attackers can use cache flushing instructions [36, 39, 40], eviction buffers [37, 38, 45], and non-temporal store instructions [47].

B. *Countermeasures to Rowhammer*

Due to the possible catastrophic damages caused by *Rowhammer* attacks, solutions to protect users from such attacks are urgently needed. Hardware solutions to DRAM scaling challenges such as error correcting code or target row refresh [44] may be able to protect future memories from *Rowhammer* attacks, but cannot be deployed in existing hardware. Software-based protections [36, 37, 45, 46], which can be implemented on existing systems, cannot defend against all types of advanced *Rowhammer* attacks.

Previous research [45] has shown the feasibility of using hardware performance counters to monitor the last-level cache miss rate and compared it with a fixed threshold to detect *Rowhammer* attacks. To reduce the number of false positives, we proposed to train the base classifier with models of different complexity and use the Weighted Moving Average (WMA) from the output of the base classifier over a sliding window during program execution. In this paper, we take *Rowhammer* as an example to demonstrate the effectiveness of microarchitectural features to detect attacks exploiting hardware vulnerabilities and to build the foundation for future works on more advanced *Rowhammer* and other attacks.

C. *Description of Spectre*

```
void victim_function (size_t x) {
   if (x < array1_size) {
        temp &= array2[array1[x] * 512];
}}
```

Example 1: Conditional Branch Example

As reported in [20, 21], *Spectre* has two variants: bounds check bypass and branch target injection. The first variant exploits conditional branch mispredictions. For example, the victim function in Example 1 receives an integer x from an untrusted source. The function does a bound check on x to prevent the process from reading unauthorized memory outside *array1* to ensure security. However, speculative execution can lead to out-of-bounds memory reads. Suppose the attacker makes several calls to *victim_function()* to train the branch predictor to expect taking the branch by feeding it with valid values of *x*, then calls the same function with an out-of-bound *x* that points to a secret byte in the victim's memory.

The attack usually consists of three phases. In general, it starts with the setup phase where the attacker prepares the side channel to leak the victim's sensitive information, and other necessary pre-requisites such as to mis-train the branch predictor to take erroneous execution path, and to load target memory location into registers, *etc*. In the following phase, the attacker diverts confidential information from the victim's context to a microarchitectural side channel by exploiting different hardware vulnerabilities such as out-of-order execution or speculative execution. Then during the final phase, the attacker gains access to the secret data through the prepared side channel in the previous stages.

In the setup phase of *Spectre* attacks, the adversary prepares for three conditions. Firstly, a malicious out-of-bound value of *x* is chosen such that *array1[x]* points to a secret byte *s* in the victim's memory. The cache is then configured so that *array1_size* and *array2* are not in the cache, but the secret *s* in the cache. Lastly, the branch predictor is trained to expect the condition $x <$ *array1_size* to be true.

During the second phase, the victim function starts by comparing the malicious value of *x* with *array1_size*. Because *array1_size* is not cached, the mis-trained branch predictor assumes the condition to be true while waiting for the value of *array1_size* to be loaded from memory. The processor speculatively executes the instruction inside the *if* condition. Reading secret data *s* from *array1[x]* is fast because it is a cache hit. Then s is used to compute the address of *array2[s*512]* and the program quickly starts to read data from this address in memory. While waiting for the read to return, the value of *array1_size* has finally arrived. The processor then realizes the branch predictor selected the wrong execution path which compels it to roll back the register states. However, the cache state of *array2* has already been affected by the memory read from the specific address related to the secret *s* and is not reversed.

In the final phase of the attack, using cache side channel attacks such as the method proposed in [13] can identify which cache line in *array2* was loaded by measuring the timing differences. Such information can then be used to infer the value of the secret byte.

The second variant of *Spectre* targets the indirect jump target prediction. It allows the program to jump to an address contained in a register, in a memory location, or in the stack. Like the first variant, if the target address is not in the cache, the attacker can train the branch predictor to jump to an erroneous location.

The two variants discussed above rely on the changes in the state of the cache caused by the speculative execution. There could be potential other variants of the attack. In general, any observable changes from speculatively executed code may cause the leaking of confidential information.

D. *Mitigating Spectre*

Mitigating the effects of *Spectre* is difficult because there are many variations of possible attacks. To prevent conditional branch vulnerability, speculative execution needs to be stopped on all potentially sensitive execution paths. However, insertion of such blocking mechanisms in all conditional branches and their destinations by compiler would severely degrade performance. Software mitigation to indirect branch vulnerability is even more challenging as there is no architecturally-defined method to block it and indirect jumps vary across different processors.

E. *Detecting Spectre*

Besides patching the systems, it is also important to proactively detect malicious attacks and stop them as early as possible. To our knowledge, no existing research shows the effectiveness of detecting *Spectre* using hardware performance counters. Despite the fact that many different variants of *Spectre* attacks exist, they all entail training the branch predictor to take the wrong execution path, and then leak the confidential information through an observable microarchitectural side channel.

For the case of conditional branch example in Example 1, the attacker calls the victim function multiple times so as to cause the condition to be true. In turn, this means that the branch misprediction rate will be reduced during the attack. In the final step of the attack, the secret data are leaked through cache side channels. The attacker needs to constantly flush the cache to make sure *array2* and *array1_size* are not cached, which means that the cache miss rate will likely rise. Consequently, monitoring the deviation of these two microarchitectural behaviors is one possible hint as to how to detect the attack.

To further validate our hypothesis, we conducted experimental attacks based on the proof of concept code in [21] and periodically collected microarchitectural traces from hardware performance counters. An analysis of these results is presented in subsequent sections. The proposed detection method can be further extended to other variations of *Spectre* attacks by monitoring additional microarchitectural features depending on the side channels likely to be exploited.

