Detecting Spectre Attacks Using Hardware Performance Counters

Congmiao Li, *Member, IEEE* and Jean-Luc Gaudiot, *Life Fellow, IEEE*

Abstract—*Spectre* attacks can be catastrophic and widespread because they exploit common design flaws caused by the speculative capabilities in modern processors to leak sensitive data through side channels. Completely fixing the problem would require a redesign of the architecture for transient execution or the implementation of a new design on re-configurable hardware. However, such fixes cannot be backported to old machines with fixed hardware design. Completely replacing those machines will take a long time. Moreover, existing software patches may cause significant performance overhead. This paper proposes to detect *Spectre* by monitoring deviations in microarchitectural events using hardware performance counters with promising accuracy above 90 percent under a variety of workload conditions. However, the attacker may attempt to evade detection by slowing down the attack or mimicking benign programs. This paper thus compares different evasion strategies quantitatively and demonstrates that it is possible for the attacker to avoid detection when operating the attacks at a lower speed while maintaining a reasonable attack success rate. Then, we show that, in order to resist evasion, the original detector must be enhanced by randomly switching between a set of detectors using different features and sampling periods so we can keep the detection accuracy above 80 percent.

Index Terms—Evasive malware, microarchitectural attacks, security

1 Introduction

Confidentiality is an important component of ensuring information security. It often means protecting information from access by unauthorized parties. Information can be leaked to unprivileged parties through unintended side channels. These unintentional side channels can be timing information [1], [2], [3], [4], power consumption information [5, 6, 7], electromagnetic radiation [8], [9], light emission [10] or sound [11]. By measuring and analyzing the differences in side channel information, confidential data could be discovered by adversaries. In microarchitectural side-channel attacks, malicious processes attempt to interfere with the victim through shared microarchitectural resources such as cache [12], [13], [14], [15], branch predictor [16], [17] or Branch Target Buffers [18], [19]. The interference pattern can then be measured to infer secrets.

Speculative execution reduces the processor idle time to improve performance by executing a predicted path before the actual path is confirmed. However, this may leave observable side effects that allow attackers to obtain confidential data. The branch predictor guesses which branch is more likely to be taken according to the execution history. The processor fetches the associated data and instructions. Then it starts speculatively executing them. If the branch predictor guessed correctly, the processor continues running without delay. Otherwise, the processor will roll back the execution state. Instructions only retire after the correct path is known. However, changes to other microarchitectural features such as cache contents are not reversed and it is this information that an attacker will seek to harvest.

Spectre attacks [20] exploit speculative execution to leak confidential information through unintended side channels by tricking the processor into taking carefully designed malicious code. *Spectre* attacks have different variants and work on various Intel, AMD and ARM platforms. Therefore, there is no single patch for the whole class of attacks. Furthermore, software patches experience large performance overhead. To completely solve the problem, changes to the processor design and instruction set architectures (ISAs) are required.

In this paper, we aim to develop an online detector to detect *Spectre* during the attack and improve its resilience to evasion. We first propose to use low-level microarchitectural features which can be extracted from HPCs to detect the original *Spectre Variant 1* [20] during the attack with offline supervised machine learning algorithms. We then study how the original *Spectre* could be even more maliciously updated to operate effectively without being detected by HPC-based classifiers in a quantitative way. Our research demonstrates that it is possible to profile the attack in a given system setup so as to construct such an evasive version of *Spectre* by changing the microarchitectural characteristics of a *Spectre* attack. We also quantitatively compare different strategies to determine the best way the attackers could use to evade detection while maintaining a reasonable attack success rate and attack speed. Finally, we improve our proposed *Spectre* detector by randomly switching between different detectors to simulate counter evasion.

2 Related Work

Traditional antivirus (AV) software scans suspicious instructions in binaries or traces in system logs files to statically

• *The authors are with the Department of Electrical Engineering and Computer Science, University of California Irvine, Irvine, CA 92697 USA. E-mail: {congmial, gaudiot}@uci.edu.*

Manuscript received 30 May 2020; revised 9 Feb. 2021; accepted 16 May 2021. Date of publication 20 May 2021; date of current version 10 May 2022. (Corresponding author: Congmiao Li.) Recommended for acceptance by A. Karanth. Digital Object Identifier no. 10.1109/TC.2021.3082471

detect malicious attacks. Unfortunately, *Spectre* usually does not leave traces in system log files. Thus, it is difficult to detect *Spectre* with a static analysis. Recent research [21], [22], [23] has shown that malware can be detected by using dynamic microarchitectural execution patterns gleaned from existing Hardware Performance Counters (HPC) in modern processors. Demme *et al.* [21] adopted an offline analysis based on various supervised machine learning algorithms using a complete trace of the program behavior after execution to detect malware in the Android OS. Tang *et al.* [22] deployed unsupervised machine learning techniques to detect return-oriented programming and buffer overflow attacks. Khasawneh *et al.* [23] proposed to use ensemble learning to detect malware at runtime. Researchers [24] also used HPCs to detect side channel attacks. Mambretti *et al.* [25] developed a performance-counter-based tool to aid in designing speculative execution attacks and mitigation.

3 Background

Spectre exploits critical design flaws in some modern processor design to allow attackers to steal sensitive information from users' devices through microarchitectural side channels. In this section, we describe in detail the attack and review existing mitigation techniques. In addition, we review possible detection approaches and some known evasive malware.

3.1 Description of *Spectre*

As reported in [20], the original *Spectre* has two variants: bounds check bypass and branch target injection. The first variant exploits conditional branch mispredictions. The second variant targets the indirect jump target prediction. Subsequently, Intel® identified eight new similar variants, referred to as *Spectre Next Generation (Spectre-NG)* [26]. One of the new variants allows the execution of malicious code from a virtual machine (VM) to attack the host machine or other VMs on the same server. Another variant exploits return stack buffers (RSBs) to trigger mispredictions of return addresses and leaks sensitive information across processes [27]. All the variants follow the same principles in that they rely on the changes in the state of the cache caused by the speculative execution. The behavior of a processor at runtime may be different depending on whether one looks at it from the architectural or the microarchitectural point of view. For example, architecturally, a program that loads a value from a particular address in memory will wait until the address is known before the load. However, at the micro-architecture level, the processor may speculatively load the value from the predicted address in memory. If the prediction was wrong, the processor will load again from the correct address which preserves the architectural behavior. However, the microarchitectural behavior such as the content of the cache can be changed, thereby allowing the attacker to infer the values stored in memory. In general, any observable changes from the speculatively executed code may cause the leaking of confidential information.

All variants of *Spectre* exploit processor microarchitectural vulnerabilities in the similar way it was discussed above. *Spectre Variant 1 (Spectre-V1)* is the most difficult to detect and defend against. In this paper, we take *Spectre-V1* as our benchmark and study possible detection techniques in details. Different variants exploit different kinds of speculative

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

Fig. 1. Conditional branch example.

executions following different control or data misprediction, but in the final phase, secret information is usually leaked through a cache side channel. Therefore, we can monitor the cache behavior to detect the attacks. *Spectre-V1* is activated as the result of a branch misprediction. For example, the victim function in 1 receives an integer `x` from an untrusted source. To ensure security, the function does a bound check on `x` to prevent the process from an unauthorized memory read outside `array1`. However, speculative execution can lead to out-of-bounds memory reads: assume 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.

