Systematical Evasion from Learning-based Microarchitecture Detection Tools

Debopriya Roy Dipta, Jonathan Tan and Berk Gulmezoglu

***Abstract*—Microarchitectural attacks threaten the security of individuals in a diverse set of platforms, such as personal computers, mobile phones, cloud environments, and AR/VR devices. Chip vendors are struggling to patch every hardware vulnerability in a timely manner, leaving billions of people's private information under threat. Hence, dynamic attack detection tools have become a popular way to detect ongoing attacks with the utilization of hardware performance counters and machine learning models. In this study, we evaluate the robustness of various ML-based detection models with a sophisticated fuzzing framework. The framework manipulates the hardware performance counters in a controlled manner with the help of individual fuzzing blocks. Later, the framework is leveraged to modify the microarchitecture attack source code and to evade the detection tools. We evaluate our fuzzing framework with time overhead, achieved leakage rate, and the number of trials to successfully evade the detection.**

***Index Terms*—anomaly detection, microarchitectural attacks, machine learning, fuzzing, side-channel attacks.**

I. Introduction

Side-channel attacks have been a serious threat against individuals and government entities in the last two decades. The attacks have been improved with more sophisticated techniques by leveraging power consumption measurements, timing differences, acoustic signals, and electromagnetic emissions to leak sensitive information. Microarchitectural attacks have emerged as remote side-channel attacks since attackers can collect side-channel information by only running benign-looking software on the victim's device or a shared computing platform. These attacks exploit secret-dependent operations by carefully monitoring microarchitecture state changes or manipulating hardware components. Nevertheless, hardware designers cannot keep up with the speed of new microarchitectural attacks to patch all the vulnerabilities. Hence, several Spectre-type [21], cache [16], [17], [20], Translation Look-aside Buffer [15], and port contention [4] attacks are still feasible in the latest generations of Intel and AMD processors.

The lack of timely patches pushes both cloud providers and hardware designers to implement real-time monitoring techniques to detect ongoing microarchitectural attacks. The proposed detection methodologies mostly rely on the available sensors in the system, such as power consumption [14], performance counters [18], [28], and thermal readings [33], to collect benign and malicious activities. The collected data samples are then processed with either statistical methods or machine learning models to distinguish any anomalies.

The authors are with the Department of Electrical and Computer Engineering, Iowa State University, Ames, IA 50010 USA (e-mails: roydipta@iastate.edu, jona1115@iastate.edu, bgulmez@iastate.edu).

Among the detection methods, the most popular ones are based on performance counters and machine learning to create more sophisticated models. These methods need to choose 1) the hardware performance counter (HPC) framework, 2) relevant performance counters for the targeted attacks, 3) the resolution of the data collection, and 4) ML/DL model that can detect anomalies. While a large portion of these detection models claim that their detection accuracy is considerably high, they are not tested against adversarial examples. This leads to the robustness issue of the trained models, making the detection models less trustworthy. Even worse, each detection model utilizes a different benchmark for their benign application dataset, which makes it more challenging to evaluate the performance of the models.

In this paper, our aim is to evaluate the robustness of proposed machine learning-based detection models with a unified benchmark dataset and an automated fuzzing framework. Our contributions are summarized as follows:

- Each performance counter used in the detection models is manipulated with specially crafted code snippets, including cache and branch-related counters.
- An automated fuzzing framework is developed to select the most appropriate counter-manipulation techniques and parameters to evade ML-based microarchitectural attack detection models.
- We introduce an average of 50% time overhead on the original attack source codes to be able to bypass detection tools.
- The weaknesses of the current ML-based detection methods are explained with empirical results and potential improvements are discussed for further research.

Outline. The rest of the paper is organized as follows: Section II provides background on microarchitectural attacks and defense mechanisms. Section III describes the threat model for the study. Section V outlines the overview of the proposed fuzzing technique. Section IV explains the created fuzzing blocks in detail. Section VI gives an overview of the reproduced detection models. Section VII evaluates the fuzzing tool against detection models. Section VIII provides an overview of closely-related studies. Section IX and Section X conclude the paper with discussion and important results.

II. Background

A. Microarchitectural Attacks

Microarchitectural attacks exploit the intricacies of hardware design to leak secure information, often bypassing traditional security measures. These attacks take advantage of

the side effects of normal operations within a computer's physical architecture, such as cache timings [17], [20], branch prediction [2], and speculative execution [21]. Depending on the instructions executed on the hardware, attackers can leak passwords [13], cryptographic keys [20], visited websites [31], and so on.

B. Hardware Performance Counters (HPCs)

Hardware Performance Counters (HPCs) provide low-level event information through dedicated model specific registers during the execution of software. The events cover a wide range of microarchitectural behaviors such as cache activity, branch prediction performance, front-end and back-end characteristics, and so on. These counters are used to profile a running code in a more detailed way. The counter values are then examined for the potential bottleneck of the software by developers. All Intel and AMD devices include these counters in their design to help software developers. The counters are mostly accessible through libraries such as PAPI [27], perf [9], and Intel PCM [1].

While these counters are actively utilized for reverse engineering [26] and side-channel attack implementation [19], their usage in dynamic microarchitecture attack detection is more prevalent in personal computers [18], and cloud environments [39]. The number of events, their diversity, and appropriate sampling rates remain active research areas as the microarchitecture attacks evolve, changing their effects on HPCs. Hence, many studies focus on selecting the best combination of counters to detect a diverse set of microarchitectural attacks with high accuracy.

C. Machine Learning-based Microarchitectural Attack Detection

There are many studies attempting to detect microarchitectural attacks in real-time. Some studies leverage basic statistical measures to distinguish attack executions and benign workloads []. However, they are not good at learning the complex behavior of microarchitecture attacks, leading to weak detection methods. Hence, machine learning and deep learning-based detection techniques have become more popular due to their superiority in learning relations between data samples []. The ML models used in these detection techniques span from basic models such as SVMs to more advanced models such as CNNs. These models can adapt themselves to the changing landscape of attacks better than statistical methods, making them more suitable for attack detection. A large portion of these studies train supervised models with benign and attack datasets and evaluate their models with real-time collected performance counter values.

III. Threat Model

In our threat model, we assume that an attacker is co-located with the victim on the same hardware (either a personal laptop or a shared cloud environment). The victim machine has an active dynamic detection tool, which monitors pre-determined hardware counters and distinguishes between benign and attack executions. The attacker aims to evade the detection tool while leaking secrets. The attacker has no information on what type of ML model is used to train the detection tool. The attacker only knows which counters are monitored by the detection tool. The attacker prepares the malicious code in a separate machine and the code is sent to the victim machine through a network channel. When the attacker executes a code snippet, the execution is stopped by the detection tool if an attack is detected. The attacker has no other feedback from the detection tool such as confidence ratio from the ML model. There is no limit on the number of executions an attacker can make on the victim's device. We perform our experiments on an Intel Comet Lake microarchitecture.

IV. Fuzzing Framework Overview

The fuzzing framework is designed to automatically modify the attack source code, requiring minimal human intervention and manual coding. The framework is based on the Python language to ease the insertion of code blocks to the source code. The purpose of the framework is to manipulate the targeted hardware performance counters by modifying the source code by inserting wait times, and additional instructions and changing the number of measurements while leaking secrets and evading ML-based detection tools.