Attackers may attempt to evade detection by reverse-engineering the detector and mimicking the behavior of normal programs. Several researchers (including Khasawneh *et al.* [48]) have developed this kind of evasive malware by adding instructions into the control flow graph of the malware without changing the execution state of the program. However, for *Spectre* attacks to avoid detection, attackers must also ensure that cache and branch predictor states are not affected, which we believe would be extremely difficult to effectively design for several reasons. First the attack itself is very time consuming: for each bit the attacker must perform multiple training rounds on the branch predictor followed by cache side channel attacks. Adding additional delays may make the attack extremely slow. Second, adding delays may make the cache side channel ineffective as other running programs may change the cache status during the delay. Third, as the attack is hardware specific, it is very difficult to design evasive attacks that work for different machines. In addition, if an attacker could still hide a *Spectre* attack from detection without slowing it down, we could adopt the evasion-resilient detection method proposed in [48] and randomize the performance counter data collection periods to defend such evasion.

III. Proposed Online Detection Approach

Our proposed detection approach first collects microarchitectural features from performance counters in every sampling period. For *Rowhammer* and the variant of *Spectre* attacks discussed in the previous section using conditional branches, we choose to monitor 4 events related to cache access and branch prediction behaviors, namely cache references, cache misses, branch instructions retired and branch mispredictions. The data are collected in a "clean" environment where the computer runs typical desktop applications like web browser, video player, and text editors, as well as in an environment under attack (the same desktop applications are run in the two cases). The data are then labeled to train the machine learning classifier offline. At runtime, the trained classifier classifies the input data at each time interval and the output time series is fed into the online detection mechanism to decide whether the system is under attack. The online detector is implemented as a software thread and continuously running on a separate core to collect performance counter data and classify malicious attacks. This section describes the machine learning algorithms we used to train the classifier and our proposed online detection approach in details.

A. *Machine Learning Classifiers*

Machine learning algorithms can be used to train classifiers that determine the class *y* to which a given data set *x* belongs. In most cases, the relationship between *x* and *y* is described by a probability distribution *P(x,y)*. The optimal class membership decision is to choose the class label *y* such that the posterior distribution *P(y|x)* is maximized [24]. We use supervised learning [25] to train the attack detector with a set of pre-labeled examples. Supervised learning entails a training phase and a testing phase.

For each type of attacks, we collect data in 10 independent runs and use the same number (1,200) of samples from both classes to avoid any bias. Then, we randomly divide the collected data into training (80%) and test (20%) data, then separate the training data into training (80%) and validation (20%) data.

We choose the following three different machine learning algorithms to build the classifier with increasing complexity of the model (many other machine learning algorithms are available, but for demonstration purposes, we chose 3 commonly used ones with different complexities according to the data size):

1) Logistic Regression (LR): LR is a simple linear classification algorithm. It attemps to separate multi-dimensional input data points by hyperplanes where points on one side of the plane belong to the "normal" class and points on the other side belong to the "malicious" class. In general, the programs are not linearly separable, so LR gives a probability between 0 and 1 for the likelihood of a program trace being malicious. This probability is then converted into a binary decision by comparing it with a pre-defined threshold. Compared with other non-linear models, LR has fewer parameters and requires less time to train.

2) Support Vector Machine (SVM): SVM finds the optimal separating boundaries between data sets by modeling and solving the classification problem as a constrained quadratic optimization problem [26, 27]. The degrees of nonlinearity and flexiblity can be adjusted by using different kernel functions such as polynomial kernel, radial basis function kernel (RBF), etc. The classification result is dichotomous where the membership function could be either 0 or 1 without probability distribution. SVM has received considerable research interest over the past years because its performance is comparable with other non-linear models for many classification problems and it is less complex than artifical neural networks.

3) Artificial Neural Networks (ANN): ANNs consist in networks of perceptrons (Multilayer Perceptron - MLP) that approximate a classification function for the training data. ANNs usually contain an input layer, an output layer, and multiple hidden layers of perceptrons in between. It is a popular and promising machine learning technique due to its capability of mapping highly nonlinear data samples unlike any other statistical regression models. When there is no hidden layer, the network is actually identical to the LR model if the logistic activation function is used [28, 29]. By introducing nonlinear hidden neurons to the network, the output of the network can become a nonlinear function of the inputs. In classification problems, this can model the problem of nonlinear decision boundaries. Researchers have developed many models based on Back Propagation Networks (BPN) and Radial Basis Function Networks (RBFN) for highly nonlinear time series predictions [30, 31]. In general, ANN is more flexible than LR to model more complex data. However, it requires much longer time and more data to train the model.

B. *Online Attack Detection*

We use similar sliding-window based online classification methods as proposed in [49] to detect malicious behaviors at runtime. To this end, we collect microarchitectural features periodically (every 100 ms). The multidimensional data are then feed to a machine learning classifier to make decisions as to whether malicious code is being executed or not. The problem of detecting malicious attacks in real time is to make decisions according to the binary time series generated by the base classifier.

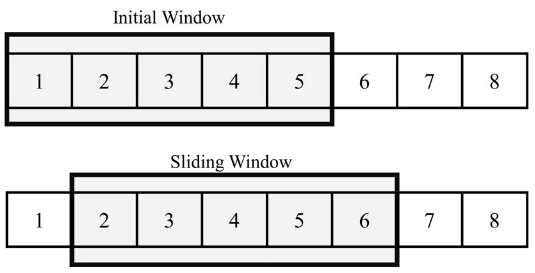

Fig. 1. Example of sliding window.

To smooth the fluctuated time series data, a Weighted Moving Average (WMA) is used to filter out noise for better decision making by assigning a weight factor to each element in the time series. More recent data are assigned with higher weights. Then we segment the data using a sliding window [32, 33] to calculate the average of consecutive decisions within the current window. If the average is above a certain threshold, we conclude that an attack (malicious code) is in progress. Fig. 1 illustrates an example of a sliding window process with a window size of 5. Each numbered segment corresponds to the classification result of each sampling period. The initial window contains the first 5 decisions. The data within this window are used to determine the final classification result for the current time. The detection runs continuously for the next period where the window slides to the right by one segment to cover data from segment 2 to 6, and it moves on to the next window accordingly.

