CR-Spectre: Defense-Aware ROP Injected Code-Reuse Based Dynamic Spectre

Abhijitt Dhavlle*, Setareh Rafatirad†, Houman Homayoun†, Sai Manoj Pudukotai Dinakarrao*
*George Mason University, Fairfax, VA, USA. †University of California Davis, Davis, CA, USA.
Email: {adhavlle, spudukot}@gmu.edu, {srafatirad, hhomayoun}@ucdavis.edu

***Abstract*—Side-channel attacks have been a constant threat to computing systems. In recent times, vulnerabilities in the architecture were discovered and exploited to mount and execute a state-of-the-art attack such as Spectre. The Spectre attack exploits a vulnerability in the Intel-based processors to leak confidential data through the covert channel. There exist some defenses to mitigate the Spectre attack. Among multiple defenses, hardware-assisted attack/intrusion detection (HID) systems have received overwhelming response due to its low overhead and efficient attack detection. The HID systems deploy machine learning (ML) classifiers to perform anomaly detection to determine whether the system is under attack. For this purpose, a performance monitoring tool profiles the applications to record hardware performance counters (HPC), utilized for anomaly detection. Previous HID systems assume that the Spectre is executed as a standalone application. In contrast, we propose an attack that dynamically generates variations in the injected code to evade detection. The attack is injected into a benign application. In this manner, the attack conceals itself as a benign application and generates perturbations to avoid detection. For the attack injection, we exploit a return-oriented programming (ROP)-based code-injection technique that reuses the code, called gadgets, present in the exploited victim's (host) memory to execute the attack, which, in our case, is the CR-Spectre attack to steal sensitive data from a target victim (target) application. Our work focuses on proposing a dynamic attack that can evade HID detection by injecting perturbations, and its dynamically generated variations thereof, under the cloak of a benign application. We evaluate the proposed attack on the MiBench suite as the host. From our experiments, the HID performance degrades from 90% to 16%, indicating our Spectre-CR attack avoids detection successfully.**

I. Introduction

The insurgence of the side-channel attacks [1], [2], attacks that exploit the inherent vulnerability in a system while trying to snoop on secure sensitive applications, at an alarming rate, is considered one of the pivotal issues. Spectre [3] is one such recently introduced powerful exploit that targets the vulnerability in modern branch predictors. Spectre 'mistrains' a branch predictor to perform legitimate operations initially, and later, it forces an erroneous speculative execution, which leaks sensitive data over a covert channel.

There exist detection mechanisms [4]–[8] to mitigate Spectre attacks by employing machine learning (ML)-based detectors, also known as Hardware-assisted Intrusion Detection (HIDs). The seminal theme of these works [4]–[8] is to train the ML-based detectors on the microarchitectural patterns[1] of the executing applications. The performance counters can extract different microarchitectural information regarding the application, such as cache-hits, cache-miss, total cycles, instruction count, etc. Spectre affects the branch predictor, cache, memory-related instructions' patterns during its execution [4], [5]. The existing defense techniques such as [4]–[7] utilize the affected performance counters' patterns to differentiate an attack and a benign application.

Traditionally, Spectre is launched as a standalone attack. However, in a system where an adversary does not have the permissions to execute a malicious binary as a standalone application, there is a need to evade the conventional launch process. Also, the HID detects and protects the system by profiling the applications, thus guarding the system against side-channel attacks, such as Spectre.

On the other hand, there exists a genre of attacks known as *Code-reuse* attacks [9], [10], which operate by subverting the control flow of the victim without directly manipulating the victim application or memory. Return-oriented programming (ROP) [10], [11] is one such example of a code-reuse attack. The methodology of the ROP attack is to target fragments of code, generically known as *gadgets* in the victim that end with *ret* (return) instruction. By chaining such gadgets together, an attacker can perform a Turing-complete manipulation to execute malicious instructions. Hence, the attack is also known as an ROP-chain attack. The attack redirects the program flow to the malicious code, thus, hijacking the control flow of the victim application. With the instruction code already existing in the memory, victim application can be forced by ROP attack to execute a malicious code, hence the term 'code-reuse.'