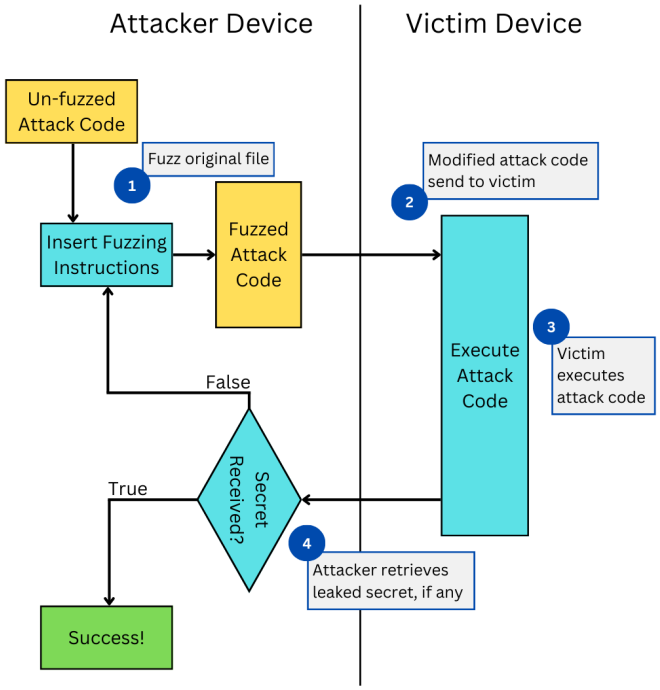

Fig. 1: Fuzzing Framework

Figure 1 outlines the generic fuzzing framework. In Step 1, the attacker chooses the attack type that will be performed on the victim's machine. The required files, such as C codes, header files, and Makefile for the attack code, are prepared as an initial step. While the header files and Makefile remain the same, the C code is updated every iteration in the fuzzing process. A config file is written to tell the framework what configuration the experiment should be.

In step 2, the framework inserts C and/or assembly code snippets into certain parts of the attack code, targeting specific

HPCs. The insertion is fully automated in the framework as specific locations are pre-determined in the source code such that the side-channel information is not destroyed by an unexpected behavior of the newly added code snippets. After that, the modified files are compiled in the attacker machine and sent to the victim machine to be executed.

In Step 3, the attacker executes the attack code in the victim's machine. Once the attack code starts executing, the counter monitoring tool starts to collect statistics from the counters. We only consider PAPI [27] and perf [9] tools as monitoring tools in this study as they are the most popular ones. Once the attack code execution finishes, the monitoring tool stops the monitoring process. The side-channel measurements are written to a file and sent to the attacker's machine. We consider two potential outcomes in Step 3: 1) The attack code could be detected as a malicious activity and stopped executing the program, 2) The attack code executes the code without detected and the measurements are sent to the attacker-controlled machine.

In step 4, the attacker analyzes the returned measurements. If there are no measurements, it means the attack code stopped, and the secret cannot be leaked. In this case, the attack code goes through another iteration of fuzzing starting from Step 2. Otherwise, the secret is leaked, and the attack is successful. The attacker keeps the statistics related to the attack, such as the number of characters leaked (Spectre attack) or the percentage of recovered secret keys (cache attacks) as well as the time spent on leaking the secret. If the fuzzing is successful in leaking the secret, the framework terminates, and the attacker can view all fuzzing modifications to the original attack code file.

V. Fuzzing Technique

In this section, we discuss our fuzzing primitives that can effectively manipulate hardware performance counters (HPCs). We evaluate their functionality based on their effectiveness in modifying targeted counters, side-effects on un-targeted counters, and time overhead introduced by modifications.

There are many ways an attacker can introduce dummy code blocks into an attack source code to increase or decrease hardware counter values that are used by ML-based detection models [36]. As these counters are the only source of profiling the current activity in a system, they are crucial for the detection model to distinguish benign and malicious code executions. In our framework, we create fuzzing blocks that can be adapted to changing detection models, such as utilized performance counters, trained machine learning models, and profiling resolution. These changes are controlled by creating parameterized code blocks. For example, the number of iterations, the length of an array, and the number of instructions control the amount of cache misses. Given the parameterized nature of our fuzzing technique, we are able to manipulate the counters in a controlled manner in our framework.

Our purpose is to manipulate each counter separately while minimizing the effect of each fuzzing block on other counters. For this purpose, we evaluate each fuzzing block based on its side-effect on other counters as well as the time overhead brought by additional instructions. As an example, the number of branch instructions can be increased by introducing empty *for* loops. However, when the number of conditional jump instructions increases, the introduced time overhead rises in parallel. Hence, we seek to find a balance between time overhead and its effectiveness. At the same time, we aim to reduce the side effects of each fuzzing block, meaning that when targeting a specific HPC, the change in other counters is minimized.

A. Fuzzing Cache Related HPCs

The first fuzzing block targets cache-related counters. The attacks considered in the ML-based detection models leave a distinct fingerprint on the cache counters, such as L1/L3 cache misses and cache references. The manipulation of these counters is extremely important to create an efficient evasion method with fuzzing. The fuzzing framework is designed to insert a set of instructions that copies elements from one array to another, which increases the cache-related counters. In contrast, several slow-down techniques are integrated into the framework to decrease the occurrence of cache events.

The fuzzing block given in Listing 1 copies elements from an array s to another array d, which is located far apart in memory. We allocated both array's locations in memory greater than 2MB away to make sure that the prefetching mechanism does not fetch nearby cache blocks from memory. This code block allows an increase in the number of cache-related activities. Before running the function, the source array, s, is initialized to some arbitrary data using `memset`. After copying elements from array s to array d, `fuzz_simple_lw_sw` 2, both arrays are then flushed using the *clflush* instruction to ensure maximum cache access and/or miss rate.

It is to be noted that `memset` is used instead of `for` loops while initializing the source array (s) to lower fuzzing overhead. We noticed that when disassembling the C code, `memset` is disassembled to `call memset@PLT`, which runs 99% faster than for loops. The decrease in overhead is verified not only using PAPI, but also as an observation of the reduction in time overhead during the fuzzing process.

```
uint8_t *b = (uint8_t*) malloc(6 * ONE_MEGA_BYTE *
    sizeof(uint8_t));
uint8_t *s = b;
uint8_t *d = b + 2 * ONE_MEGA_BYTE;

memset(s, 0xA, FUZZ_SIMPLE_LW_SW_LEN);

fuzz_simple_lw_sw(s, d, FUZZ_SIMPLE_LW_SW_LEN);

k = 8; // Size in bytes of a x86 cache line
for (int i = 0; i < LEN * k; i += 8 * k) {
    flush((void *)(s + i));
    flush((void *)(d + i));
}
free(b);
```