In our experiments, we choose the window size to be 10 and sampling period of performance counters data collection to be 100 ms, which means our detector makes decision on whether the system is under attack every second. While window size and sampling period could affect detection accuracy and overall system performance overhead for different hardware systems, we selected the above numbers so as to yield satisfiable detection accuracy without a major slowdown of the system. In future work, we plan to use a variety of windows sizes and sampling periods to improve the resilience of our detector and keep the performance overhead minimum.

C. *Evaluation of Detection Performance*

A key criterion to evaluate the detection performance is the accuracy of the model used to make decisions on previously unseen data. To characterize accuracy, we use metrics such as False Positives (FP) which is the percentage of misclassified malicious instances, and False Negatives (FN) which is the percentage of misclassified normal instances. A detection approach with good performance is expected to minimize both.

To visualize the tradeoff between percentage of correctly identified malicious instances and the percentage of normal instances misclassified, we use Receiver Operating Characteristic (ROC) graphs plotting True Positives (TP = 100% - FN) against FP. In addition, to compare the performance of different models, we compute and compare the area under the ROC curve for each model. The Area Under Curve (AUC) score, also known as the c-index, provides a quantitative metric of how well an attack detection approach can distinguish between malicious and normal execution with a higher AUC value for better performance.

IV. Experimental Setup

To examine whether performance counter data can be used to effectively detect attacks targeting hardware design vulnerabilities, we take *Rowhammer* and *Spectre* attacks as examples and collect events which have the potential to be affected by the attacks (this includes cache references, cache misses, branch instructions retired and branch mispredictions from running the attacks on top of normal programs and running typical benign applications alone respectively). The results are then analyzed and preprocessed before being used to train different classifiers to detect malicious behavior. In this section, we describe the details of the data collection mechanism and the system settings under attack and in normal conditions. In this paper, we demonstrate the proposed online detector under a standalone personal computer environment, but our work could easily be extended to server environment in the future.

A. *Data Collection Mechanism*

We run the attack on a typical personal laptop with Debian Linux 4.8.5 OS on Intel® Core™ i3-3217U 1.8 GHz processor with 3MB cache and 4GB of DDR3 memory from Micron. The Intel processor contains a model-specific performance counter monitor (PCM) and can be configured to count four different hardware events at the same time. According to the discussion on the nature of *Rowhammer* and *Spectre* attacks in Section II, we choose the following available events for our system:

- Last-level cache reference event (LLC references)
- Last-level cache misses event (LLC misses)
- Branch instruction retired event (branches)
- Branch mispredict retired event (branch mispredictions)

We use the standard profiling infrastructure of Linux, *perf* tools, to obtain system-wide performance counter data. We run perf every 100 *ms* to record the required data without excessively degrading the performance.

B. *Test Environment Setup*

In a clean environment, we sought to create realistic scenarios by randomly browsing popular websites (according to Wikipedia in FireFox) in different orders and by streaming videos from browser plug-ins. In addition, we also ran text editors to read and edit files. For data collection when the system is under malicious attack, we launched malicious attacks on top of normal running applications. To demonstrate the effectiveness of detecting attacks exploiting hardware vulnerabilities, we chose the double-sided *Rowhammer* attack and the *Spectre* proof of concept attack and ran them independently. The system status is reset after each run to ensure the measurements are independent across different clean and exploit runs. We collect overall performance counter data across the system rather than for individual processes. This is to make the classification problem closer to real world conditions, albeit more difficult.

V. Results

In the experiment described in section III, we collected data from four performance counters at the same time periodically in 10 separate runs. Each run produced 1,200 malicious and normal samples respectively. In this section, we first analyzed the collected raw data to assess whether it is feasible to differentiate measurements in a clean environment from those under attack by visualizing the distribution of data. Then, we used different machine learning algorithms to train the classifier and build the real-time attack detector using the sliding window approach discussed previously.

A. Data Distribution Analysis

Our data collection mechanism produces 4-dimensional time series data. Each sample contains event counts for branch mispredictions, LLC misses, branches, and LLC references during the sampling period. We also calculate the branch miss rate (1) and LLC miss rate (2) for each interval as:

$$\textit{branch miss rate} = \textit{branch mispredictions} / \textit{branches} \quad (1)$$

$$\textit{LLC miss rate} = \textit{LLC misses} / \textit{LLC references} \quad (2)$$

We use *boxplot* to visualize the range and variance of the measured data for each individual microarchitectural feature. Fig. 2 and Fig. 3 give a direct indication of the feasibility to detect malicious attacks using one particular feature.

For *Rowhammer* attacks as shown in Fig. 2, the LLC misses, LLC references and LLC miss rate are all concentrated in regions higher during the attack than during normal operations. The reason is that in order to successfully flip bits in the memory, the attacker has to bypass the cache and access the neighboring rows next to the victim row in DRAM in rapid sequence. This results in more cache misses within a short time of period. Unlike previous work which only considers the impact on cache behavior, we also notice that the number of branch miss rate is greatly reduced. This is because the exploit keeps looping through the same instructions to repeatedly access the same rows in memory. This extra feature could help to reduce the error rate in the detection of *Rowhammer*.