There exist techniques that can mitigate the ROP attack, such as Stack Canaries [12], and Address Space Layout Randomization (ASLR) [13]. ASLR works by randomizing the addresses in the memory. Although address randomization can deter an ROP-attack by randomizing the address space, the ASLR defense can be circumvented [14]–[17]. Stack Canaries [12] is memory protection that inserts a randomly chosen value in the stack between the local variables and return address. When a function call returns, it checks the value for any corruption. If the value is overwritten, the program exits without executing further. Similar to ASLR, Stack Canaries technique can also be evaded to launch a ROP attack.

Yet another class of defenses for thwarting Spectre attack are InvisiSpec [18] and Context-Sensitive Fencing [19]. The former technique works by making speculative execution invisible to the system and other applications. It uses a *speculative buffer* to save data from load instruction until the load is deemed safe; later, the data is re-loaded to local caches, which also affects the microarchitecture. The latter defense employs a microcode customization mechanism allowing processors to insert fences

[1]Microarchitectural traces are obtained from the performance monitoring unit (PMU). These features are also known as hardware performance counters.

978-3-9819263-6-1/DATE22/©2022 EDAA

into the dynamic instruction stream to mitigate undesirable side-effects of speculative execution [19]. Both the defenses are employed at software-level, inducing overheads and require architecture level modifications [18], [19]. In contrast, this work targets system that demonstrate machine learning assisted mechanisms in detecting and mitigating Spectre. And, for injecting the attack, this work exploits buffer overflow vulnerability.

In this work, we exploit the ROP code-injection attack as a promising methodology to launch a malicious application, such as Spectre, that intends to steal sensitive information. The mechanism offers the benefit of attack injection without explicitly writing to victim's memory and using existing code in the memory. Mere integration of code-reuse attack with Spectre can be still vulnerable to the existing defense techniques [4]–[8]. To alleviate the detection, we pivot the proposed *code-reuse Spectre (CR-Spectre)* attack on the ROP injection and dynamic adaptation to keep the malicious behavior undetected by existing detection techniques. The dynamic adaptation in the CR-Spectre generates perturbations, thus contaminating the HPC generated by the host with the injected malicious application. These dynamic patterns intend to degrade HID performance, forcing misclassification of the attack.

The advantage of the proposed CR-Spectre compared to other Spectre variants [3], [20], [21] is its distributed nature and the capability to be a moving target for the defender, especially the ML-based solutions such as [4], [5], [7], [8]. Our proposed attack is capable of extracting secret information from an application while evading ML-based detection. This work evaluates the proposed attack on a Hardware-based intrusion detection system (HID) utilizing machine-learning (ML) [4]–[6], [22], that provides inference based on unique application traces, hardware performance counters (HPC), rendered by hardware performance monitoring unit (PMU).

In summary, the essential contributions of this work are:

1) Propose CR-Spectre attack, capable of executing under the cloak of a benign (white-listed) application as a launching mechanism.
2) A dynamic attack capable of modifying microarchitectural state to render the attack more robust against an HID is introduced.
3) A thorough evaluation of the performance of CR-Spectre under different scenarios are presented. Another aspect is to evaluate the overhead of the proposed attack.

We present our results and evaluation of the proposed attack and demonstrate that CR-Spectre can help degrade HID performance, thus misclassifying benign from an attack application. Experiments are performed using Spectre [3], and MiBench benchmark suite [23]. In our experiments, the HID performance degrades from 90% to 16%, indicating our CR-Spectre attack evades detection successfully.

II. Proposed CR-Spectre Attack

A. Threat Model

There exists an adversary that intends to steal secret data from an application that processes sensitive data. The adversary employs attack code to steal secret data from the target application (target). The exploited victim (host application) is the application into which the adversary injects the attack - the attack refers to CR-spectre - and the intention is to steal data from the target application (target). For the demonstration of the attack, we keep the *secret* as an array that is stored in the host application; the host never accesses the secret. The CR-Spectre attempts to read the *secret* in the array. Similar to [3], we assume that the adversary knows the address of the *secret* processed by target. The adversary has no special or root privileges to execute the attack. CR-Spectre is tested on HIDs that are inspired from the recent works presented in [4]–[6], [24], all of which utilize the HPC information for training the machine learning classifiers. CR-Spectre attempts to inject malicious code to steal secret information from the target while evading HID detection using perturbations. For an ROP-chain attack to function, there needs to be a mechanism to overflow the buffer and rewrite the stack contents. Hence, it is a prerequisite that the host application makes a write operation to the buffer, controlled by the adversary.