Listing 1: Fuzzing code targeting cache and cache HPCs

```
asm volatile (
  "movq $0, %%rcx\n\t" //Initialize counter to 0
  "loop_start_2:\n\t"
  "movq (%%rsi, %%rcx, 4), %%rax\n\t" // Load
    integer from source array.
```

```
  "movq %%rax, (%%rdi, %%rcx, 4)\n\t" // Store it into destination array.
  "incq %%rcx\n\t" // Increment counter
  "cmpq %[count], %%rcx\n\t" // Compare counter with count
  "jl loop_start_2\n\t" // If counter is less, loop_start_2

// Output operands
  : [count] "r" ((uint64_t) {LEN}), "S" (s), "D" (d) // Input: count, src (esi), dest (edi)
  : "%rax", "%rcx", "memory" // Clobbered: rax, ecx, and memory to indicate memory is being modified
 );
```

Listing 2: Fuction `fuzz_simple_lw_sw`

Evaluation. The number of introduced cache misses and cache references are controlled by an input to *memset* command. This parameter decides how many copy operations will be executed to increase the cache-related events. In Figure 2, we show that 10,000 copy operations create around 26,000 cache misses and 57,000 cache references. These numbers gradually increase with the increasing number of copy operations, which can be managed by the parameterized input to the fuzzing block. Our experiments show that the time overhead is around 100μs if 10,000 copy operations are executed. The time overhead can rise up to 1.2 ms when 1.9×10^5 copy operations are inserted into the attack code. The time overhead indicates that an attacker can even increase these counters further since typical detection models sample the counters either every 10 ms or 100 ms. Hence, the balance between the side-channel measurement collection and dummy operations is crucial to evade the detection while introducing lower time overhead during the attack execution.

```
  movq  -80(%rbp), %rax
  movl  $10000, %edx
  movl  $10, %esi
  movq  %rax, %rdi
  call  memset@PLT
```

Listing 3: memset Disassembled

```
  movl  $0, -184(%rbp)
  jmp  .L9
.L10:
  movl    -184(%rbp), %eax
  movl    %eax, %ecx
  movl    -184(%rbp), %eax
  movslq  %eax, %rdx
  movq    -80(%rbp), %rax
  addq    %rdx, %rax
  leal    -24(%rcx), %edx
  movb    %dl, (%rax)
  addl    $1, -184(%rbp)
.L9:
  cmpl  $9999, -184(%rbp)
  jle  .L10
```

Listing 4: For Loops Disassembled

B. Fuzzing Branch Related HPCs

1) Retired Branch Instructions. Retired branch instructions are the branch instructions (call, jmp, jne, etc.) that have been successfully executed and no longer have any dependencies with other micro-operations in the instruction pool. They are considered "retired" once they are committed as part of the architectural state and leave the retirement unit. In Listing 5, the code snippet is leveraged to alter the number of retired branch instructions. Based on our observation, the number of retired branch instructions stays within a certain range for Spectre-type attacks. In general, the number of retired branch instructions is expected to be low for benign processes that create less workload on the system. However, in benign applications with high workloads, such as multi-thread applications, we observed that many benign applications could generate far more retired branch instructions than the Spectre attacks. Therefore, if an attacker can alter the number of retired branch instructions to increase drastically, it is possible to evade ML-based detection models. We consider the code snippet in Listing 5: `Branch_Inst_HPCs` fuzzing block to manipulate the number of retired branch instructions by inserting them into the attack code.

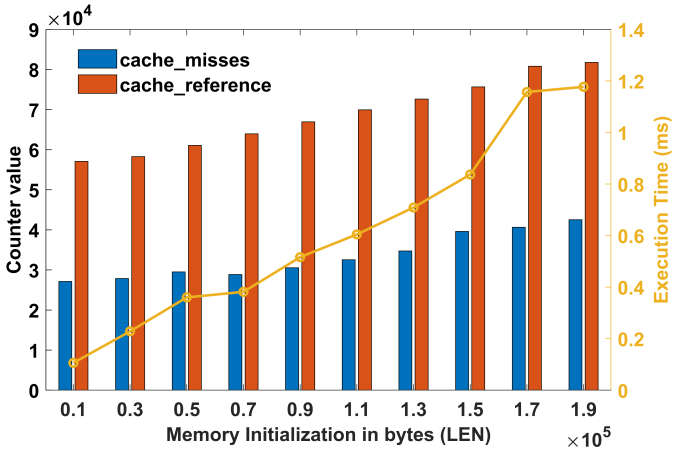

Fig. 2: The comparison of introduced cache misses and cache references events with varying numbers of memory transfers. The yellow line represents the additional time overhead for each parameter.

```
        /* Fuzzing Module: Branch_Inst_HPCs */

int retired_branches = 0;
asm volatile (
  "movl $0, %%eax\n\t" // Initialize counter to 0
  "movl ${initial_value}, %%ecx\\n" // Set loop counter
  "1:\n\t"  // Label 1 for the loop start
  "incl %%eax\n\t"  // Increment the EAX register

  // Introduce multiple branch instructions
  "testl %%eax, %%eax\n\t" // Test EAX (logical compare)
  "jz 2f\n\t" // Jump to label 2 if zero
  "jmp 3f\n\t" // Jump to label 3 unconditionally
  "2:\n\t"  // Label 2 (Always skipped)
  "movl %%eax, %%eax\n\t"
  "jmp 4f\n\t"
  "3:\n\t"  // Label 3
  "movl %%eax, %%eax\\n\\t"
  "4:\n\t"  // Label 4

  "decl %%ecx\n\t"  // Decrement loop counter
  "jne 1b\n\t"   // Jump to Label 1 if not zero
  : "=a" (retired_branches) // Output: final value of EAX (not used)
  :   // No input
  : "%ecx" // Clobbered register
```

```
26  );
```

Listing 5: Fuzzing code targeting retired number of branch instruction

This fuzzing block introduces several branch instructions to impact the number of retired branch instructions. This block takes `initial_value` as an input to set the loop counter, `ECX`. This input is automatically set by the fuzzing framework to manipulate the branch-related HPCs in a controlled way. A loop is initiated at line 7 ("1:"), in which the `EAX` register is incremented. There are two consecutive branch instructions followed by a `testl` operation. The first one (Line 12) is a conditional jump that could jump to Label 2 if the previous `testl` instruction sets the zero flag. This jump instruction is never taken as `EAX` is always non-zero due to the preceding increment. The second branch instruction is an unconditional jump, which is always taken and could jump to Label 3. At Line 21, the loop counter `ECX` is decremented, and a conditional branch instruction could make a backward jump to start from label 1 until the `ECX` sets to zero. Therefore, this fuzzing block controls the number of backward jumps taken by the module by varying the `initial_value` parameter, leading to a controlled manipulation of the retired branch counter.

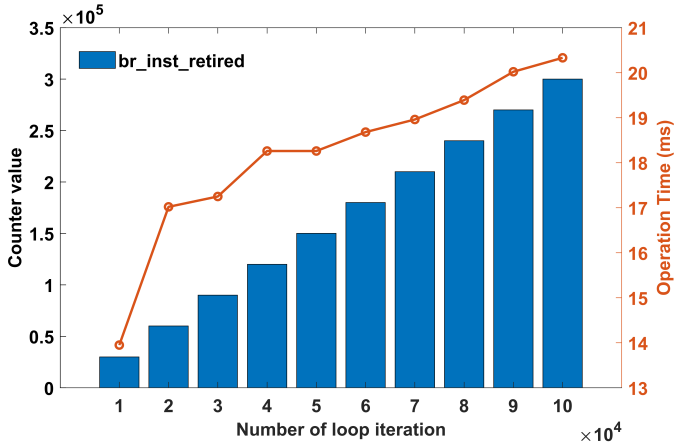

Fig. 3: The comparison of introduced branch instructions retired events with varying numbers of loop iterations. The orange line represents the additional time overhead for each number of iterations.