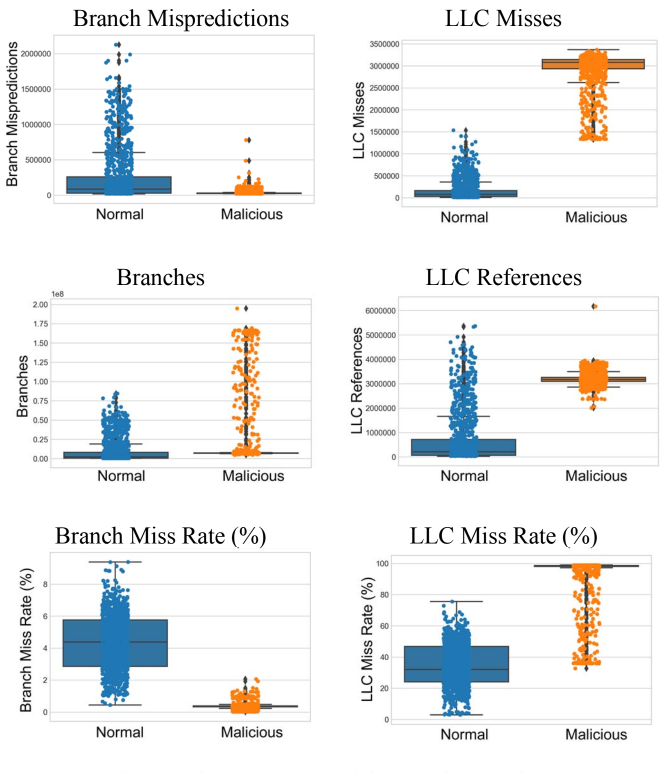

Fig. 2. Distribution of microarchitectual features from performance counters for *Rowhammer.*

For the *Spectre* attacks shown in Fig. 3, we observe an increased number of branches and branch mispredictions during the attacks. In contrast, the branch miss rate is decreased. This is because the attacker tries to train the branch predictor by calling the conditional branch many times with different input values that make the condition true. For the LLC, the number of references and misses are both increased with the miss rate concentrated on the higher percentage region due to cache side channel attacks. The experimental results validate our hypothesis proposed in section II.

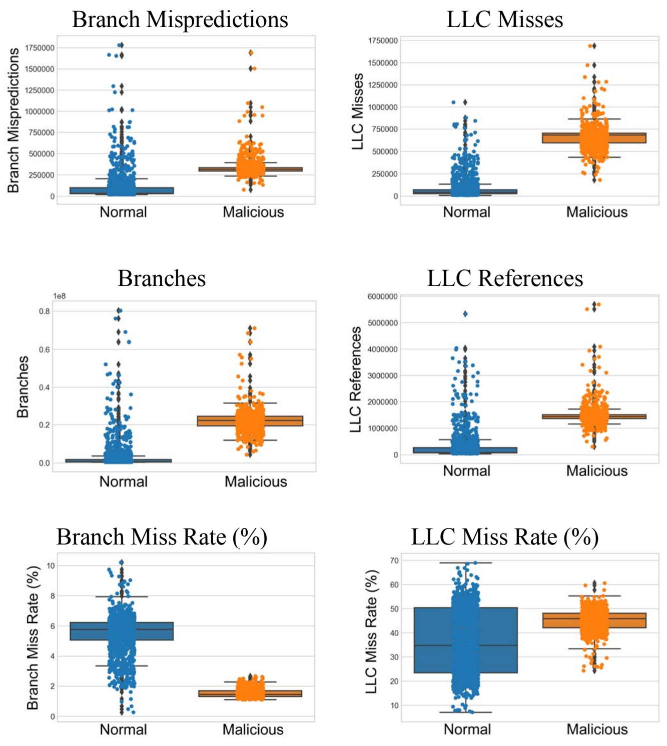

Fig. 3. Distribution of microarchitectual features from performance counters for *Spectre.*

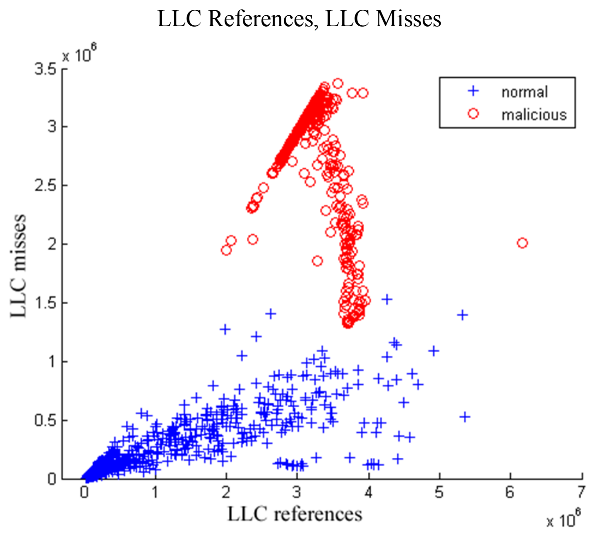

Fig. 4. Distribution of branch miss rate and LLC miss rate features for *Rowhammer.*

In addition, we also analyze the feasibility of distinguishing the collected performance counter data using more than one feature by plotting the sample points in 2D and 3D graphs with each dimension corresponding to one feature.

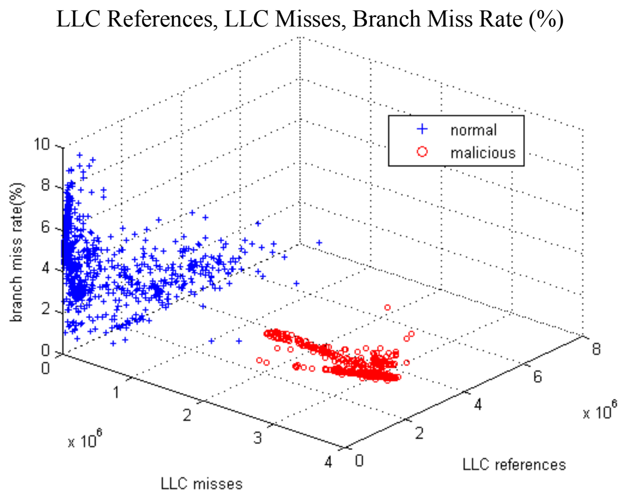

Fig. 5. Distribution of LLC references, LLC misses and branch miss rate features for *Rowhammer*.

Fig. 4 shows the distribution of normal and malicious sample points for *Rowhammer* attacks using only cache related features including LLC references and LLC misses. Fig. 5 uses LLC references, LLC misses and branch miss rate. We can observe the data points of two different classes distribute in two different regions and the boundaries between the two are obvious in both plots. Therefore, we believe it is feasible to use the chosen microarchitectural features to detect *Rowhammer* attacks. In particular, with an additional branch feature, the boundary is clearer in Fig. 5 which can lead to better classification performance.