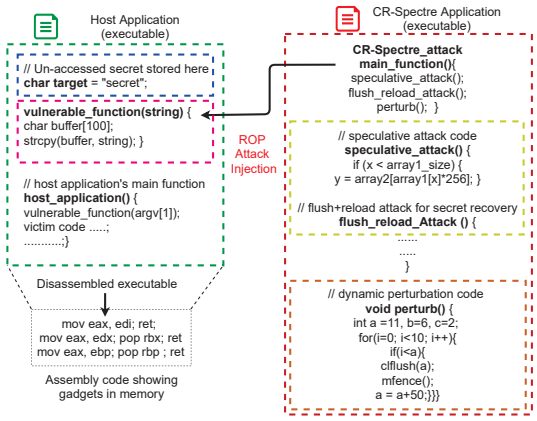

Fig. 1. CR-Spectre program flow

B. Overview of the Proposed CR-Spectre

Here, we explain the overview of the attack injection, dynamic perturbations, and HID for detection. Figure 1 shows the attack process flow depicting various aspects of the attack. There are five aspects to CR-Spectre attack - host, vulnerable code fragment, target, speculative attack code, and dynamic perturbations. The host is the application to which the malicious code (Spectre) is injected. The adversary attempts to access the *secret* stored in the target application. The vulnerable code is the host application's code fragment that serves as a point-of-entry for the ROP attack. The speculative attack code exploits the vulnerability in computing systems to access unauthorized memory locations. At the same time, the dynamic perturbation is proposed to contaminate the victim's (host) and speculative attack's HPCs to degrade HID performance. The secret (target) is stored in the same application as the host, sharing the memory space, but it can be contained as a standalone application. The proposed attack is initiated with the knowledge of the vulnerability in the host. In our case, we utilize buffer overflow vulnerability to launch the ROP attack.

The host expects a string of a certain length, and it is stored in a buffer. The ROP attack is deployed by passing to a host a string that exceeds the buffer's capacity/length. This overwrites the contents in the stack space, which corrupts the return address of the calling function as shown in Algorithm 1. The string passed to the host also contains arguments that will be needed by the 'execve' as its argument, for example, the address of the malicious binary. Addresses of the ROP gadgets are also provided as the input string to the host. Hence, the host returns to a series of gadgets carefully chosen to make an 'execve' system call and inject the malicious binary. After injecting the malicious binary, speculative execution application, it will attempt to access the secret in the target. The address of the secret in the target is known to the adversary. The computing system is protected by HID, which samples the HPCs of applications executing on the system in runtime.

The HID can detect the speculative attack explained above with high accuracy. Hence, it becomes necessary to conceal the attack. We propose to conceal by introducing dynamic perturbations. The dynamic perturbations are generated by calling functions containing a couple of 'if' loops that execute based on the values of the attack parameters (variables in the loop). The *clflush* and the *mfence* instructions ensure that the data is flushed each time the function executes to cause variations in the HPC patterns. The perturbations can be modified dynamically by varying the parameters; thus, each generated variant producing a different HPC pattern. The dynamic perturbation is discussed in detail in Section II-E and Algorithm 2. With the CR-Spectre, the HPCs of both the host and attack are contaminated, leading to performance degradation of the HID, thus evading the defense.

C. CR-Spectre: Attack Methodology and Gadget Generation

Referring to Section I, and for conciseness, we present our proposed CR-Spectre by considering a simple application as shown in Algorithm 1 that stores the string provided as an argument in the buffer. The *host_application()* is the *main* function of the host application. For our experiments, we utilize the MiBench suite as the host; any other application could be used as a host, as the proposed technique is not bound to host application. We load the compiled victim binary in the Linux Debugger (GDB) to search for all instructions that end in a *ret* instruction. We then carefully chose instructions such that by chaining them together, the ROP attack makes a system call, executing the malicious attack. Due to Data Execution Prevention (DEP), system-level memory protection that marks stack and heap as non-executable, we cannot write malicious code to the victim's memory; an ROP-chain attack utilizes existing code in victim's memory to evade DEP protection.