Evaluation. The number of retired branch instructions can be increased faster than the cache-related events because the fuzzing block does not spend a large amount of time executing branch instructions compared to memory transfers, as illustrated in Figure 3. Hence, with 10,000 iterations, the number of branch instructions rises up to 4×10^4 in around 14 ms. This number can gradually increase up to 3.3×10^5 when the number of iterations reaches 10,000. The iteration number is the parameterized input to this fuzzing block to control the counter values in the sampling interval (100ms for the tested detection tools).

2) Retired Mispredicted Branch Instruction. We alter the number of retired branch instructions in the previous code snippet to introduce additional mispredicted branch instructions. We modify Listing 5 by creating two alternate paths that change randomly based on an input. The frequent and random changes in the conditional branch instructions make it more challenging to guess the correct path for the processor branch prediction unit as given in Listing 6). The number of mispredictions is controlled by the number of loops. However, when the branch prediction unit obtains a longer history, it is expected that increasing the number of iterations will have less effect on the mispredicted branch counter as the number of mispredictions will not increase with the same amount. This is why we expect a non-linear behavior in the increase of the counter compared to the retired branch instructions. The number of newly introduced mispredicted branch instructions are still expected to manipulate the counters sufficiently to evade the detection tools.

```
1  int mispredict = 0;
2
3  // Assembly block to create frequent branch
     mispredictions
4  asm volatile (
5    "movl $0, %%eax\n\t" // Initialize counter to 0
6    "movl ${initial_value}, %%ecx\n" // Set loop
       counter
7    "1:\n\t"    // Label for the loop start
8    "cmpl $0, %%eax\n\t"  // Compare counter with 0
9    "je 2f\n\t"   // Jump if equal to label 2
10   "jmp 3f\n\t"  // Jump to label 3
11   "2:\n\t"      // Label 2
12   "movl $1, %%eax\n\t" // Set counter to 1 to
       alternate path
13   "jmp 4f\n\t"   // Jump to label 4
14   "3:\n\t"       // Label 3
15   "movl $0, %%eax\n\t" // Set counter to 0 to
       alternate path
16   "4:\n\t"    // Label 4
17   "decl %%ecx\n\t"  // Decrement loop counter
18   "jne 1b\n\t"  // Jump to start of loop if not zero
19   : "=a" (mispredict) // Output: final value of EAX
       (not used)
20   :    // No input
21   : "%ecx" // Clobbered register
22 );
```

Listing 6: Fuzzing code targeting retired number of mispredicted branch instruction

Evaluation. Increasing the number of mispredictions is a more challenging task since modern branch prediction units have better capabilities to learn complex branch patterns. Hence, the number of introduced branch instructions in a given time frame is relatively lower than other counters, as given in Figure 4. More importantly, the counter values are not deterministic since the prediction unit makes random guesses, affecting the counter values. Our fuzzing block can generate up to 10×10^4 mispredicted branch events in 23 ms. Note that the time overhead introduced by this block is comparably higher than other counter-manipulation techniques due to back-to-back conditional branch instructions. The number of loops for the branch instructions clearly affects the counter value, which can manipulate counter values randomly. Since the purpose of our fuzzing tool is to introduce random noise to the attack codes, this fuzzing block is still useful for evading the detection tools as detailed in Section VII.

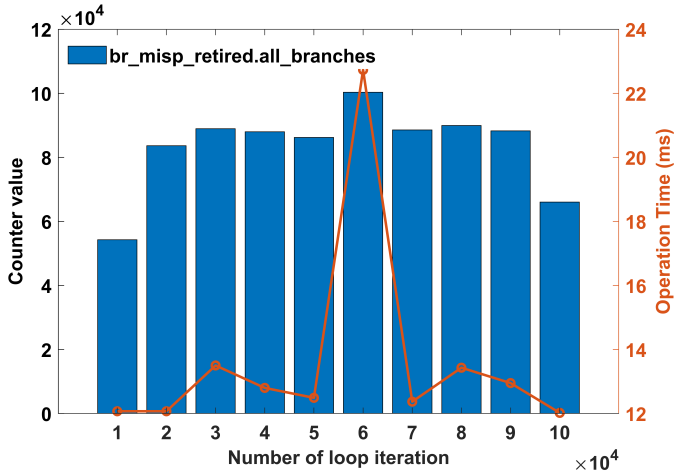

Fig. 4: The comparison of introduced mispredicted branch instructions events with varying numbers of loop iterations. The orange line represents the additional time overhead for each number of iterations.

C. Fuzzing Total Instructions Related HPCs

The number of instructions can be impacted by slowing down the attack code. In this fuzzing block, we insert a `sleep` instruction as a fuzzing snippet into the attack code. However, the sleep period requires to be adjusted in a controlled manner as it can introduce a high time overhead during the attack execution, even leading to lower leakage rate. As an example, for Spectre variant 1, adding an additional 75 μs sleep amount in each attack iteration reduces the leakage accuracy significantly. Therefore, the proposed fuzzing tool iterates over different sleep times and chooses the most appropriate value that ensures both the efficacy of the fuzzing tool and the attack success rate.

D. Randomizing the Attack Code Over Multiple Encryptions

In cache-based attacks, e.g., Flush+Flush, Flush+Reload, and Prime+Probe, the attacker runs the attack over thousands of AES encryptions or RSA decryptions to leak the correct cryptographic key. In the case of AES key recovery attacks, the same attack rounds are repeated, and the attack execution leaves a unique signature for detection models to detect ongoing attacks. However, we observed that the number of encryptions considered by the detection models is extremely high to distinguish attack executions. It is actually possible to maintain a similar leakage rate with a comparatively smaller number of encryptions. Therefore, our fuzzing block randomizes the number of actual attack rounds over the thousands of benign encryption to confuse the detection model. In other words, the attacker code does not perform the attack at each encryption. If the attack code iterates over N number of encryption, our fuzzing tool will restrict the attack code to perform cache profiling M times where $M < N$. Thus, ideally, the attack code will generate attack signatures only over M encryption, while for the rest $(N - M)$ number of encryptions, it will simply generate benign features. As most of the detection models consider a length of prediction window to confirm an attack scenario, this technique works incredibly well to evade the detection as the attack code creates mixed HPCs from both benign and attack features. This fuzzing block is extremely useful for the detection techniques that solely profile the running applications (process-specific) as there is no system noise. Hence, randomizing the number of attack rounds over a fixed number of encryptions can change the counter values significantly.

VI. Reproduced ML-based Detection Tools

HPCs are state-of-the-art sensors to design attack detection tools to identify various types of micro-architectural attacks. Although there are various HPC-based detection models in the literature, most of them are not open-source and it is challenging to reproduce all of them due to the unavailability of the source code and outdated or unsupported HPC events in newer processors. Among different detection models, we selected four detection tools to evaluate the efficacy of our fuzzing tool in evading the detection. In the detection tool selection process, we gave priority to the diversity of implemented ML models as well as the utilized performance counters. In total, we test nine ML models and six performance counters. These variations in detection models create a more challenging scenario for our fuzzing tool, leading to a better understanding of its capabilities.