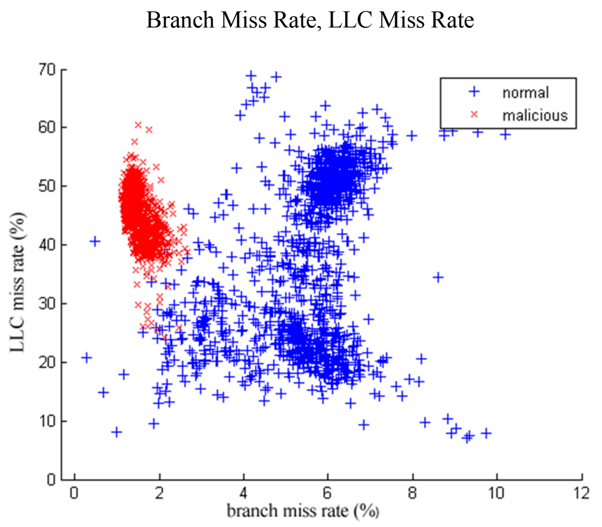

Fig. 6. Distribution of branch miss rate and LLC miss rate features for *Spectre*.

Similarly, Fig. 6 shows the distribution of normal and malicious sample points for *Spectre* attacks using branch miss rate and LLC miss rate parameters. Fig. 7 uses LLC references, LLC misses and branch miss rate. We can also see that there are clear boundaries between the two classes in both figures. Therefore, we believe we can detect *Spectre* attacks using the microarchitectural features we selected.

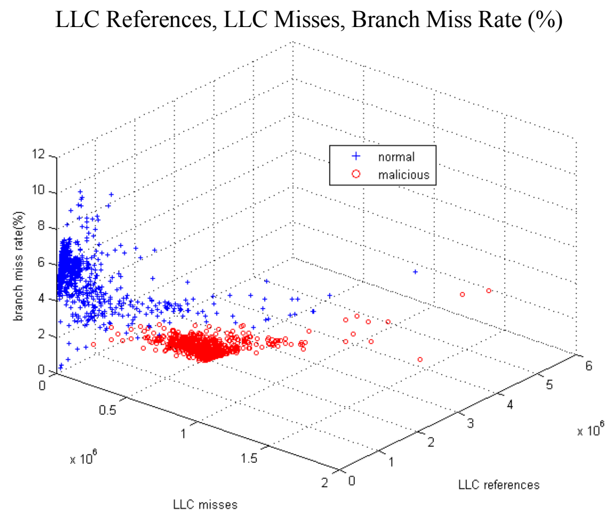

Fig. 7. Distribution of LLC references, LLC misses and branch miss rate features for *Spectre*.

B. Online Detection Performance

Depending on the distribution of the collected performance counter data, we use different machine learning algorithms of different complexity. As mentioned in section III.A, this is used to train the base classifier. We then smooth the output of the base classifier with WMA and finally build the online detector based on the sliding window approach described earlier. The performance of the detection varies with the classifiers used.

A simple model using Logistic Regression is first built with default parameters. To further enhance the detection accuracy, different parameters are tuned for the model. If the theoretical best accuracy is not reached, we then move on to more complex models using Support Vector Machine, then Multi Layer Perceptron. We use randomized search over different parameters of different models to find the best combination, where each setting is sampled over a distribution of possible parameter values. Compared with exhaustive search, it is less computationally expensive and gives results that are close to the optimal solution. The parameters for different classifiers used in our experiments are as follows:

1) *Logistic Regression (LR):*

a) Regularization strength C: Trades off misclassificaiton of training examples against simplicity of decision boundary. A smaller C gives a smoother boundary for stronger regularization,

b) Regularization parameters L1 and L2: Prevents overfitting by imposing a penalty on the coefficients.

2) *Support Vector Machine (SVM):*

a) Gamma parameter: Defines how far the influence of a single training sample reaches.

b) Regularization strength C: Works similarly to the C paramenter in LR.

c) Kernel: Includes linear, radial basis function (RBF), or polynomial kernel.

3) Multi Layer Perceptron (MLP):

a) Hidden layer sizes: Defines the number of hidden layers in the network and number of hidden neurons in each layer.

b) Activation function: Actives the neurons in the hidden layers, can be logistic, rectified linear unit function or hyperbolic tangent function.

c) Regularization parameter alpha: Avoids overfitting.

To choose the most suitable classifier, in real world applications such those found in embedded systems, we need to consider constraints of time, power consumption, memory resources, *etc.* In addition, we need to know how much the system is allowed to tolerate in terms of false positives and false negatives.

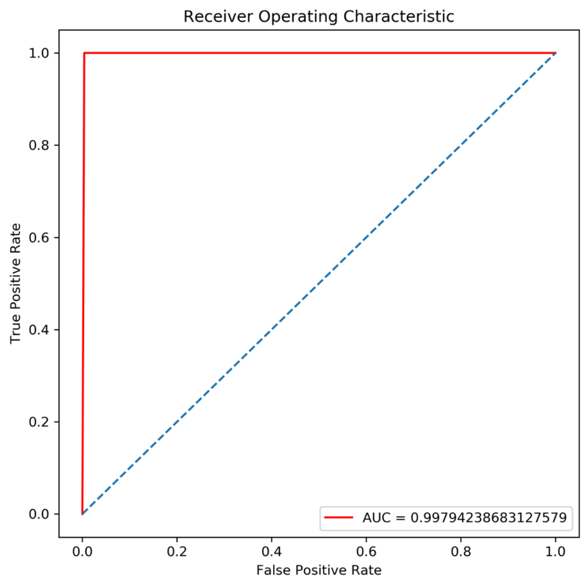

Fig. 8. ROC for online detection of *Rowhammer* using logistic regression.