As the existing code in memory is marked executable, the aim is to setup the stack memory such that the sequential execution (chain) of gadgets executes a system call, "execve" in this case, which takes the path name of the CR-Spectre binary as an argument. The ROP attack exploits a vulnerable function, *vulnerable_function*, to serve as an entry point for the attack. However, the proposed attack with ROP-chain [9], [11] can be extended to any victim where a return address can be manipulated to execute a gadget.

Algorithm 1 Pseudocode for code reuse attack on victim

```
1: vulnerable_function(char* string) {
2:         char buffer[100];
3:         strcpy(buffer, string); // ROP attack injection with
                                   buffer overflow exploit }
4: host_application(int argc, char** argv) {
5:         vulnerable_function(argv[1]);
6:         victim code line 2 ...
7:         victim code line 3 ...
8:         victim code line 4 ...
9:         victim code line 5 ...
10:        return 0; }
```

In Algorithm 1, the buffer overflow manipulates the return address, replacing it with an address of a gadget. Likewise, all the addresses of the necessary gadgets are provided as arguments to the vulnerable function, thereby chaining them to execute the CR-Spectre binary using the system call. A binary compiled using GCC has various other libraries linked with it, thus providing more gadgets than available only with the host. With sufficient gadgets, there exist innumerable possibilities of what could be executed within the victim [11]. The content of the argument, as shown in Listing 1, is 108 bytes (0x6C) of random data (all 'D's along with four bytes of 'FFFF' used to fill the buffer), followed by the address of *execve* function, followed by four bytes (ABCD), finally followed by the address of the Spectre binary.

Listing 1. Attack payload passed as argument for ROP attack

```
./victim_application "$(python -c 'print"D"*0x6C
+ "FFFF" + "address of system"
+ "ABCD" + "address of attack function"')"
```

For conciseness, we omit to show all the addresses of the gadgets accessed before finally making the system call. A working example of the code-reuse ROP attack is available on our anonymous repository[2]. The argument essentially fills in all the space in the buffer, shown in Algorithm 1, overwrites the return address to the address of the gadget in memory, address of the second gadget in the chain, and so on. Finally, it is followed by the "execve" gadget address and the address of the CR-Spectre binary executable, which is external to the host application. Hence, the host can execute a malicious code without writing to its memory, utilizing the gadgets already in the memory. The attack code is not contained in the host's code segment, instead it is injected in runtime; hence the HID cannot abort it by analyzing offline.

D. Attacking HID

Figure 2(a) shows how CR-Spectre attacks the HID. The CR-Spectre code is injected[3] into the host application. During execution, the application is profiled by the detector to record performance counters in runtime [4], [5], [7]. The HID monitors the recorded traces, and inference is provided - attack or benign. The decision is measured in terms of accuracy over time. The HID performance is discussed in Section III. For

[2]https://github.com/hartanonymous3512/CR-Spectre

[3]Injection refers to the ROP attack that subverts the control of the host application forcing it to execute a malicious code.

Algorithm 2 Pseudocode for generating dynamic perturbations for the CR-Spectre attack

```
1:  void perturb() {
2:     int a =11, b=6;
3:     for(i=0;i<10;i++) {
4:        if(i < a) {
5:           cflush(address(a));
6:           mfence();
7:           a = a+50; }
8:
9:        if(i < b) {
10:          cflush(address(b));
11:          mfence();
12:          b = b+10;
13:          cflush(address(b));
14:          mfence();
15:          b = b-10; }
16:          .....More loops can be added here.....
17:                    }
18:                 }
```

an HID, a higher accuracy refers to distinguishing between benign and attack situations more accurately. The purpose of the proposed CR-Spectre is to degrade the performance of the HID to evade detection. Figure 2(b),(c) visually present the difference between traditional Spectre and CR-Spectre. In (b), the adversary exploits attack code to steal secret data from the target application. On the contrary, as shown in (c), CR-Spectre injects the malicious code in a host (benign) application and executes it under the umbrella of the host.