In Table I, we outline the detection tools that have been reproduced to check the viability of the proposed fuzzing tool. Our reproduced detection tools only differ in terms of the benign applications used in the training process compared to the original work. As any chosen benign application simply acts as a subset of the world-wide benign applications, we adopt our own set of applications with wide variation and keep them consistent for all the reproduced models. For our experiment, we selected the Phoronix test-suite framework as our base, which supports more than 200 test applications. This framework is chosen because the test applications mimic real-world applications by stressing out the processor, memory, or the entire system. The variation of their workload, as well as their impact on different hardware features, make them a more realistic and practical choice to train the models. In the literature, most of the time, the considered benign applications generate lower counter values compared to an attack process. However, this is not the case for all benign applications. While testing with the Phoronix-test-suite framework, we noticed that the counter values might increase beyond the attack process for some benchmark applications. Therefore, a single threshold-based prediction model is not suitable in scenarios where the victim runs high-stress workloads.

The first reproduced detection tool (Tool 1) was proposed by C. Li et al. [24], where the authors utilize the system-wide HPCs to detect Spectre variant 1. The authors used `perf` framework to collect four hardware events with 100 ms resolution and trained three separate machine learning models with Support Vector Machine (SVM), Multilayer Perceptron (MLP), and K-Nearest Neighbor (KNN) algorithms. The performance of the reproduced model in our test setup is listed in Table II. Both SVM and KNN-based detection model

TABLE I: Reproduced ML-based detection tools to evaluate the efficiency of our fuzzing framework. (Note: s.w.=system-wide, p.s.=process-specific)

Detection Tool	HPC framework	Hardware Events	Resolution	ML/DL Model	Targeted Attack
C. Li et al. [25]	perf (s.w.)	cache-misses cache-references br_inst_retired.all_branches br_misp_retired.all_branches	100 ms	SVM MLP KNN	Spectre v1
J. Depoix et al. [11]	papi (p.s.)	L3 cache misses (L3_TCM) L3 cache accesses (L3_TCA) Total number of instructions (TOT_INS)	100 ms	NN	Spectre v1,v2
Mushtaq et al. [30]	PAPI (s.w.)	Total-CPU-Cycles (TOT_CPU_CYC) L1-Data-Cache-Misses (L1_DCM) L3 Total Cache Misses (L3_TCM) L3 Total Cache Accesses (L3_TCA)	10-100 encryption rounds 100 ms for Spectre/Meltdown	DT RF SVM	P+P F+R F+F Spectre Meltdown
B. A. Ahmad et al. [3]	papi, perf (s.w., p.s.)	L3 cache misses (L3_TCM) L3 cache accesses (L3_TCA) Total number of instructions (TOT_INS) Total branch instruction (BR_INS) Branch Miss-Predictions (BR_MSP)	100 ms	LDA LR SVM CNN	Spectre v1, v2 Meltdown

TABLE II: Performance of the reproduced models.

Tool #	ML/DL Model	Detected Attack	Obtained Acc. (%)
Tool 1 [25]	SVM/KNN/MLP	Spectre v1	99.8/99.8/83.5
Tool 2 [11]	NN	Spectre v1	97.5
		Spectre v2	93.5
Tool 3 [3]	DT/RF/SVM	F+R	99.9/100/99.2
Tools 4 [30]	SVM/CNN/LDA/LR	Spectre v1	98.7/97.9/79.9/79.2
		Spectre v2	98.4/98.3/81.5/83.3

provides 99.8% accuracy, however, the MLP-based model can only detect the attack with 83.5% accuracy. As the MLP-based model does not perform well in our setup, this model is excluded from the fuzzing framework evaluation experiments.

The second detection tool (Tool 2) is reproduced from the work of J. Depoix et al. [11]. The `PAPI` framework is used to record three hardware events with 100ms resolution. Tool 2 targets first two variants of Spectre attack. Therefore, two separate Neural Network-based detection model is trained for Spectre v1 and Spectre v2. In other words, Tool 2 comprises of two different detection tools for targeting each specific attack. The accuracy of Tool 2 for detecting Spectre v1 and Spectre v2 are 95.9% and 93%, respectively.

Tool 3 is derived from [30], where the authors target five different attacks (Table I). In our reproduced model, we only target Flush+Reload (F+R) attack out of all the other cache-based attacks. The data is collected after every 100 encryption rounds using the `PAPI` framework. We expect that if the fuzzing tool works with F+R, it is expected to be viable with both Flush+Flush (F+F) and Prime+Probe (P+P) attacks. For this attack, three separate ML-based models are trained that are built with Decision tree (DT), Random Forest (RF), and SVM, respectively. The accuracy of Tool 3 is always more than 99% for all cases including different attacks and ML models, aligning with the results presented in [30].

The last tool is reproduced from [3], where the authors recorded five hardware events to create the detection tools for Spectre and Meltdown attacks. As mentioned earlier, we exclude Meltdown attack as it is no longer effective in the current test setup. Therefore, Tool 4 comprises of two detection tools for Spectre v1 and Spectre v2. Four ML-based models are created for each attack types that employs Linear Discriminant Analysis (LDA), Logistic Regression (LR), SVM, and Convolutional Neural Network (CNN) algorithms. Although LDA and LR-based models do not perform well in our setup, both CNN and SVM-based model achieve around 98% accuracy. In Section VII, we evaluate the performance of our proposed fuzzing tool against the two best performing models.

VII. Evaluation

The performance of the proposed fuzzing tool is evaluated with four detection models (Table II) that have high diversity in terms of HPC events and machine learning algorithms. As mentioned in Section V, the fuzzing tool supports five fuzzing blocks capable of altering different HPC events. The fuzzing tool will randomly choose one of the five modules and insert it into a target attack code. Each fuzzing block has one parameter to control the degree of HPC-specific manipulation during the attack code execution. The fuzzing tool first selects different values for the parameter of a randomly selected module to evade the detection tool. If the parameter's chosen values are insufficient to fool the detection tool, it will select another module and repeat the parameter selection process. The fuzzing tool may go through multiple iterations to find the appropriate parameter to evade the detection tool successfully.

In this section, the performance of the fuzzing tool is evaluated separately against each detection tool in terms of fuzzing time, number of attempts, and preservation of the original leakage rate. We also evaluate how each proposed fuzzing block impacts the HPCs while inserted into the attack code to give a better insight. In Table III, all the proposed fuzzing blocks are listed with the associated parameters that can control the manipulation of specific counter values. For every block, the fuzzing tool iterates the parameter over a pre-determined range (Start-Stop range) with a fixed step value to find the best parameter value for the fuzzing module.

TABLE III: The list of created fuzzing blocks for the proposed fuzzing technique.