3.2 Mitigating *Spectre*

Mitigating the effects of *Spectre* is difficult because there are many variations of possible attacks. For example, to prevent *Spectre-V1*, 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. There is also a lack of efficient, architecture-independent ways of addressing *Spectre-V1* in user space code. OS and firmware updates for other variants of *Spectre* published have been reported to slow down the computer systems significantly.

3.3 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 tricking the processor to take the wrong execution path, and then leak the confidential information through an observable microarchitectural side channel.

For the conditional branch example in Fig. 1, the attacker calls the victim function multiple times so as to cause the condition predicted 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 [20] 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.

3.4 Evasive Malware

Attackers may attempt to evade detection by reverse-engineering the detector and mimicking the behavior of normal programs. Previous research [28], [29], [30] suggested generating *evasive malware* (also known as *mimicry attack*) by insertion of no-op instructions, code obfuscation, or calling of benign functions in between malignant payloads so as to bypass software malware detectors. Researchers [31] also discussed evading detection by hardware malware detectors after adding instructions into the control flow graph of the malware without changing the execution state of the program. However, for *Spectre* attacks to be successful, attackers must also ensure that cache and branch predictor states are not affected. Therefore, our research also studies the feasibility of developing evasive *Spectre* with reasonable attack success rate and the countermeasures to detect such attacks.

3.5 Hardware Performance Counters

Modern processors provide a variety of Hardware Performance Counters (HPCs) for monitoring and measuring events during program execution such as instruction retired, cache hits or misses and branch misprediction *etc.*. Profiling tools such as *perf* can be used to access HPCs measurements through a simple commandline interface. *Perf* can be read using two modes, namely counting or sampling. In the counting mode, the occurrences of events are simply aggregated and presented on standard output at the end of an application run. In the sampling mode, it samples various metrics based on the occurrence of a certain number of events. We use the sampling mode in our experiments.

4 Detection of the Original Spectre Attack

In this section, we propose and demonstrate an approach which will permit us to monitor microarchitectural events from existing CPU performance counters and to detect the original *Spectre-V1* while under attack, with high detection accuracy and minimal performance overhead.

4.1 Proposed Online Detection Approach

Our proposed detection approach periodically collected microarchitectural features from performance counters. For the *Spectre-V1* attack discussed in Section 3.1 (exploiting conditional branches), we chose to monitor four events related to the cache miss rate and the branch misprediction rate: namely cache references, cache misses, branch instructions retired, and branch mispredictions. The data was collected from a "clean" environment where the computer runs typical desktop applications such as web browser, video player, and text editors, as well as from an environment under *Spectre* attack while running the same desktop applications. The data was then labeled to train the machine learning classifier to classify the input data at each time interval. At runtime, the output time series from the trained classifier was fed into the online detection mechanism to decide if the system was under attack. This section describes in detail the machine learning algorithms we used to train the classifier and our proposed online detection approach.

4.1.1 *Machine Learning Classifiers*

Machine learning (ML) can be used to train classifiers that determine the class to which a given data set belongs. We used supervised learning [32] 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 collected data in ten independent runs and used the same number (1,200) of samples from both classes to avoid any bias. Then, we randomly divided the collected data into training (80 percent) and test (20 percent) data, and separated the data into training (80 percent) and validation (20 percent) data.

We chose three different machine learning algorithms including Logistic Regression (LR) [33], Support Vector Machine (SVM) [34], [35], and Artificial Neural Networks (ANN) (also known as Multilayer Perceptron (MLP)) [36], [37], [38], to build the classifiers with increasing model complexity (many other machine learning algorithms exist, but for demonstration purposes, we chose three commonly used algorithms with different complexities according to the data size).

4.1.2 *Online Attack Detection*

We used a sliding window-based online classification methods similar to that proposed in [39] to detect malicious behavior at runtime. To this end, we collected microarchitectural features at a 100 ms sample period. The multidimensional data were then fed to a machine learning classifier to make decisions as to whether malicious code was being executed or not. The problem of detecting malicious attacks in real time was to make decisions according to the binary time series generated by the base classifier.

To smooth the fluctuated time series data, a Weighted Moving Average (WMA) was used to filter out noise for better decision making by assigning a weight factor to each element in the time series. More recent data were given higher weights. Then we segmented the data using a sliding window [40], [41] 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. Attackers could use evasion techniques such as slowing down the attack to keep the average below the threshold. We discuss the possible evasion strategies in detail in Section 5. Fig. 2 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

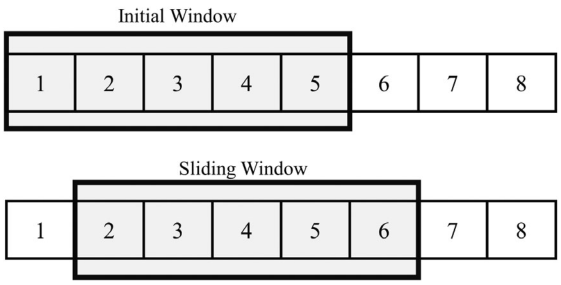


Fig. 2. Sliding window example.

within this window were used to determine the final classification result for the current time. The detection ran continuously for the next period where the window slid to the right by one segment to cover data from segments 2 to 6, before it moved on to the next window.

In this experiment, we chose the window size to be 10 and sampling period of performance counters data collection to be 100 ms, which means our detector makes a decision every second on whether the system is under attack. In Section 5 we study multiple ways to evade detection such as slowing down the attack. 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 a satisfactory detection accuracy without a major slowdown of the system. In Section 6, we used randomized sampling periods to improve the resilience of our detector.

4.1.3 *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 used metrics such as False Positives (FP), the percentage of misclassified malicious instances, and False Negatives (FN), the percentage of misclassified normal instances. A detection approach with good performance is expected to minimize both.

To visualize the tradeoff between the percentage of correctly identified malicious instances and the percentage of normal instances misclassified, we used Receiver Operating Characteristic (ROC) graphs plotting True Positives ($TP = 100\% - FN$) against FP. To compare the performance of different models, we computed and compared 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.

4.2 Experimental Setup

To examine whether data from performance counters can be used to effectively detect *Spectre*, we collected events which are expected to be affected by the attack (this includes cache references, cache misses, branch instructions retired and branch mispredictions from running the attack on top of normal programs and running typical benign applications alone respectively). The data were then pre-processed before being used to train different machine learning classifiers to detect malicious behavior. In this section, we described the details of the data collection mechanism and the system settings under attack and in normal conditions. We demonstrated the proposed online detector under a standalone personal computer environment, but our work could easily be extended to server environments in the future.

Overfitting is a common problem for machine learning classifiers. To allay this problem, we first sought to include more data by collecting data from performance counters periodically in 10 independent runs and used the same number (1,200) of samples from malicious and normal classes. We used the same number of samples from both classes to ensure the training data are not biased by different classes. Then, we randomly split the collected data into training (80 percent) and test (20 percent) data, then separated the training data into training (80 percent) and validation (20 percent) data for cross-validation. We also included regularization parameters in the machine learning models during training. In addition, we compared the detection accuracy under normal and high system load conditions to study the effectiveness of the malware classifiers.