To quantitatively evaluate the performance of online detection based on different classifiers, we look at the Receiver Operating Characteristic (ROC) curves which plots false positive rate as the x-axis against true positives as the y-axis as shown in Fig. 8, 9, 10. Indeed, ROC curves are typically used to show the tradeoff between false positives and true positives. If we allow a higher rate of false positives (in other words, moving towards the right of the graph), the detector should be able to catch more malicious attacks. The dotted diagonal line connecting (0,0) and (1,1) represents the performance of a classifier that randomly guesses. For a classifier performs better than random guess, its ROC will lie above the diagonal. We can see that all our trained classifiers have better performance than random guess.

Fig. 8 shows the ROC curve for online detection of *Rowhammer* attacks using simple Logistic Regression with default parameters. We can see the trade-off is minimal in this case: as we move from the top left corner of the ROC curve to the right along the curve to allow more false positives, the true positive rate does not increase much (from 99.77% to 100%). To choose the best configuration, we pick the point on the curve at the top left corner which gives the lowest sum of false negatives and false positives. For a simple model such as LR, we are able to obtain an AUC value of 0.9979 which is obviously very close to 1.

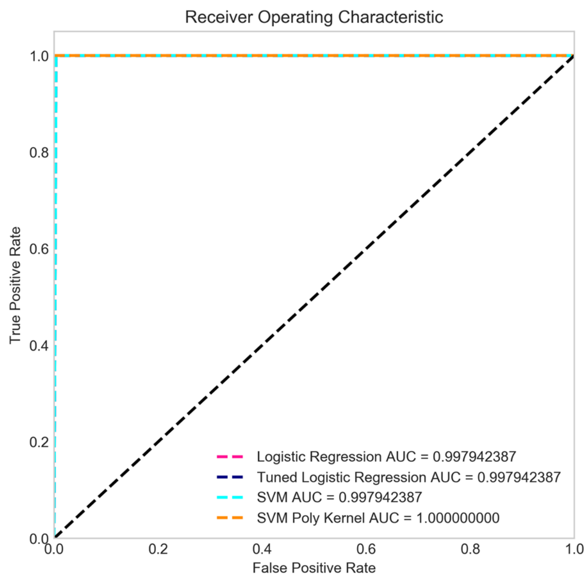

Fig. 9. ROC for online detection of *Rowhammer* using different classifiers.

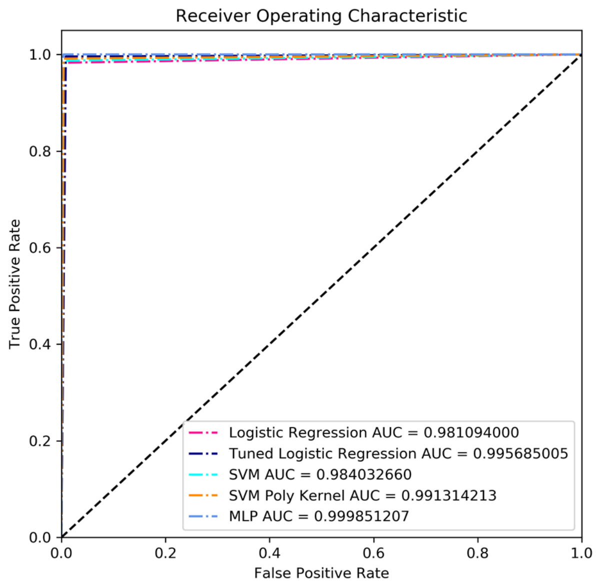

Fig. 10. ROC for online detection of *Spectre* using different classifiers.

Fig. 9 compares the ROC curves for all the trained classifiers for *Rowhammer* attacks in our experiment. We can see that they all perform very well with an AUC value above 0.99. As we were building models with increasing complexity, the SVM with polynomial kernel of degree 3 could reach the best possible AUC value of 1, which means a 0% error rate. Therefore, we conclude that increasing the complexity of models based on MLP would not lead to any further improvement.

Similarly, in Fig. 10, we plot the AUC curves for the online detectors using different classifiers for *Spectre* attacks. Compared with Rowhammer attacks, the performance counter data are spread more randomly as we discovered while training and testing the classifiers with increasing complexities. Therefore, we built more complex models using MLP with 2 hidden layers. In general, all the classifiers give fairly good results with AUC values above 0.98. Overall, MLP outperforms other classifiers with the highest AUC value.

TABLE I. PERFORMANCE OF DIFFERENT CLASSIFIERS FOR ROWHAMMER

Classifier	AUC	FP (%)	FN (%)	Training Time (sec)
LR	0.9979423868	0	0.23	0.03
Tuned LR	0.9979423868	0	0.23	0.03
SVM	0.9979423868	0	0.23	0.05
SVM with Polynomial Kernel	1	0	0	7

TABLE II. PERFORMANCE OF DIFFERENT CLASSIFIERS FOR SPECTRE

Classifier	AUC	FP (%)	FN (%)	Training Time (sec)
LR	0.9810939999	3.83	3.40	0.04
Tuned LR	0.9956850054	1.15	2.43	0.04
SVM	0.9840326600	0.77	2.43	0.06
SVM with Polynomial Kernel	0.9913142134	0.77	0.97	9.8
MLP	0.9998512071	0.77	0	95

We compare the performance of each classifier quantitatively using the AUC index and we choose the best point on the ROC which gives the minimum FP and FN shown in Table I and Table II. For *Rowhammer* attacks, using SVM with a polynomial kernel of degree 3 can already achieve perfect results (0% error rate). Our detector achieved better detection accuracy than the previous *Rowhammer* detector ANVIL [45] which exhibits 1% false positives. For *Spectre* attacks, the best case using MLP with 2 hidden layers gives 0% false negatives with only 0.77% false positives. We also observe that more complex models require longer training time since there are more parameters.