Module's Name	Variable	Start-Stop range	Step value
Cache_HPCs	Memory Initialization Length (LEN)	10000-200000 B	10000 B
Branch_Misp_HPCs	Loop Counter	30000-50000	1000
Branch_Inst_HPCs	Loop Counter	30000-50000	1000
Total_Inst_HPCs	Sleep time	10-70 us	10 us
Randomize_Attack	# attack per 100 encryption	100-10	-10

TABLE IV: Performance evaluation of the Fuzzing tool against Detection Tool 1

Detection Tool 1					
Run Time	**Selected Module**	**Tuned Param.**	**# of Attempt**	**Time Spent**	**Time Overhead**
#1	Branch_Misp_HPCs	31000	2	20.9s	×1.6
#2	Cache_HPCs	70000	7	71.3s	×2.1
#3	Branch_Inst_HPCs	24000	5	46.9s	×1.3
#4	Branch_Misp_HPCs	31000	4	48.5s	×1.6
#5	Branch_Inst_HPCs	24000	5	46.0s	×1.3
Avg.				**46.7s**	×**1.6**

Detection Tool 1. As mentioned in Section VI, Detection Tool 1 is trained with four hardware features to detect Spectre v1. The four HPC events are cache misses, cache references, branch instruction retired, and branch misprediction retired. The fuzzing tool randomly chooses one of the fuzzing blocks outlined in Table III to alter the counter values of the HPC events. This approach is more suitable in a scenario where the attacker is not aware of the HPC events beforehand, based on which the machine learning model is trained. The fuzzing tool is tested five times to fuzz the Spectre v1 code, and the modified attack code can successfully evade the detection tool in each run. It is to be noted that Detection Tool 1 comprises three ML-based models. The outcome is only considered a success if the fuzzed attack code can evade all the models.

The results of the five run-times against Detection Tool 1 are presented in Table IV. We observed that the fuzzing framework selects either Branch_Misp_HPCs, Branch_Inst_HPCs, or Cache_HPCs as the final fuzzing block to evade the detection tool in different runs. For Run-Time 1, the fuzzing tool can evade the detection tool in 20.9 seconds with only 2 attempts. The Branch_Misp_HPCs block can successfully modify the attack code to evade the detection tool with an additional 31000 conditional loops while the leakage rate for the Spectre v1 attack remains the same. The attack code only slows down 1.6 times compared to the baseline attack execution. The number of attempts rises to 7 for Run-Time 2 when the Cache_HPCs block is used for fuzzing. This indicates that the fuzzing tool selected other fuzzing blocks that failed to evade the detection tool in the first iterations. In other words, it takes 7 attempts to find the appropriate module with the correct parameter to evade the detection tool finally. This fuzzing runs for 71.3 seconds to create the attack code, transfer it to the victim machine, execute the code, and return the results. This process is repeated 7 times to obtain the evasive attack code. The performance overhead of the evasive attack code is 2.1 times. As the order of the fuzzing blocks testing for the fuzzing tool is random, we calculated the average time over five different run-times to provide an estimation of the entire fuzzing process, which is 46.7 seconds. Also, the average performance overhead is 1.6 times for the evasive attack codes. Our results show that it takes less than a minute to create an evasive attack code with the fuzzing framework for Detection Tool 1.

Detection Tool 2. Detection Tool 2 comprises two NN-based models that can detect Spectre v1 and Spectre v2. The models are trained with three HPC features, namely, L3 cache misses, L3 cache accesses, and the total number of retired instructions. We tested our fuzzing tool with this detection tool, and the results on time overhead and the time spent on fuzzing process are described in Table V. For Spectre v1, the Branch_Misp_HPCs module cannot evade the detection tool with the generalized setup of our fuzzing tool. The rest of the modules are capable of evading the detection, and the fuzzing tool requires comparatively less number of attempts to successfully create an evasive attack code. The highest number of attempts among the five run-times is 22, which indicates that the fuzzing tool first started with the Branch_Misp_HPCs block. It tries 20 different values for the loop counter parameter to evade the detection. In the end, it switches the fuzzing block to Cache_HPCs, and after two additional attempts, the attack finally bypasses the detection. The average time to create the modified fuzzed version of the attack code is 52.2 seconds calculated over five run-times. The inserted modules incur an average time overhead of 1.2 times compared to the execution time of the original attack code.

In Table V, it is shown that all of our fuzzing blocks are capable of evading the detection by modifying the Spectre v2 attack code. Interestingly, the fuzzing tool does not even have to go through multiple iterations to try different values for each parameter in each fuzzing block to avoid detection. However, it also indicates that our initial selection of values for the parameters might be relatively higher than it actually requires to avoid the detection for this detection tool. For example, the fuzzing tool starts by initializing the memory length from 10000 memory operations for the Cache_HPCs block (Table III), which incurs around 1.5 times time-overhead. Although the fuzzing tool can bypass the detection tool with just one attempt, the initial parameter value may not be the most optimum value in terms of the overhead. This result shows that there is a trade-off between speed and time overhead. The range of the input variables and the step values for each of these fuzzing blocks can be set by the attacker through a configuration file before implementing the fuzzing tool. Thus, the attacker will have the option to choose the best trade-off options based on specific scenarios. As we mentioned earlier, all the modules can evade the detection for Spectre v2 with just one attempt; hence, the average time to find the appropriate fuzzing takes only 7.98 sec with an average time-overhead of 1.4 times compared to the baseline attack code.

Detection Tool 3. Detection Tool 3 targets Flush+Reload attacks, and the tool is trained with cache-related HPCs.

TABLE V: Performance evaluation of the Fuzzing tool against Detection Tool 2

Run Time	Selected Module	Tuned Param.	# of Attempt	Time Spent	Time Overhead
Detection Tool 2 (Spectre v1)					
#1	Cache_HPCs	20000	2	16.7s	×1.3
#2	Total_Inst_HPCs	20	2	20.5s	×1.04
#3	Branch_Inst_HPCs	30000	1	8.8s	×1.3
#4	Cache_HPCs	20000	22	205.9s	×1.3
#5	Branch_Inst_HPCs	30000	1	9.2s	×1.3
Avg.				**52.2s**	×**1.2**
Detection Tool 2 (Spectre v2)					
#1	Branch_Inst_HPCs	30000	1	7.8s	×1.6
#2	Cache_HPCs	10000	1	7.5s	×1.5
#3	Total_Inst_HPCs	10	1	8.8s	×1.07
#4	Cache_HPCs	10000	1	7.4s	×1.1
#5	Branch_Misp_HPCs	30000	1	8.4s	×1.8
Avg.				**7.98s**	×**1.4**

TABLE VI: Performance evaluation of the Fuzzing tool against Detection Tool 4

Run Time	Selected Module	Tuned Param.	# of Attempt	Time Spent	Time Overhead (%)
Detection Tool 4 (Spectre v1)					
#1	Cache_HPCs	20000	22	191.0s	×1.3
#2	Total_Inst_HPCs	50	5	50.9s	×1.1
#3	Branch_Inst_HPCs	30000	21	183.5s	×1.3
#4	Cache_HPCs	20000	2	14.6s	×1.3
#5	Branch_Inst_HPCs	30000	1	7.8s	×1.3
Avg.				**89.6s**	×**1.3**
Detection Tool 4 (Spectre v2)					
#1	Branch_Misp_HPCs	30000	1	8.0s	×1.8
#2	Cache_HPCs	20000	2	13.7s	×1.5
#3	Branch_Inst_HPCCs	30000	1	7.2s	×1.5
#4	Total_Inst_HPCs	20	2	13.8s	×1.1
#5	Cache_HPCs	20000	2	13.8s	×1.5
Avg.				**11.3s**	×**1.5**

Since the Spectre-type attacks are not in the scope, branch-related HPCs are not included in the counter list. More importantly, the detection model leverages process-specific monitoring, leading to less noisy measurements compared to other detection tools. This tool only captures the counter values every 100 encryption rounds. We noticed that benign workloads have considerably less number of cache misses and the cache-related fuzzing blocks always increase the counter values. Hence, increasing the counter values is not a viable option to bypass the detection tool. The only module that works against Flush+Reload is the Randomize_attack fuzzing block as explained in Section V. The fuzzing block controls the number of attacks per 100 encryption while maintaining the leakage percentage as 100%. The fuzzing tool is capable of evading the detection with 4 attempts by restricting the number of attacks up to 70 out of 100 per encryption. Since the monitoring time for the attack code is longer than the other detection tools, the fuzzing process takes around 665 sec to complete with a 100% leakage rate. The performance overhead is only 1.4 times compared to the baseline attack code.