4.2.1 *Data Collection Mechanism*

We ran the attack on a typical personal laptop with Debian Linux 4.8.5 OS on an Intel® Core™ i3-3217U 1.8 GHz processor with 3 MB cache and 4 GB 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 *Spectre* attack in Section 3.1, we chose 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 used the standard profiling infrastructure of Linux, *perf* tools, to obtain performance counter data. We ran *perf* in sampling mode and record the measurements every 100 ms so as not to excessively degrade the performance. We collected system-wide data rather than for individual processes for the following reasons. First, monitoring each process separately may introduce significant performance overhead, it may not be a scalable solution for systems running many programs. Second, classification using system-wide data is more difficult since the training data labelled “malicious” includes noise from legitimate programs. Previous researchers [21] also adopted a similar method. Using system-wide data can better demonstrate the feasibility of our proposed detector than using data from individual processes since the classification results of individual processes are expected to be better with less false positives.

4.2.2 *Test Environment Setup*

In a clean environment, we sought to create realistic scenarios by randomly browsing through popular websites (according to Wikipedia in FireFox) 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 was under a malicious attack, we launched the *Spectre* proof of concept attack on top of normal running applications. To experiment with systems under high load conditions, besides running the above-mentioned applications, we stressed the system with (1) a memory intensive load running *memcpy* (copy 2 MB of data from a shared region to a buffer using *memcpy* and then moved the data in the buffer with *memmove*), (2) a CPU intensive load running floating-point operations *cfloat* in a loop and (3) both memory and CPU intensive workloads running in parallel. We wiped and restored all non-volatile storage and also reset the CPU memory and cache by reading the same large file (4 GB) after each run to independency of the measurements across a variety of clean and exploit runs.

4.3 Detection Results

In this section, we first analyzed the collected raw data to determine whether it is feasible to differentiate measurements in clean environments from those under attack by visualizing the distribution of data. We then used different machine learning algorithms to train the classifier and built the online attack detector using the sliding window approach discussed in Section 4.1.2.

4.3.1 Data Distribution Analysis

Our data collection mechanism produced 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 calculated the branch miss rate (1) and the LLC miss rate (2) for each interval as

$$branch\ miss\ rate = branch\ mispredictions/branches \tag{1}$$

$$LLC\ miss\ rate = LLC\ misses/LLC\ references. \tag{2}$$

We used *boxplot* to visualize the range and variance of the measured data for each individual microarchitectural feature. Fig. 3 gives a direct indication of the feasibility to detect malicious attacks using one particular feature. We observed an increased number of branches and branch mispredictions during the attacks. In contrast, the branch miss rate was decreased. This was because the attacker attempted 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 were both increased with the miss rate concentrated on the higher percentage region due to cache side channel attacks. The experimental results validated the hypothesis we proposed in Section 3.3.

We also analyzed the feasibility of distinguishing the collected performance counter data using more than one feature by plotting the sample points in 2D graph with each dimension corresponding to one feature. Fig. 4 shows the distribution of normal and malicious sample points for *Spectre* attacks using branch miss rate and LLC miss rate parameters. We could observe the data points of two different classes distributed in two different regions and the boundaries between the two are obvious. This demonstrated that

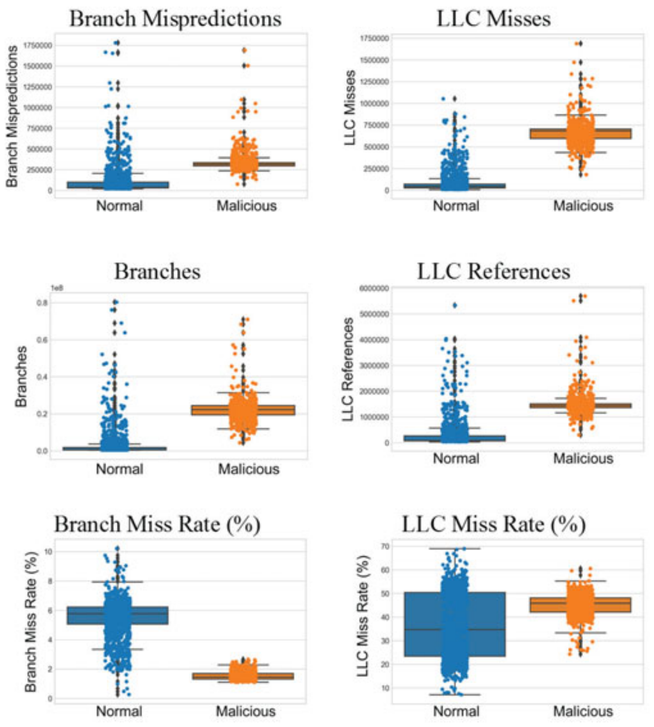


Fig. 3. Distribution of microarchitectural features from performance counters for *Spectre*.

we could detect *Spectre* attacks using the selected microarchitectural features.

4.3.2 Online Detection Performance

We used the three different machine learning algorithms mentioned in Section 4.1.1 to train the base classifier, then smoothed the output time series with WMA, and finally built the online detector based on the sliding window approach described in Section 4.1.2.

To enhance detection accuracy, different parameters were selected for each ML model. We used a randomized search over different parameters to find the best combination, where each setting was sampled over a distribution of possible parameter values. Compared with an exhaustive search, it is less computationally expensive and gives results that are close to the optimal solution.

To choose the most suitable classifier, in real world applications such as those found in embedded systems, we need to consider constraints of time, energy 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.

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. 5. 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

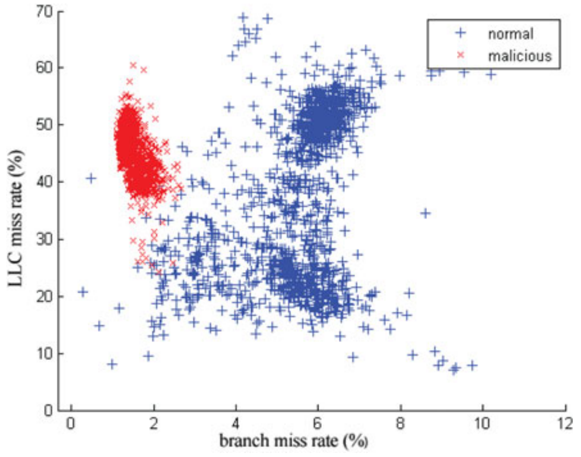


Fig. 4. Distribution of branch miss rate and LLC miss rate features for *Spectre*.

classifier to perform better than a random guess, its ROC will lie above the diagonal. Obviously, all our trained classifiers had significantly better performance than a random guess. To choose the best configuration, we picked the point on the curve at the top left corner which gives the lowest combined number of false negatives and false positives. We can see that they all performed quite well with an AUC above 0.99. Overall, MLP outperformed other classifiers with the highest AUC value for the best case scenario. This was as expected because MLP is more flexible than the other methods used.

False positives are common in antivirus software. When deploying the detector in real systems, we propose several ways to handle such cases. First, the detector will send a warning message to the user when malicious activity is detected. The user will decide if it is a false positive or choose to stop malicious applications. Second, we will allow the user to send a feedback to the antivirus developer to improve the detector. Online learning algorithms can also be implemented in the detector to received feedback directly from the user and automatically improve the detection accuracy. Finally, we can

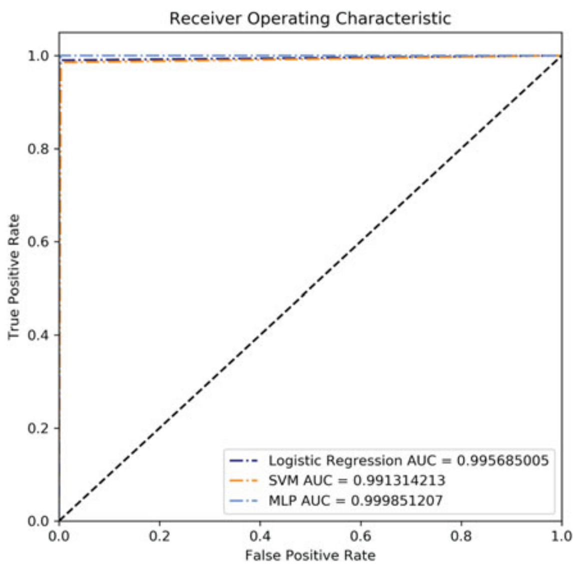


Fig. 5. ROC for online detection of *Spectre* using different classifiers.