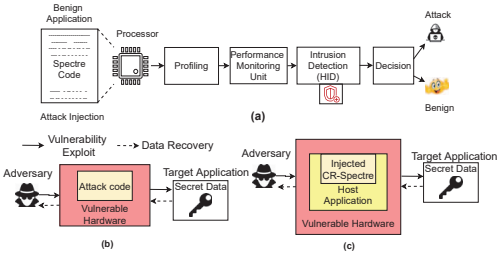

Fig. 2. (a) Code injection of Spectre attack to evaluate HID performance, (b) Traditional Spectre attack strategy, (c) CR-Spectre attack strategy

E. Defense-aware Dynamic Perturbation Generation

With the previously explained attack methodology, there can exist scenarios where the attack cannot evade the detection because the HID can learn, online learning or retraining, or the application can be tagged as an attack by the human-in-the-loop. Online learning type HIDs are retrained on the augmented dataset, the profiled HPC patterns of the applications during the runtime for robust threat detection. To add better evasion despite having online learning capable HIDs, we propose dynamicity in the CR-Spectre attack injected through ROP discussed in Section II-C. This affects the microarchitectural behavior of the application such that the monitored information by the HID can be different from the traces on which the HID is trained.

Figure 3 shows the process of the attack and generation of perturbed variations. CR-Spectre generates a perturbed version of the speculative attack code and injects it into the host. The applications, benign and the CR-Spectre attack, are profiled to record performance counters (HPCs). The profiled traces are fed to the ML-based HID. The HID provides the inference with a certain accuracy, indicating if the attack is detected or not. For

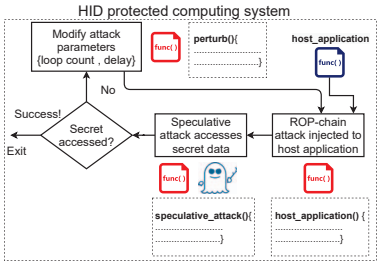

Fig. 3. Process flow of the ROP attack and generation of perturbed code

the attack to evade the HID detector, we consider accuracy of 55% or less. Suppose the HID inference result accuracy is less than 55%. In that case, the attack successfully degrades the detector performance while the malicious attack steals secret data from the target. If the HID detects the attack with high accuracy (>80%), we consider that the attack was detected. The HID performs realtime profiling of the applications executing on the system [4], [5], [7].

Upon detecting the CR-Spectre attack, we modify the perturbation code's parameters to generate a variant, the HPC traces of which differ from the previous variant. A variant is generated by modifying the attack parameters like the loop count and operation variables, 'a' and 'b', as shown in Line 2 of Algorithm 2. The parameters are utilized in the algorithm as shown in Lines 4, 7, 9, 12, 15, 17, 20, and 23. The parameters affect the *clflush* instruction; hence it varies the HPC patterns as well. With different attack parameters, the generated HPC patterns are modified. The attack process is repeated to steal secrets from the target. The benign applications running on the system are also profiled and fed to the HID. This is necessary because, in a real-world situation, the system executes multiple applications. Hence, we profile applications like browsers, text editors, etc., and train the HID to emulate a practical situation. The code shown in Algorithm 2 is called from within the malicious code, Spectre.

The *cflush* on the arithmetic operation triggers a *cache_miss* and affects other hardware events such as those related to branch prediction, the number of instructions executed, and the cache access cycles. The *mfence* instruction ensures that the previous operation, clflush, completes before proceeding with the operation below. The data recovery process is elaborated in [3]. For conciseness, we only discussed situations where the generated perturbations (HPC) increase in magnitude. Nevertheless, we can use a delay loop to disperse generated perturbations, thus distributing them in time. In this manner, the generated HPC patterns can also reduce in magnitude.