Detection Tool 4. Detection Tool 4 supports two separate models targeting Spectre v1 and Spectre v2. Each of these models is trained using four ML-based algorithms. The fuzzing tool is tested to avoid all the models of Detection Tool 4. Except for Total_Inst_HPCs, all the other fuzzing modules are capable of fuzzing the attack codes in an evasive manner. The performance evaluation of the proposed tool is demonstrated in Table VI. The average time of the fuzzing tool to successfully alter the Spectre v1 code is 89.6 seconds without losing any leakage rate. For Cache_HPCs module, the fuzzing tool has to increase the memory initialization length up to 20000 bytes. The first run takes a considerably longer time to find the evasive attack since it chose the inappropriate parameters and fuzzing blocks in its initial tries. On the contrary, if the fuzzing tool starts with Branch_Inst_HPCs block, it can find the evasive attack code with only 1 attempt. However, the most efficient fuzzing block is based on Total_Inst_HPCs block as it only introduces 10% performance overhead for the attack code. Our results show that we can bypass all the trained models for Spectre-v1 in Detection Tool 4 with three fuzzing blocks in an average of 89.6 seconds of fuzzing run. The average performance overhead is only 1.3 times compared to the baseline attack code.

For Spectre v2, we observed that the same fuzzing blocks are capable of evading the detection. The branch-related blocks almost act in a similar manner as we have seen for Spectre v1. The Cache_HPCs module can evade the detection with 20,000 memory operations inserted into the attack code. The average time to complete the fuzzing is 11.3 seconds as the the number of attempts is comparatively less than Spectre-v1 scenario. Interestingly, the average leakage rate of Spectre v2 without fuzzing is 80.6% in our test setup. After the fuzzing, the computed average leakage rate over the five run-times is 78.8%, which is comparable with the original leakage rate. Our fuzzing blocks introduce 1.5 times performance overhead on average to successfully evade the detection. It is important to note that the same parameters for both branch instructions and cache operations are required for Spectre-v1 and Spectre-v2 attacks. We also show that decreasing the total number of instructions slightly can reduce the efficiency of Detection Tool 4 significantly while incurring a small amount of overhead to the attack code.

VIII. Related Work

A. ML-based Microarchitecture Attack Detection Techniques

While there are several microarchitecture attack detection tools, the early ones mostly relied on statistical measures such as threshold decisions [16], dynamic time warping [39], and sudden changes in the counter values [6]. However, the decisions made by these tools can be bypassed with simple changes in the counters since there is no advanced learning process. Hence, many detection tools started to leverage ML models to learn the complex behavior of benign and attack executions. One of the first ML-based detection tools [8] is based on neural networks to distinguish benign and attack executions. Later, other tools leveraged ML models such as SVM [3], [25], DT [29], kNN [25], CNN [3], and LSTMs [18].

Each detection tool relies on the capabilities of ML models in learning complex behaviors, which has advanced the detection accuracy and diversity of targeted attack implementations. Both supervised [28] and unsupervised [6], [8], [14], [18] learning techniques have been utilized to distinguish benign and attack workloads. In the unsupervised learning approach, tools train the model with benign execution, and attack executions are treated as anomalies. This approach is more vulnerable to evasion attacks because attack codes hidden in benign applications can stress the counter values slightly, behaving like a benign application [22]. Hence, supervised detection models are more robust against evasion attacks as long as the benign dataset is sufficiently diverse.

B. Evasion against ML-based Detectors

ML-based models are used in many security-critical fields to detect ongoing attack attempts. The most popular use case for ML models is the malware detection [5], [38]. However, since the first example of adversarial attacks [32] on ML models, ML-based malware detectors have become a target for adversarial attacks [7], [10], [12]. Similarly, individual ML-based microarchitectural attack detection tools are also evaluated against different variants of the same attack by changing the leakage rate [23], [34], [37]. However, these studies only evaluate their own models against evasive attacks. Song et al. [35] create specially crafted code snippets to bypass the profiling-based detection systems. Their technique is only tested on cache-based attacks with a limited number of profiling detection tools. A recent study [22] evaluated the microarchitecture detection models with different criteria such as accuracy, detection speed, overhead, and threat models. The paper also creates evasive attack codes manually by placing attack primitives into benign applications. Due to the lack of open-source detection models, this study only considers two models to evaluate their effectiveness against evasive attacks. On the other hand, our study creates an automated fuzzing framework to evaluate multiple models that are not open-source.

IX. Discussion and Limitations

Benign Dataset Selection. The benign dataset selection is important for creating a robust detection tool. When the tool is trained with a diverse set of benign applications, the detection tool outputs less number of false positives and negatives. However, we noticed that most of the studies choose workloads with single-threaded and less memory operations. Hence, the counter values remain lower than a full-speed attack execution, making it easier to distinguish benign and attack executions and achieving success rates close to 100%. There are multi-threaded workloads that utilize a high number of CPU cores, leading to even higher counter values compared to attack executions. We believe this type of benign dataset is more realistic to evaluate the effectiveness of the detection models.

Limitations on Counter Manipulation. Depending on the counter-sampling resolution, the fuzzing blocks have a limited amount of time to manipulate the counter values. Especially, the upper bound relies on the sampling resolution. However, changing a counter value could also affect other counters as well. As an example, the cache references counter is affected by cache miss counters because any request to cache is counted as an event in the cache references counter. Hence, manipulating one counter could also change other counters in parallel. This makes it difficult to isolate different counters from each other if the detection model monitors related counters.

Transferability of the Evasive Attack Codes. It is still not clear whether an evasive attack code can bypass other models as well. This phenomenon still remains unanswered when multiple detection models are trained with different counters, learning algorithms, resolutions, and threat scenarios. However, many detection tools have no public repositories, which makes it more challenging to evaluate their effectiveness against evasive attacks. We also leave the transferability of evasive attacks between different microarchitectures as a potential future work.