TABLE 1
Performance of Different Classifiers for *Spectre*

Classifier	AUC	FP(%)	FN(%)	Training Time (sec)
LR	0.9956850054	1.15	2.43	0.04
SVM	0.9913142134	0.77	0.97	9.8
MLP	0.9998512071	0.77	0	95

allow the user to define a set of benign applications to be excluded from the monitoring.

We compared the performance of each classifier quantitatively using the AUC index and chose the best point on the ROC which gives the minimum FP and FN as shown in Table 1. The best case using MLP gives 0 percent false negatives with only 0.77 percent false positives. We also observed that more complex models required longer training time since there were more parameters.

Table 2 compares the detection accuracy under different system load conditions. The classifiers were trained using data collected under normal load condition. By running different kinds of additional workloads on top of the normal workload, we studied how the detection accuracy of the original classifier varies. To better understand the experimental results. We compare the classification features including branch mispredictions and cache misses for system running memory intensive load and CPU intensive load. We found there was no significant difference for branch misprediction under different workloads. However, the difference in LLC misses was very obvious as shown in Fig. 6. the number of LLC misses reduces when CPU intensive workloads were running on top of attack because CPU intensive workloads slowed down the attack without introducing more LLC misses. Whereas memory intensive workloads introduced more LLC misses and made it easier to classify malicious attack from normal condition. Therefore, the detection accuracy drops when we stress the system with CPU intensive workloads for all kinds of machine learning classifiers. However, the detection accuracy increases when we run memory intensive workloads. When the system is stressed by both kinds of workloads, the detection accuracy drops less than when running CPU intensive workloads alone. In general, our detector is able to maintain a detection accuracy above 90 percent for all the scenarios we tested.

5 Evasive Spectre

In this section, we imagine quantitatively how the original *Spectre* [20] could be even more maliciously updated to operate effectively without being detected by our previously proposed detector in Section 4. As discussed in Section 3.4, developing evasive microarchitectural side channel attacks such as *Spectre* has additional requirements compared with traditional evasive malware: the attacker must ensure that the relevant microarchitectural status remains unchanged during the attack in order to perform "successfully" (success obviously being from the point of view of the attacker). We first study the feasibility of constructing an "evasive *Spectre*" that is able to bypass HPC-based detectors while maintaining a reasonable attack success rate, and also the trade-off between

TABLE 2
Detection Accuracy for System Under Different Load Conditions(%)

Classifier	Normal	Memory Intensive	CPU Intensive	Memory & CPU Intensive
LR	96.42	97.32	90.77	94.79
SVM	98.26	99	91.63	96.63
MLP	99.23	99.57	94.42	96.97

attack success rate and attack evasiveness. Then we compare different evasion strategies to determine how an attacker could best evade detection while maintaining a reasonable success rate and speed. To achieve reasonable attack success rates, the attacker has to insert instructions or put the attack to sleep at a coarser granularity than a basic block in the control flow graph. Therefore, we define atomic tasks (detailed definition in Section 5.2.1) and reshape the microarchitectural features in the granularity of the atomic task level. If an atomic task is interrupted, the chances of success for an attack would be greatly reduced.

5.1 Threat Model

We assume that the victim's machine is running an HPC-based malware detector such as proposed in [42] to defend from *Spectre* attacks. The detector monitors four microarchitectural features including Last-Level Cache references (LLC references), Last-Level Cache misses (LLC misses), branch instructions retired (branches), and branch mispredict retired (branch mispredictions) at a fixed sampling rate on a separate core. In future research, the condition will be relaxed for mixed sampling rate and the detector can be implemented in dedicated hardware to reduce performance overhead. We assume the attacker's goal is to reveal some confidential memory content on the victim machine without being detected as malware. To achieve this goal, we further assume the attacker can observe the behavior of the malware classifier from a machine with a similar HPC-based detector as the victim machine. The attacker can evade detection by changing the microarchitectural characteristics of the updated *Spectre* so as to behave like a benign program.

We assume the attacker knows the features being monitored by the malware classifier. This is reasonable because the attacker knows that the original *Spectre* [20] would cause increased cache misses and reduced branch mispredictions. However, the attacker does not know the sampling period of the detector. Yet this can be reverse engineered (see [31]).

As a *Spectre* attack runs in a loop, we assume the attacker can slow down the attack by calling the victim function/API at specific intervals. In addition, we assume the attacker can manipulate the performance counters by inserting instructions that reduce LLC cache misses and increase branch mispredictions by exploiting other vulnerabilities such as just-in-time code reuse attacks. This gives the attacker more privileges and could help us test the detector's resilience to evasion in extreme conditions.

Previous work [31] shows that the accuracy of HPC-based detectors decreases significantly as the number of instructions inserted in the original attack increases and assumes the

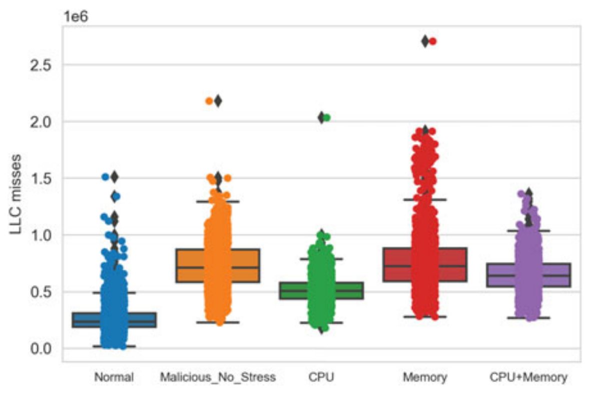


Fig. 6. Distribution of LLC misses for *Spectre* under different workloads.

attacker is interested in maintaining a reasonable performance of the attacks. However, this did not consider how the inserted instructions may affect the success rate of the detector in inferring the correct content. Since *Spectre* is time sensitive, the inserted instructions may change or provide opportunities for other running programs to change the cache status and cause the attack to read the wrong content. Therefore, besides maintaining performance (side channel bandwidth), we further assume the attacker aims at maintaining a reasonable success rate. Without these assumptions, it is impossible to detect an attack when the attack is running indefinitely slow and reading the wrong content.

5.2 Development of an "Evasive *Spectre*"

To avoid detection by HPC-based detectors, the attacker would seek to shape its microarchitectural trace to mimic that of benign programs. In so doing, the attacker would have to sacrifice the efficiency of the attack and perform it more slowly (reduced side channel bandwidth). However, slowing down *Spectre* could result in a failure of the attack. In this section, we profile the original *Spectre-V1* to study its feasibility to perform an attack without detection and discuss strategies to develop an "Evasive *Spectre*." We use the same experiment setup as discussed in Section 4.2 under normal workload conditions and launch "Evasive *Spectre*" attacks using the strategies proposed in Section 5.2.2 on top of normal applications.