VI. CONCLUSIONS

The impact of malicious attacks targeting hardware vulnerabilities can be catastrophic and widespread as they usually can bypass traditional software-based security defenses. This paper has proposed to detect such attacks by monitoring microarchitectural features deviations. This is done by collecting related data from performance counters. We take *Rowhammer* (exploits DRAM disturbance error vulnerability) and *Spectre* (exploits speculative execution and side channel vulnerabilities) attacks to demonstrate the feasibility and effectiveness to detect such attacks using microarchitectural features. The features are collected from hardware performance counters normally available in modern processors. An online detection method is adopted to detect malicious behaviors during the attack at early stage rather than offline detection after the damage is done. The experimental results show promising detection accuracy with 0% overall error rate for *Rowhammer* attacks using SVM, and only 0.77% false positives and 0% false negatives for *Spectre* attacks using a trained multilayer Perceptron classifier. As complete mitigation to the *Spectre* is challenging, it is imperative to dynamically detect such attacks.

There are many variants of *Spectre* depending on the types of hardware design flaws and side channels being exploited. New variants are continuously discovered and researchers recently identified a new speculative store bypass vulnerability [34]. It should be noted however, that all the variants use a side channel to infer confidential information in the final attack stage. In addition, evasive attacks are difficult to perform effectively especially when evasion-resilient detector [48] is used. We thus conclude it is possible to detect malicious behaviors by monitoring changes in these hardware side channels. For *Rowhammer* attacks, there are also many new variants that circumvent recent security defenses for different architectures ranging from mobile devices to cloud servers. The research we have just presented has shown that the proposed approach is particularly effective for *Rowhammer* and *Spectre* attacks targeting hardware vulnerabilities. Future work will examine other variants of Rowhammer and Spectre attacks and different attacks which exploit other hardware design vulnerabilities in different domains such as CPU, memory, GPU, *etc* and in server environments. To further improve the detection performance, we will experiment online machining learning algorithms and implement the detector in dedicated FPGA.

REFERENCES

[1] P. Kocher, "Timing attacks in implementations of Diffie-Hellman, RSA, DSS, and other systems," Proceedings Crypto '96, Lecture Notes in Computer Science, vol. 1109, Springer-Verlag, pp. 104-113.

[2] Werner Schindler, "A Timing Attack against RSA with the Chinese Remainder Theorem," CHES 2000, pp.109-124, 2000.

[3] Werner Schindler, "Optimized Timing Attacks against Public Key Cryptosystems," Statistics and Decisions, 20:191-210, 2002.

[4] David Brumley and Dan Boneh, "Remote Timing Attacks are Practical," Proceedings of the 12th USENIX Security Symposium, pp.1-14, August 2003.

[5] C. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," Advances in Cryptology - CRYPTO '99, vol. 1666 of Lecture Notes in Computer Science, pp.388-397, Springer-Verlag, 1999.

[6] J.-S. Coron and L. Goubin, "On Boolean and arithmetic masking against differential power analysis," Cryptographic Hardware and Embedded Systems (CHES 2000), vol. 1965 of Lecture Notes in Computer Science, pp. 231-237, Springer-Verlag, 2000.

[7] J. Waddle and D. Wagner, "Towards efficient second order power analysis," Cryptographic Hardware and Embedded Systems (CHES 2004), vol. 3156 of Lecture Notes in Computer Science, pp. 1-15, Springer-Verlag, 2004.

[8] K. Gandolfi,C.Mourte, F. Olivier, "Electromagnetic Analysis: Concrete Results," CHES 2001, LNCS 2162, pp.251-261, 2001.

[9] J.J. Quisquater, D. Samyde, "Electromagnetic analysis (EMA): measures and countermeasures for smart cards," E-smart 2001, LNCS 2140, pp.200–210, 2001.

[10] M. Kuhn, "Optical Time-Domain Eavesdropping Risks of CRT Displays," Proc of the 2002 Symposium on Security and Privacy, pp.3-18,2002.

[11] A. Shamir, E.Tramer, “Acoustic cryptanalysis: on nosy people and noisy machines,” Eurocrypt 2004 rump session, 2004.

[12] D. A. Osvik, A. Shamir and E. Tromer, “Cache attacks and Countermeasures: the Case of AES,” Cryptology ePrint Archive, Report 2005/271, 2005.

[13] D. J. Bernstein, “Cache-timing Attacks on AES”, http://cr.yp.to/antiforgery/cachetiming20050414.pdf, 2005.

[14] Y. Yarom and K. Falkner, “FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack,” USENIX Security, San Diego, CA, US, Aug 2014, pp.719–732.

[15] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, “Last-level cache side-channel attacks are practical,” IEEE Symposium on Security and Privacy (S&P), May 2015, pp. 605–622.

[16] Aciiçmez, O., Gueron, S., and Seifert, J.-P., “New branch prediction vulnerabilities in OpenSSL and necessary software countermeasures,” 11th IMA International Conference on Cryptography and Coding (Dec. 2007), S. D. Galbraith, Ed., vol. 4887 of Lecture Notes in Computer Science, Springer, Heidelberg, pp. 185–203.

[17] Aciiçmez, O., Koç, Çetin Kaya., and Seifert, J.-P., “Predicting secret keys via branch prediction,” Topics in Cryptology CT-RSA 2007 (Feb. 2007), M. Abe, Ed., vol. 4377 of Lecture Notes in Computer Science, Springer, Heidelberg, pp. 225–242.

[18] Evtyushkin, D., Ponomarev, D. V., and Abughazaleh, N. B., “Jump over ASLR: attacking branch predictors to bypass ASLR,” MICRO (2016), IEEE Computer Society, pp. 1–13.

[19] Lee, S., Shih, M., Gera, P., Kim, T., Kim, H., and Peinado, M., “Inferring fine-grained control flow inside SGX enclaves with branch shadowing,” 26th USENIX Security Symposium, USENIX Security 2017, pp. 557–574.

[20] Horn, J., “Reading privileged memory with a side-channel,” https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html, 2018.