III. Results and Evaluation

A. Experimental setup

MiBench [23], Spectre [3] are used as the host and malicious attack applications, respectively. The CR-Spectre attack is not limited to the applications reported here, but it can exploit other vulnerable applications, thus reading a specified unauthorized memory location in the system. PAPI-based profiling tool [7] is utilized for recording the performance monitoring unit's output,

hardware performance counters (HPCs). All experiments, application profiling, ROP attacks are executed on Ubuntu 18.04 running on an Intel i5 with 16 GB RAM. We collect a total of 2000 samples for each class, CR-Spectre, and host; the scope of applications profiled also includes the host and other benign applications like browsers, text editors, etc. The training and testing datasets are separated in a ratio of 70/30. We evaluate HID performance on MLP (Sklearn) [4], Neural Network (NN) from Tensorflow [5], [6], Logistic Regression (LR) [4], [5], and a SVM [4], [5] classifier. The HIDs' parameters are as follows: the neural networks have 6-layers using 'Relu' activation; SVM classifier uses a linear kernel for classification; the MLP is 3-layer network-based classifier. The parameters for the hidden layers are determined experimentally. We choose the parameters that deliver high accuracy in detecting CR-Spectre from other applications profiled. Features used for the training are 'total cache misses', 'total cache accesses', 'total branch instructions', 'branch mispredictions', 'total number of instructions', and 'total cycles'. The first five features are affected by Spectre attack as presented in works [4], [5]. The last feature is utilized for the IPC metric for overhead analysis.

B. Results

1) HID Performance on Spectre Detection: Figure 4 shows the performance (accuracy) of the HID in detecting/differentiating the benign (host and other applications) and Spectre applications. The HID is inspired from [4]–[6] and utilizes similar features for Spectre detection. The features fed to HID are the recorded performance events (HPCs). We collect a total of 56 performance events available on the system (offline). For real-time monitoring of the events, a limit is imposed on the number of events counted simultaneously. Hence, we present the results with multiple feature sizes (1, 2, 4, 8, and 16) to show the efficiency of the HID system deployed in this work for Spectre detection. Figure 4 shows the performance of the HID in differentiating MiBench and Spectre [3] applications. We experimented with different variants of the Spectre attack, discussed in [20], [21]. The accuracy shown in the figure is the average of the variants of Spectre. The legend `Spectre_1` indicates the performance in classification of Spectre - and other variants averaged - and MiBench application-1 (Math application as listed in Table I), similarly we can interpret other legends. Performance with a few MiBench applications in Figure 4 are shown for conciseness. As seen, accuracy of more than 80% for feature sizes 16, 8, 4, and 2 in Spectre detection irrespective of the MiBench application used. However, using only one feature for classification is inefficient due to its inability to capture the HPC variations in a single feature. To alleviate the monitoring and computational overheads, we consider utilizing 4 features in this work for Spectre detection that lead to >90% accuracy on average. For the rest of the article, we consider a feature size of 4 which can be recorded in runtime on modern processors [5].

2) Does CR-Spectre Evade HID?: Figures 5 and 6 present the HID performance under attack. The accuracy metric is plotted against the number of CR-Spectre attack attempts over time. We study two scenarios for the attack, offline and online

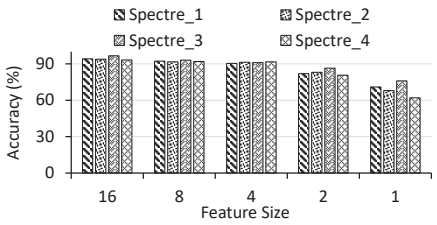

Fig. 4. HID performance for four benign (host) applications and original Spectre attack studied for different feature sizes

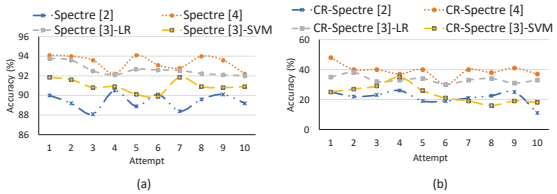

Fig. 5. Comparison of offline-type HID performance with Spectre and CR-Spectre attack

learning HID. The offline learning HID is a static type that does not retrain itself (or retrained by the defender) during runtime, i.e., similar to the [22]. On the contrary, we deploy an online learning version of the HIDs which are retrained during runtime on newer traces to enhance attack detection capability on unseen data.