5.2.1 *Feasibility Analysis of "Evasive* Spectre*"*

In order for the attack to be successful, the attacker must complete malicious tasks faster than the detection frequency. Thus, the microarchitectural trace of the attack should be reshaped so the attack can make progress faster than each detection interval. We thus define an *atomic task* as a sequence of instructions that should not be interrupted during execution if progress is to be made towards the completion of a malicious task to achieve success. We identified three *atomic tasks* in the proof of concept *Spectre-V1* [20]: *(1) Flushing cache lines, (2) Mistraining branch predictor, (3) Attempting to infer the secret byte that is loaded into the cache.* We compared the attack success rate of interrupting the attack during atomic tasks and between atomic tasks by inserting the same instructions. Table 3 shows that the attack success rate drops significantly when defined atomic tasks are interrupted. Therefore, we chose to reshape the

TABLE 3
Attack Success Rate After Interruption at Different Levels

	Task 1	Task 2	Task 3
Interrupt During Atomic Task	65%	58%	60%
Interrupt After Atomic Task	89%	90%	92%

microarchitectural features for evasion at the atomic task level rather than at a finer granularity in the control flow graph. The three *atomic tasks* respectively took 10μs, 13μs, 38μs to complete on average. For each byte, the three tasks were performed multiple times to get the best results. The original attack read secret bytes at an approximate bandwidth of 2 KB/second on average.

As discussed in Section 5.1, we assumed the attacker knew the features being monitored but did not know the classification period. we used the method proposed in [31] to collect multiple pairs of testing and training data sets of the same features using different collection periods and trained a reverse-engineered detector for each data set. The victim's collection period (100 ms) was the same as the collection period of the reverse-engineered detector with the highest accuracy.

Since the sampling period of the victim detector is much larger than the time taken to perform each *atomic task*, the attacker can transform the microarchitectural profile of *Spectre* by inserting instructions or "sleeping" at a finer granularity than the sampling rate of the detector. To analyze the feasibility of evading detection, we executed *atomic tasks* using 20 percent of each sampling period and put the attack to sleep for the remainder. Fig. 7 shows the distribution of (1) benign, (2) malicious and (3) evasive sample points using cache miss rate and branch miss rate features. It shows a clear boundary between normal and malicious sample points while the evasive sample points shift the original malicious program to overlap with normal ones (they cannot fully overlap because unlike a normal program, the evasive attack still needs to perform malicious tasks). Modified *Spectre* performs with an 89 percent success rate.

5.2.2 Strategies to Construct "*Evasive* Spectre"

As discussed in Section 3.3, the original *Spectre* increases LLC misses and reduces branch mispredictions. To evade detection, the attack could be slowed by putting it to sleep or inserting instructions that reduce the number of LLC misses (reading the same memory bytes) and increase the number of branch mispredictions (adding unpredictable branches). Assuming the attack runs in a loop, at each cycle, the attacker needs to complete a series of atomic attacks to retrieve one secret byte from the victim. We thus considered the following four strategies:

1) Put the attack to sleep in between *atomic tasks*.
2) Put the attack to sleep after all tasks have completed.
3) Insert instructions in between *atomic tasks*.
4) Insert instructions after all tasks have completed.

The first two strategies slow down the attack while strategies 3 & 4 directly manipulate the performance counters.

Varying sleep time or looping the instructions that reshape the microarchitectural profile different times will accordingly change the attack bandwidth. We studied the effectiveness of different strategies by gradually reducing the attack bandwidth and analyzing the results in the following section.

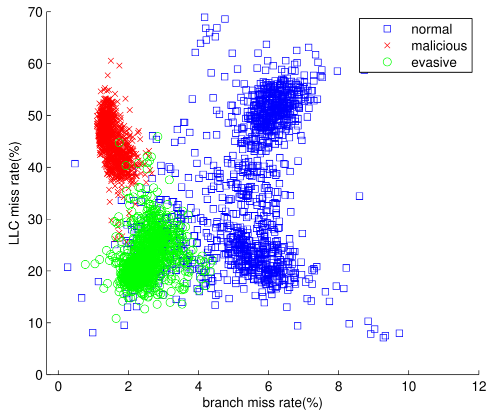


Fig. 7. Branch miss rate *versus* LLC miss rate for "evasive *Spectre*".

5.3 Experimental Results

We now evaluate different evasion strategies proposed in Section 5.2.2 by varying the bandwidth reduction from 1X to 7X and comparing the detection accuracy and the attack success rate. We define the success rate as the percentage of correct bytes inferred by the attacker over the total number of bytes inferred. Putting the attack to sleep or inserting instructions to reshape the microarchitectural profile of the attack reduces the rate of confidential content read. The longer the attack takes to reshape the profile, the higher the bandwidth reduction it will incur and the lower the success rate, due to higher TLB and cache pollution. Therefore, an effective evasion strategy should result in a low detection accuracy and maintain a reasonable attack success rate and bandwidth.

For each experimental setup with different evasion strategies and bandwidth reduction, we recorded the attack success rate and detection accuracy using the existing victim detector with different Machine Learning classifiers. We collected data in 10 independent runs for each setup and calculated the average values.

Figs. 9 to 10 show detection accuracy *versus* bandwidth reduction using different ML classifiers for each of the four strategies. In all cases, the detection accuracy drops with the

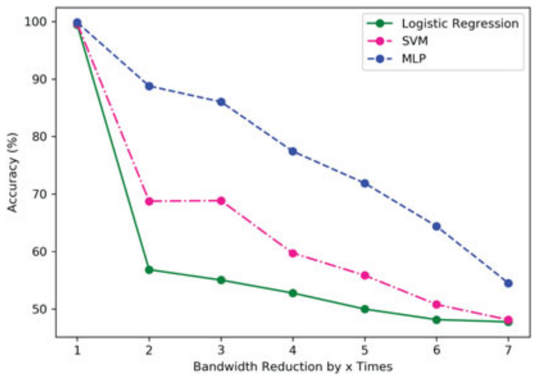


Fig. 8. Detection accuracy - strategy 1 (sleep between atomic tasks).

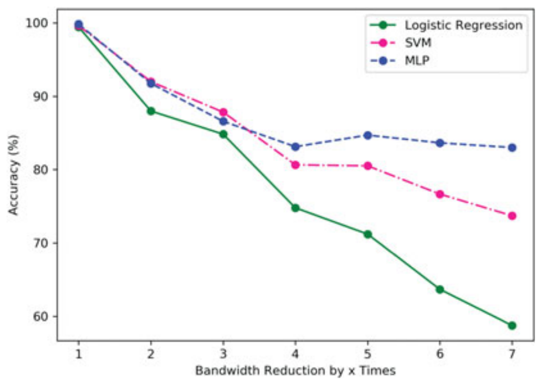


Fig. 9. Detection accuracy - strategy 2 (sleep after all tasks).

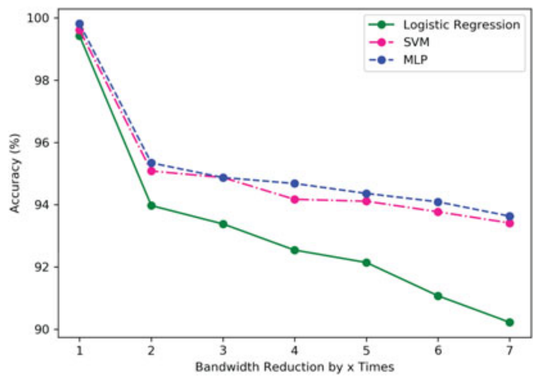


Fig. 11. Detection accuracy - strategy 4 (insert instructions after all tasks).

bandwidth because the attack becomes more evasive and closer to benign programs as it runs slower. In addition, the MLP classifier retains a better detection accuracy as the bandwidth drops. Therefore, MLP has a higher resiliency to evasive attacks. In contrast, it is easier to avoid detection by a simple LR classifier.