[21] Kocher, P., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y., “Spectre attacks: Exploiting speculative execution,” ArXiv e-prints, Jan. 2018.

[22] Gruss, D., Lipp, M., Schwarz, M., Fellner, R., Maurice, C., and Mangard, S., “KASLR is dead: long live KASLR,” International Symposium on Engineering Secure Software and Systems, Springer, pp. 161–176, 2017

[23] Demme, J., Maycock, M., Schmitz, J., Tang, A., Waksman, A., Sethumadhavan, S., Stolfo, S., “On the feasibility of online malware detection with performance counters,” Proceedings of the International Symposium on Computer Architecture (ISCA), 2013.

[24] Duda R, Hart P, Stork D., “Pattern classification. 2nd ed,” New York: Wiley/Interscience; 2000.

[25] J. Frank, “Machine learning and intrusion detection: Current and future directions,” in Proc. National 17th Computer Security Conference, Washington,D.C., October 1994.

[26] Cristianini N, Shawe-Taylor J., “An introduction to support vector machines and other kernel-based learning methods,” Cambridge: Cambridge University Press; 2000.

[27] Schölkopf B, Smola A., “Learning with kernels: support vector machines, regularization, optimization, and beyond,” Cambridge, MA: MIT Press; 2002.

[28] Bishop C., “Neural networks for pattern recognition,” Oxford: Oxford University Press; 1995.

[29] Hastie T, Tibshirani R, Friedman J., “The elements of statistical learning: data mining, inference, and prediction,” New York: Springer; 2001.

[30] Sharma, D.K., Sharma, H.P. & Hota, H.S., “Future Value Prediction of US Stock Market Using ARIMA and RBFN,” International Research Journal of Finance and Economics (IRJFE), 2015, 134, 136-145.

[31] Handa, R., Hota, H.S., & Tandan, S.R., “Stock Market Prediction with various technical indicators using Neural Network techniques,” International Journal for research in Applied Science and Engineering Technology (IJRASET), 2015, 3(4) , 604-608.

[32] E. Keogh, S. Chu, D. Hart, M. Pazzani, “Segmenting time series: A survey and novel approach,” Data Mining Time Series Databases, vol. 57, pp. 1-22, 2004.

[33] Vafaeipour, M., Rahbari, O., Rosen, M.A., Fazelpour, F. & Ansarirad, P., “Application of sliding window technique for prediction of wind velocity time series,” International journal of Energy and environmental engeering (springer), 5,105-111, 2014.

[34] Horn, J., “Speculative execution, variant 4: speculative store bypass”, https://bugs.chromium.org/p/project-zero/issues/detail?id=1528, May 2018.

[35] Kim, Y.R., Daly, J., Kim, C., Fallin, J., Lee, H., Lee, D., Wilkerson, C., Lai, K., Mutlu, O, “Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors,” ACM/IEEE 41st International Symposium on Computer Architecture (ISCA), Minneapolis, MN, USA, 14–18 June 2014.

[36] Seaborn M. and Dullien T., “Exploiting the DRAM rowhammer bug to gain kernel privileges,” in Black Hat Briefings, 2015.

[37] Bosman, E., Razavi, K., Bos, H., Giuffrida, C., “Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector,” IEEE Symposium on Security and Privacy (SP), 2016.

[38] Gruss, D., Maurice, C., Mangard, S., “Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript,” International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA), 2016. •

[39] Razavi, K., Gras, B., Bosman, E., Preneel, B., Giuffrida, C., Bos, H., “Flip Feng Shui: Hammering a Needle in the Software Stack,” in the Proceedings of the 25th USENIX Security Symposium, 2016.

[40] Xiao, Y., Zhang, X., Zhang, Y., Teodorescu, M.R., “OneBitFlips, OneCloudFlops: Cross-VM Row Hammer Attacks and Privilege Escalation,” in the Proceedings of the 25th USENIX Security Symposium (SEC), 2016. •

[41] Van der Veen, V., Fratantonio, Y., Lindorfer, M., Gruss, D., Maurice, C., Vigna, G., •Bos, H., Razavi, K., Giuffrida, C., “Drammer: Deterministic Rowhammer Attacks on Mobile Platforms,” ACM SIGSAC Conference on Computer and Communications Security (CCS), 2016. •

[42] Tatar, A., Krishnan, R., Athanasopoulos, E., Giuffrida, C., Bos, H., And Razavi, K., “Throwhammer: Rowhammer Attacks over the Network and Defenses,” in the Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC ’18).

[43] Jang, Y., Lee, J., Lee, S., And Kim, T., “SGX-Bomb: Locking Down the Processor via Rowhammer Attack,” in Proceedings of the 2nd Workshop on System Software for Trusted Execution (SysTEX), 2017.

[44] Kasamsetty, K., “DRAM scaling challenges and solutions in LPDDR4 context,” MemCon 2014.

[45] Aweke, Z.B., Yitbarek, S.F., Qiao, R., Das, R., Hicks, M., Oren, Y., Austin, T., “ANVIL: Software-Based Protection Against Next Generation Rowhammer Attacks,” in the Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2016.

[46] Brasser, F., Davi, L., Gens, D., Liebchen, C., Sadeghi, A.R., “Can't touch this: Software-only mitigation against rowhammer attacks targeting kernel memory,” 26th USENIX Security Symposium 2017.

[47] Qiao, R., Seaborn, M., “A new approach for rowhammer attacks,” 2016 IEEE •International Symposium on Hardware Oriented Security and Trust (HOST). pp.161–166 , May 2016. •

[48] Khaled N. Khasawneh, Nael Abu-Ghazaleh, Dmitry Ponomarev, Lei Yu, “RHMD: evasion-resilient hardware malware detectors,” Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, October 14-18, 2017.

[49] M. Ozsoy, C. Donovick, I. Gorelik, N. Abu-Ghazaleh, D. Ponomarev, “Malware-aware processors: A framework for efficient online malware detection,” Proc. IEEE 21st Int. Symp. HPCA, pp. 651-661, Feb. 2015.