Figure 5 presents the offline-type HID performance for original Spectre and CR-Spectre with HID using different types of ML classifiers. In Figure 5(a), it is seen that the original Spectre attack is detected with high accuracy by the HID detector implemented with different ML classifiers. The accuracy variations are observed due to the variations in the recorded HPC traces during each attack attempt. Whereas, in Figure 5(b), the performance of HID degrades with perturbed instances of the attack. The accuracy shows a degrading trend as the offline HID is employed. It is to be noted that we do not generate dynamic perturbations for an offline-type HID. The reason being the offline-type does not 'learn' or retrain itself on newer traces. Hence, to save the overhead, CR-Spectre only generates one variation of perturbation but does not modify the attack parameters dynamically every time it attacks the HID.

Similarly, from Figure 6(a), it is observed that the HID detects Spectre with high accuracy. The patterns are leveled compared to Figure 5(a), as the online-type HID, by retraining itself on new traces, hence becomes more robust to HPC trace variations during the recording phase. A degrading trend is again observed for the HID performance in Figure 6(b). The exception is that the HID attempts to boost the detection performance owing to retraining. Yet, with the introduced dynamic perturbations, the CR-Spectre performs well in degrading the HID detection accuracy to less than 55% to the lowest observed accuracy of 16% in our experiments. Under the cloak of such degraded performance, the speculative attack recovers the *secret* data from the target.

C. Overhead analysis

We perform overhead analysis of CR-Spectre by evaluating different applications in the MiBench suite. We select instruc-

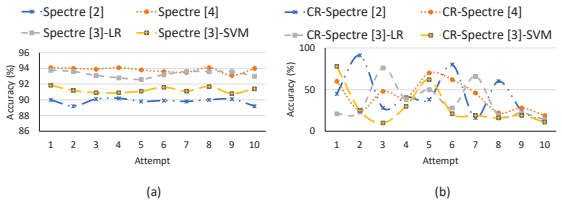

Fig. 6. Comparison of online-type HID performance with Spectre and CR-Spectre attack

tions per cycle (IPC) as an evaluation metric. Latency is also a metric other works [4]–[6], [22] have utilized. However, the latency metric could be counterproductive due to system noise. The noise is caused by other applications and the operating system running in the background. Furthermore, IPC is also considered as a trait of the application in determining the presence of abnormalities or stalls in the application. We mitigate trace fluctuations by averaging the values by iterating the same application 100 times.

TABLE I
PERFORMANCE OVERHEAD IN EVALUATED BENCHMARKS

Benchmark	Original Application (IPC)	CR-Spectre with offline-type HID (IPC)	CR-Spectre with online-type HID (IPC)
Math	1.9419	1.88	1.865
Bitcount 50M	3.041	3.05	3.031
Bitcount 100M	3.052	3.051	3.041
SHA_1	0.736	0.742	0.73
SHA_2	0.814	0.819	0.80

We report IPC values for the original application (without CR-Spectre), the offline execution of CR-Spectre, and the online execution of CR-Spectre. The aim is to deliver performance with negligible overhead. The overheads are reported in Table I. For the Math application (math_small and math_large applications averaged), the IPCs observed are 1.94, 1.88, and 1.865 for the original, offline, and online execution. Similarly, for the Bitcount with 50M operations, the IPCs are 3.041, 3.05, and 3.031, respectively. And for the SHA cryptographic algorithm, it is 0.814, 0.819, and 0.818, respectively. Again, we average the values to cover for variations. The overhead average for the offline-type and online type is 0.6% and 1.1%, respectively, compared to the Spectre-only attack without dynamic perturbations and ROP attack injection.

IV. COUNTERMEASURES

It is crucial to discuss countermeasures for proposed CR-Spectre attack to help mitigate potential security threats. Disable *clflush* and *mfence* instructions for non-privileged processes, thus disabling dynamic perturbations; Accompanying automatic HID detection with manual inspection of processes that might be vulnerable to ROP/Buffer-overflow exploits; Using a shadow memory -only accessible to the operating system - to compare and correct when return address manipulation takes place. However, further analysis and verification is needed to evaluate robustness against the proposed attack.