Fig. 12 shows the detection accuracy of different evasion strategies as bandwidth decreases using MLP. Strategy 1 causes the detection accuracy to drop fastest as bandwidth drops. The detection accuracy diminished to around 50 percent (random guess) when the bandwidth reduction was 7X. Therefore, strategy 1 produces the most evasive attack. On the other hand, strategy 4 performs the worst in this regard. Strategy 2 and 3 perform similarly in terms of evading detection. Note that shaping the microarchitectural profile in between atomic tasks yields a more evasive attack than shaping it after all the tasks are done no matter what method (sleep or insert instructions) is used.

Fig. 13 shows the attack success rate using different evasion strategies. As the attack bandwidth decreases, so does the success rate. Therefore, an attack is more likely to fail when it is running more slowly due to possible TLB and cache pollution by other processes. Similarly, inserting instructions or sleeping between atomic tasks has a higher chance to fail than inserting or sleeping after all tasks are done. The success rate dropped below 50 percent with strategy 1 at 7X bandwidth reduction.

To better evade detection (or to reduce detection accuracy), the attack must run at a lower rate and suffer a corresponding decrease in success rate. Considering the trade-off between detection accuracy and attack success rate, strategy 1 produces the most evasive attacks but the lowest success rate. Conversely, strategy 4 has the best success rate but is the least evasive. Strategy 2 yields slightly more evasive attacks than strategy 3 as bandwidth reduces further; strategy 2 has a better attack success rate than strategy 3. Therefore, strategy 2 is on the overall most effective. At a 7X bandwidth reduction, the LR classifier can only perform at 58.77 percent accuracy, i.e., no better than a random guess. Meanwhile, the attack success rate remains at 85 percent. Therefore, with strategy 2, the attacker can evade detection by an LR classifier without sacrificing much in terms of success rate and bandwidth. To evade the scrutiny of a more complex classifier such as MLP, the attacker can further reduce the bandwidth for a lower detection accuracy. At 10X bandwidth reduction, the MLP classifier performs at 70 percent accuracy and the attack success rate remains at 85 percent.

6 Evasion Resilient Spectre Detector

In this section, we show how to improve the *Spectre* detector to be evasion-resilient by randomly switching between multiples detectors with different settings. As discussed in Section 5.1, the attacker must rely on a reverse-engineering approach to obtain the detailed settings of the victim detector. Randomization introduces errors to reverse-engineering and makes the detector resistant to evasion. Researchers [31], [33] have proven theoretically that randomizing among a set of low-complexity, low-accuracy classifiers makes it inherently

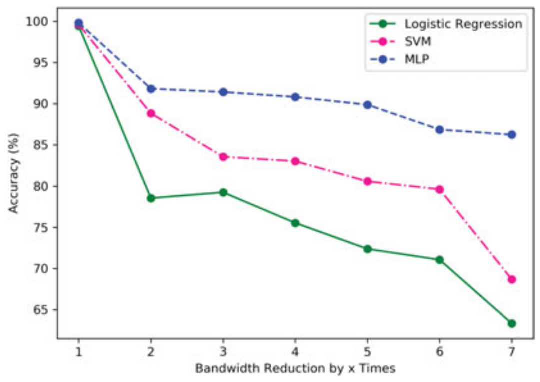


Fig. 10. Detection accuracy - strategy 3 (insert instructions between atomic tasks).

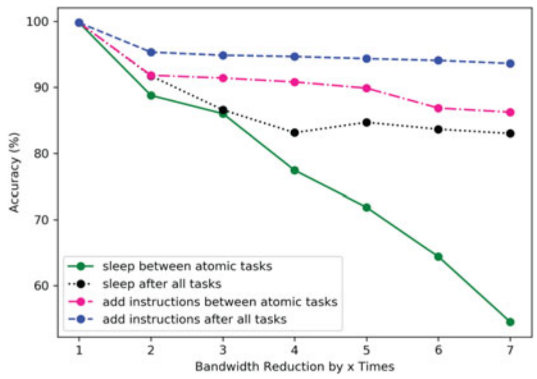


Fig. 12. Detection accuracy using Multi-Layer Perceptron.

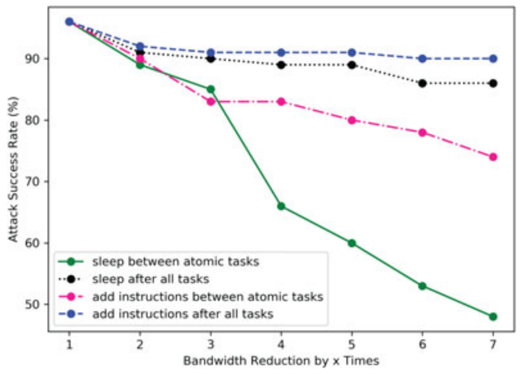


Fig. 13. Attack success rate using the proposed evasion strategies.

more difficult to reverse-engineer than using a single deterministic classifier with high complexity and accuracy.

To develop an evasive-resilient *Spectre* detector, we assume the attacker uses the most effective strategy (i.e., strategy 2: sleep after all tasks are done) as mentioned in Section 5.2.2 to evade detection. We also assume that the victim detector uses MLP as other simpler models (LR and SVM) can be easily evaded. We build detectors with different settings in terms of features used for detection and sampling periods and randomly switch between them with equal probability.

The accuracy of the reverse engineering technique is defined as the percentage of equivalent decisions made by the victim detector and reverse-engineered detector. It measures how well the attacker can mimic the victim detector. The highest accuracy occurs when the settings for the reversed-engineered detector are the same as the victim detector. Thus, the attacker can infer the victim's settings by experimenting different settings for the detector and choose the one with the highest reverse engineering accuracy.

We study how randomizing the features used for detection and sampling periods affects the accuracy of the reverse engineering and the results are presented in Figs. 14 and 15. The features used for detection include cache related features (cache references and cache misses), branch related features (branch instructions retired and branch mispredictions) or a combination of them. The sampling period used for detection is switched between 50 ms and 100 ms. The victim detector switches between a collection of randomized detectors. We built the collection of detectors with increasing diversity using different features and sampling periods. The baseline detector consists of only one detector

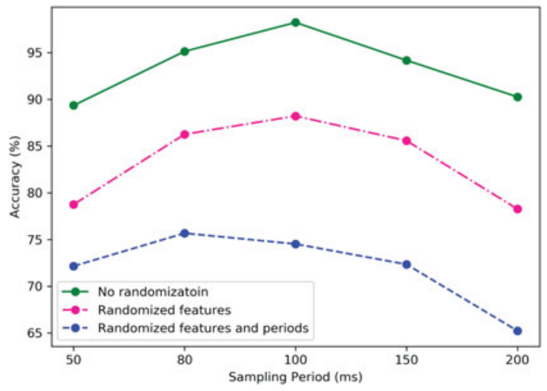


Fig. 15. Reverse-engineering the sampling period.

which uses a combination of cache and branch related features with a fixed sampling period of 100 ms. Then we built 3 detectors with randomized features to be either cache related, branch related or the combined features and switch between them in am unpredictable order. Finally, we build 6 detectors by randomizing both features and sampling periods. In general, when reverse-engineering the features used for detection (Fig. 14) and sampling period (Fig. 15), the accuracy drops as the detector diversity increases. For detectors with no randomization or only randomized features, it was possible to reverse engineer the correct settings in our experiment. However, for increased diversity with randomized features and periods, the reverse-engineering failed to infer the accurate settings of the victim.

Based on the reverse-engineered detector, the attacker could use the strategies discussed in Section 5.2.2 to evade detection. However, due to the errors introduced by randomly switching between different detectors to the reverse-engineering, it became difficult to avoid detection. Fig. 16 shows the detection accuracy for detectors of different randomization settings with increasing side channel bandwidth reduction (or attack speed reduction) up to 30X. The results show that the detection accuracy decreased as the bandwidth increased for all detectors. The baseline detector had a slightly higher detection accuracy than the other two strategies with a randomized detector for the original *Spectre* (no bandwidth reduction). However, the accuracy decreased dramatically as the bandwidth reduction increased. This means that the baseline detector could not perform well when the attack was get-

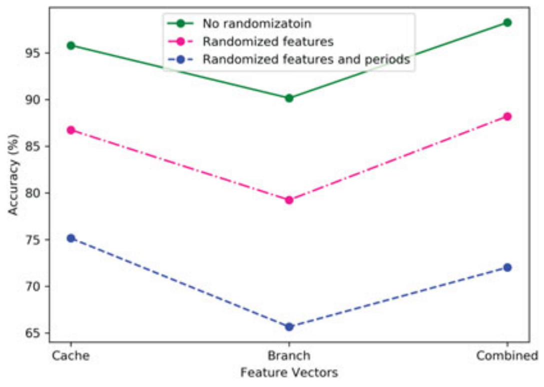


Fig. 14. Reverse-engineering features used for detection.

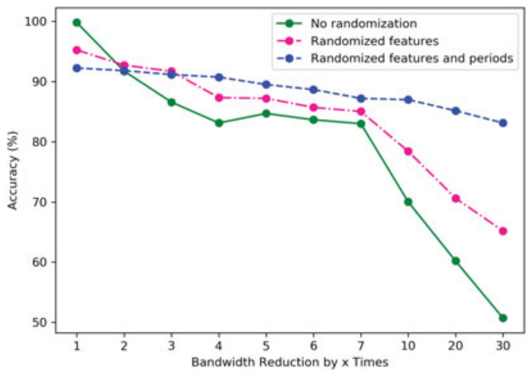


Fig. 16. Detection accuracy for evasion resilient *Spectre* detector.

ting more evasive especially in the extreme condition where bandwidth reduction was 30X. On the other hand, the detection strategy which randomly switches between detectors with different features and sampling periods was able to maintain a relative better detection accuracy when the bandwidth reduction was greater than 2X. Even when faced with the most extreme conditions (bandwidth reduction = 30X) in our experiments, the detection accuracy was above 80 percent which was much better than 50 percent of the baseline detector. Therefore, although switching between a more diverse set of detectors had lower accuracy for the original *Spectre*, it was more resilient to evasion as the bandwidth decreases.

7 Conclusion and Future Research

As complete mitigation to the *Spectre* is challenging, it is more practical now to detect such attacks proactively. We proposed to detect *Spectre* attack by monitoring microarchitectural features collected from hardware performance counters commonly available in modern processors. An online detection approach was adopted to detect malicious behaviors during the attack and the experimental results showed a promising detection accuracy with only 0.77 percent of false positives and no false negatives using a MLP classifier when detecting the original *Spectre-V1* under normal workloads. We also tested the detector under different workload and it was able to maintain an accuracy above 90 percent.

In addition, We demonstrated the feasibility of a redesigned *Spectre-V1* which evades HPC-based malware detectors and proposed strategies to reshape the microarchitectural profile of the attack by putting attacks to sleep or inserting instructions. We showed that putting *Spectre* to sleep after it has performed malicious tasks allows an attacker to effectively evade simple LR malware classifiers and maintain as high a success rate as 86 percent with a concomitant 7X bandwidth reduction. Complex models such as MLP mean higher resiliency to evasion, however, with a 10X bandwidth reduction, and a lower 70 percent detection accuracy.

Finally, we sought to improve the resiliency to evasion of the detection model by randomly switching between detectors with different settings. For the detector that consists of 6 different detectors with randomized features and sampling periods, our experimental results showed that the attacker could not reverse-engineer the accurate settings of the victim and the detector was able to maintain the detection accuracy above 80 percent to counter evade as the side channel bandwidth decreased by as much as 30X.

In terms of future research, we will look into reducing the performance overhead of the detector such as using dedicated hardware to implement the malware classifier. We will also expand the work to detect other attacks targeting GPU and other hardware vulnerabilities. As malware becomes pervasive and stealthier, future security mechanisms should actively monitor even the subtlest anomalies or signs of malware infection in every layer of the system from networks, software applications to hardware. The defense system should also be able to proactively respond to and remedy threats and be easily reconfigured to future attacks. More sophisticated machine learning algorithms using features not only from the hardware performance counters will be explored to predict program execution behaviors to prevent attacks at an earlier stage. Online learning algorithms will also be experimented with to involve real-time user feedback and respond to ever-changing malware behaviors.

Acknowledgments

This work was supported in part by the National Science Foundation (NSF) under Grant CNS-2026675. 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.

References

[1] P. C. Kocher, “Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems,” in *Proc. Annu. Int. Cryptol. Conf.*, 1996, pp. 104–113.

[2] W. Schindler, “A timing attack against RSA with the chinese remainder theorem,” in *Proc. Second Int. Workshop Cryptographic Hardware Embedded Syst.*, 2000, pp. 109–124.

[3] W. Schindler, “Optimized timing attacks against public key cryptosystems,” *Statist. Risk Model.*, vol. 20, no. 1–4, pp. 191–210, 2002.

[4] D. Brumley and D. Boneh, “Remote timing attacks are practical,” in *Proc. 12th Conf. USENIX Secur. Symp.*, 2003, pp. 1–13.

[5] P. Kocher, J. Jaffe, and B. Jun, “Differential power analysis,” in *Proc. Ann. Int. Cryptol. Conf.*, 1999, pp. 388–397.

[6] J.-S. Coron and L. Goubin, “On boolean and arithmetic masking against differential power analysis,” in *Proc. Second Int. Workshop Cryptographic Hardware Embedded Syst.*, 2000, pp. 231–237.

[7] J. Waddle and D. Wagner, “Towards efficient second-order power analysis,” in *Proc. Int. Workshop Cryptographic Hardware Embedded Syst.*, 2004, pp. 1–15.

[8] K. Gandolfi, C. Mourtel, and F. Olivier, “Electromagnetic analysis: Concrete results,” in *Proc. Int. Workshop Cryptographic Hardware Embedded Syst.*, 2001, pp. 251–261.

[9] J.-J. Quisquater and D. Samyde, “Electromagnetic analysis (EMA): Measures and counter-measures for smart cards,” in *Proc. Int. Conf. Res. Smart Card Program. Secur.*, 2001, pp. 200–210.

[10] M. G. Kuhn, “Optical time-domain Eavesdropping risks of CRT displays,” in *Proc. IEEE Symposium Secur. Privacy*, 2002, pp. 3–18.

[11] A. Shamir and E. Tromer, “Acoustic cryptanalysis: On nosy people and noisy machines,” in *Proc. Eurocrypt Rump Session*, 2004.