V. CONCLUSION

In this work, we proposed a novel ROP Injected Code-Reuse-based Spectre, CR-Spectre. The CR-Spectre exploits the ROP attack and the speculative execution vulnerability to inject malicious code in the host application. The malicious code is intended to steal secret data from the target application. We presented details on generating dynamic perturbations to evade HID defense. We discussed and evaluated HID detector performance in different scenarios. The proposed CR-Spectre was assessed on the online and offline type HIDs as well. With CR-Spectre, the HID performance is observed to degrade from 90% to 16% on an average. Based on the experimental results and evaluation, we conclude that CR-Spectre delivers high performance in degrading the HID system, yet posing negligible overhead, 0.6% for offline and 1.1% for the online-type, on the overall performance of the host application.

REFERENCES

[1] A. Dhavlle *et al.*, "Entropy-shield: Side-channel entropy maximization for timing-based side-channel attacks," in *International Symposium on Quality Electronic Design (ISQED)*, 2020.

[2] A. Dhavlle *et al.*, "Imitating functional operations for mitigating side-channel leakage," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.

[3] P. Kocher *et al.*, "Spectre attacks: Exploiting speculative execution," in *Symposium on Security and Privacy (SP)*, 2019.

[4] C. Li *et al.*, "Online detection of spectre attacks using microarchitectural traces from performance counters," in *Computer Architecture and High Performance Computing*, 2018.

[5] B. A. Ahmad, "Real time detection of spectre and meltdown attacks using machine learning," 2020.

[6] J. Depoix *et al.*, "Detecting spectre attacks by identifying cache side-channel attacks using machine learning," *Advanced Microkernel Operating Systems*, p. 75, 2018.

[7] M. Chiappetta *et al.*, "Real time detection of cache-based side-channel attacks using hardware performance counters," *Appl. Soft Comput.*, vol. 49, 2016.

[8] M. Alam *et al.*, "Performance counters to rescue: A machine learning based safeguard against micro-architectural side-channel-attacks," Cryptology ePrint Archive, Report 2017/564, 2017.

[9] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Conf. on Computer and Communications Security*, 2007.

[10] A. Bhattacharyya *et al.*, "Smotherspectre," *SIGSAC Conference on Computer and Communications Security*, Nov 2019.

[11] R. Roemer *et al.*, "Return-oriented programming: Systems, languages, and applications," vol. 15, no. 1, 2012.

[12] T. H. Dang *et al.*, "The performance cost of shadow stacks and stack canaries," in *Symposium on Information, Computer and Communications Security*, 2015.

[13] P. Bania, "Security mitigations for return-oriented programming attacks," *CoRR*, 2010.

[14] G. F. Roglia *et al.*, "Surgically returning to randomized lib(c)," in *2009 Annual Computer Security Applications Conference*, 2009, pp. 60–69.

[15] A. Sotirov *et al.*, "Bypassing browser memory protections," in *In Proceedings of BlackHat*, 2008.

[16] *https://www.computerworld.com/article/2516793/hacker-busts-ie8-on-windows-7-in-2-minutes.html*.

[17] D. Evtyushkin *et al.*, "Jump over aslr: Attacking branch predictors to bypass aslr," in *International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–13.

[18] M. Yan *et al.*, "Invisispec: Making speculative execution invisible in the cache hierarchy," in *(MICRO)*, 2018.

[19] M. Taram *et al.*, "Context-sensitive fencing: Securing speculative execution via microcode customization," in *ASPLOS*, 2019, p. 395–410.

[20] E. M. Koruyeh *et al.*, "Spectre returns! speculation attacks using the return stack buffer," in *WOOT*, 2018.

[21] V. Kiriansky *et al.*, "Speculative buffer overflows: Attacks and defenses," *CoRR*, 2018.

[22] T. Zhang *et al.*, "Cloudradar: A real-time side-channel attack detection system in clouds," in *RAID*, 2016.

[23] M. R. Guthaus *et al.*, "Mibench: A free, commercially representative embedded benchmark suite." USA: IEEE Computer Society, 2001.

[24] C. Li *et al.*, "Detecting malicious attacks exploiting hardware vulnerabilities using performance counters," in *COMPSAC*, 2019.