[12] D. A. Osvik, A. Shamir, and E. Tromer, “Cache attacks and countermeasures: The case of AES,” in *Proc. Cryptographers Track RSA Conf. Topics Cryptol.* 2006, pp. 1–20.

[13] D. J. Bernstein, “Cache-timing attacks on AES,” *Technical Report, Univ. Illinois*, IL, USA, Rep. 60607–7045, 2005.

[14] Y. Yarom and K. Falkner, “FlLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack,” in *Proc. 23rd USENIX Conf. Secur. Symp.*, 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,” in *Proc. IEEE Symp. Secur. Privacy*, 2015, pp. 605–622.

[16] O. Aciiçmez, S. Gueron, and J. P. Seifert, “New branch prediction vulnerabilities in OpenSSL and necessary software countermeasures,” in *Proc. IMA Int. Conf. Cryptogr. Coding*, 2007, pp. 185–203.

[17] O. Acıiçmez, Ç. K. Koç, and J. P. Seifert, “Predicting secret keys via branch prediction,” in *Proc. Cryptographers Track RSA Conf.*, 2007, pp. 225–242.

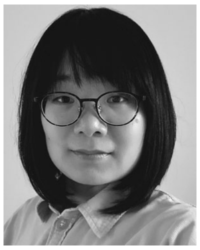
[18] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, “Jump over ASLR: Attacking branch predictors to bypass ASLR,” in *Proc. IEEE/ACM Annu. Int. Symp. Microarchit.*, 2016, pp. 1–13.

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

[20] P. Kocher *et al.*, “Spectre attacks: Exploiting speculative execution,” in *Proc. 40th IEEE Symp. Secur. Privacy*, 2019, pp. 1–19.

[21] J. Demme *et al.*, “On the feasibility of online malware detection with performance counters,” in *Proc. 40th Annu. Int. Symp. Comput. Archit.*, 2013, pp. 559–570.

[22] A. Tang, S. Sethumadhavan, and S. J. Stolfo, "Unsupervised anomaly-based malware detection using hardware features," in *Proc. Int. Workshop Recent Adv. Intrusion Detection*, 2014, pp. 109–129.
[23] K. N. Khasawneh, M. Ozsoy, C. Donovick, N. Abu-Ghazaleh, and D. Ponomarev, "Ensemble learning for low-level hardware-supported malware detection," in *Proc. 18th Int. Symp. Recent Adv. Intrusion Detection*, 2015, pp. 3–25.
[24] 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, 2017, Art. no. 564.
[25] A. Mambretti, M. Neugschwandtner, A. Sorniotti, E. Kirda, W. Robertson, and A. Kurmus, "Speculator: A tool to analyze speculative execution attacks and mitigations," in *Proc. 35th Annu. Comput. Secur. Appl. Conf.*, 2019, pp. 747–761.
[26] V. C. Windeck, "CPU vulnerabilities specter-ng: Updates are rolling in," 2018. [Online]. Available: https://www.heise.de/security/meldung/CPU-Sicherheitsluecken-Spectre-NG-Updates-rollen-an-4051900.html
[27] G. Maisuradze and C. Rossow, "Ret2spec: Speculative execution using return stack buffers," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2018, pp. 2109–2122.
[28] K. Wang, J. J. Parekh, and S. J. Stolfo, "Anagram: A content anomaly detector resistant to mimicry attack," in *Proc. 9th Int. Conf. Recent Adv. Intrusion Detection*, 2006, pp. 226–248.
[29] D. Bruschi, L. Cavallaro, and A. Lanzi, "An efficient technique for preventing mimicry and impossible paths execution attacks," in *Proc. IEEE Int. Perform., Comput. Commun. Conf.*, 2007, pp. 418–425.
[30] M. Kayaalp, T. Schmitt, J. Nomani, D. Ponomarev, and N. Abu-Ghazaleh, "Scrap: Architecture for signature-based protection from code reuse attacks," in *IEEE 19th Int. Symp. High Perform. Comput. Archit.*, 2013, pp. 258–269.
[31] K. N. Khasawneh, N. Abu-Ghazaleh, D. Ponomarev, and L. Yu, "RHMD: Evasion-Resilient Hardware Malware Detectors," in *Proc. Annu. Int. Symp. Microarchit., MICRO*, 2017, pp. 315–327.
[32] J. D. Frank, "Artificial intelligence and intrusion detection: Current and future directions," in *Proc. Nat. 17th Comput. Secur. Conf.*, 1994, pp. 1–12.
[33] Tom M. Mitchell, Machine learning, New York, NY, USA: McGraw-Hill, 1997.
[34] N. Cristianini and J. Shawe-Taylor, *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*. Cambridge Univ. New York, NY, USA: Press, 2000.
[35] B. Scholkopf and A. J. Smola, *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. Cambridge, MA, USA: MIT Press, 2001.
[36] C. M. Bishop, *Neural Networks for Pattern Recognition*. New York, NY, USA: Oxford Univ. Press, 1995.
[37] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference and Prediction*, 2nd ed. New York, NY, USA: Springer, 2009.
[38] A. Bors and M. Gabbouj, "Minimal topology for a radial basis functions neural network for pattern classification," *Digit. Signal Process.*, vol. 4, pp. 173–188, Jul. 1994.
[39] M. Ozsoy, C. Donovick, I. Gorelik, N. Abu-Ghazaleh, and D. Ponomarev, "Malware-aware processors: A framework for efficient online malware detection," in *Proc. IEEE 21st Int. Symp. High Perform. Comput. Archit.*, 2015, pp. 651–661.
[40] E. Keogh, S. Chu, D. Hart, and M. Pazzani, "Segmenting time series: A survey and novel approach," *Data Mining Time Ser. Databases*, 2004, pp. 1–21.
[41] M. Vafaeipour, O. Rahbari, M. A. Rosen, F. Fazelpour, and P. Ansarirad, "Application of sliding window technique for prediction of wind velocity time series," *Int. J. Energy Environ. Eng.*, vol. 5, no. 2, May 2014, Art. no. 105.
[42] C. Li and J.-L. Gaudiot, "Online detection of spectre attacks using microarchitectural traces from performance counters," in *Proc. 30th Int. Symp. Comput. Archit. High Perform. Comput.*, 2019, pp. 25–28.



Congmiao Li (Member, IEEE) received the BS degree in computing (computer science) with minor in mathematics and the MS degree in electrical engineering from the National University of Singapore, in 2009 and 2013, respectively, and the PhD degree in computer engineering from the University of California, Irvine (UCI), in 2020. She is currently a postdoc with UCI. Her research interests include security and computer architecture.

Jean-Luc Gaudiot (Life Fellow, IEEE) received the Diplôme d'Ingénieur from the École Supérieure d'Ingénieurs en electronique with Electrotechnique, Paris, France, in 1976, and the MS and PhD degrees in computer science from UCLA, in 1977 and 1982, respectively. He is currently a distinguished professor with the Department of Electrical Engineering and Computer Science, UC Irvine. Prior to joining UCI in 2002, he was a professor of electrical engineering with the University of Southern California in 1982. He has authored or coauthored more than 250 journal and conference papers. His research interests include multithreaded architectures, fault-tolerant multiprocessors, and implementation of reconfigurable architectures. His research has been sponsored by NSF, DoE, and DARPA, as well as a number of industrial companies. He has served the community in various positions and was the President of the IEEE Computer Society in 2017.