X. Conclusion

Our study develops an automated fuzzing-based evasive attack code generation tool. The fuzzing tool is evaluated against four ML-based detection methods to show its effectiveness in generating evasive attack code snippets while preserving the leakage rate. Our fuzzing tool incurs a small amount of time overhead on the attack code to successfully bypass the detection tools. In our threat model, we assume no knowledge of the ML model used in the detection tool, which makes the fuzzing tool more applicable in black-box scenarios. We conclude that ML-based detection tools are susceptible to evasive attacks that can be created with minimal manual attack code modification. The system security community is encouraged to explore more robust training methods to create more comprehensive detection tools.

References

[1] Intel performance counter monitor. https://github.com/intel/pcm.

[2] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *Topics in Cryptology–CT-RSA 2007: The Cryptographers' Track at the RSA Conference 2007, San Francisco, CA, USA, February 5-9, 2007. Proceedings*, pages 225–242. Springer, 2006.

[3] Bilal Ali Ahmad. Real time detection of spectre and meltdown attacks using machine learning. *arXiv preprint arXiv:2006.01442*, 2020.

[4] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 870–887. IEEE, 2019.

[5] A Ananya, A Aswathy, TR Amal, PG Swathy, P Vinod, and Shojafar Mohammad. Sysdroid: a dynamic ml-based android malware analyzer using system call traces. *Cluster Computing*, 23(4):2789–2808, 2020.

[6] Samira Briongos, Gorka Irazoqui, Pedro Malagón, and Thomas Eisenbarth. Cacheshield: Detecting cache attacks through self-observation. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, pages 224–235, 2018.

[7] Fabrício Ceschin, Marcus Botacin, Heitor Murilo Gomes, Luiz S Oliveira, and André Grégio. Shallow security: On the creation of adversarial variants to evade machine learning-based malware detectors. In *Proceedings of the 3rd Reversing and Offensive-oriented Trends Symposium*, pages 1–9, 2019.

[8] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. Real time detection of cache-based side-channel attacks using hardware performance counters. *Applied Soft Computing*, 49:1162–1174, 2016.

[9] Arnaldo Carvalho De Melo. The new linux'perf'tools. In *Slides from Linux Kongress*, volume 18, pages 1–42, 2010.

[10] Luca Demetrio, Scott E Coull, Battista Biggio, Giovanni Lagorio, Alessandro Armando, and Fabio Roli. Adversarial exemples: A survey and experimental evaluation of practical attacks on machine learning for windows malware detection. *ACM Transactions on Privacy and Security (TOPS)*, 24(4):1–31, 2021.

[11] Jonas Depoix and Philipp Altmeyer. Detecting spectre attacks by identifying cache side-channel attacks using machine learning. *Advanced Microkernel Operating Systems*, 75:48, 2018.

[12] Sai Manoj Pudukotai Dinakarrao, Sairaj Amberkar, Sahil Bhat, Abhijitt Dhavlle, Hossein Sayadi, Avesta Sasan, Houman Homayoun, and Setareh Rafatirad. Adversarial attack on microarchitectural events based malware detectors. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–6, 2019.

[13] Debopriya Roy Dipta and Berk Gulmezoglu. Df-sca: dynamic frequency side channel attacks are practical. In *Proceedings of the 38th Annual Computer Security Applications Conference*, pages 841–853, 2022.

[14] Debopriya Roy Dipta and Berk Gulmezoglu. Mad-en: Microarchitectural attack detection through system-wide energy consumption. *IEEE Transactions on Information Forensics and Security*, 2023.

[15] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with {TLB} attacks. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 955–972, 2018.

[16] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+ flush: a fast and stealthy cache attack. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings 13*, pages 279–299. Springer, 2016.

[17] Berk Gülmezoğlu, Mehmet Sinan Inci, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. A faster and more realistic flush+ reload attack on aes. In *Constructive Side-Channel Analysis and Secure Design: 6th International Workshop, COSADE 2015, Berlin, Germany, April 13-14, 2015. Revised Selected Papers 6*, pages 111–126. Springer, 2015.

[18] Berk Gulmezoglu, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. Fortuneteller: Predicting microarchitectural attacks via unsupervised deep learning. *arXiv preprint arXiv:1907.03651*, 2019.

[19] Berk Gulmezoglu, Andreas Zankl, Thomas Eisenbarth, and Berk Sunar. Perfweb: How to violate web privacy with hardware performance events. In *Computer Security–ESORICS 2017: 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part II 22*, pages 80–97. Springer, 2017.

[20] Mehmet Sinan Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cache attacks enable bulk key recovery on the cloud. In *Cryptographic Hardware and Embedded Systems–CHES 2016: 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings 18*, pages 368–388. Springer, 2016.

[21] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. *Communications of the ACM*, 63(7):93–101, 2020.

[22] William Kosasih, Yusi Feng, Chitchanok Chuengsatiansup, Yuval Yarom, and Ziyuan Zhu. Sok: Can we really detect cache side-channel attacks by monitoring performance counters?

[23] Yusuf Kulah, Berkay Dincer, Cemal Yilmaz, and Erkay Savas. Spydetector: An approach for detecting side-channel attacks at runtime. *International Journal of Information Security*, 18:393–422, 2019.

[24] Congmiao Li and Jean-Luc Gaudiot. Online detection of spectre attacks using microarchitectural traces from performance counters. In *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 25–28. IEEE, 2018.

[25] Congmiao Li and Jean-Luc Gaudiot. Detecting spectre attacks using hardware performance counters. *IEEE Transactions on Computers*, 71(6):1320–1331, 2021.

[26] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse engineering intel last-level cache complex addressing using performance counters. In *Research in Attacks, Intrusions, and Defenses: 18th International Symposium, RAID 2015, Kyoto, Japan, November 2-4, 2015. Proceedings 18*, pages 48–65. Springer, 2015.

[27] Philip J Mucci, Shirley Browne, Christine Deane, and George Ho. Papi: A portable interface to hardware performance counters. In *Proceedings of the department of defense HPCMP users group conference*, volume 710, 1999.

[28] Maria Mushtaq, Ayaz Akram, Muhammad Khurram Bhatti, Maham Chaudhry, Vianney Lapotre, and Guy Gogniat. Nights-watch: A cache-based side-channel intrusion detector using hardware performance counters. In *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*, pages 1–8, 2018.

[29] Maria Mushtaq, Ayaz Akram, Muhammad Khurram Bhatti, Rao Naveed Bin Rais, Vianney Lapotre, and Guy Gogniat. Run-time detection of prime+ probe side-channel attack on aes encryption algorithm. In *2018 Global Information Infrastructure and Networking Symposium (GIIS)*, pages 1–5. IEEE, 2018.

[30] Maria Mushtaq, Jeremy Bricq, Muhammad Khurram Bhatti, Ayaz Akram, Vianney Lapotre, Guy Gogniat, and Pascal Benoit. Whisper: A tool for run-time detection of side-channel attacks. *IEEE Access*, 8:83871–83900, 2020.

[31] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1406–1418, 2015.

[32] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. Practical black-box attacks against machine learning. In *Proceedings of the 2017 ACM on Asia conference on computer and communications security*, pages 506–519, 2017.

[33] Naman Patel, Prashanth Krishnamurthy, Hussam Amrouch, Jörg Henkel, Michael Shamouilian, Ramesh Karri, and Farshad Khorrami. Towards a new thermal monitoring based framework for embedded cps device security. *IEEE Transactions on Dependable and Secure Computing*, 19(1):524–536, 2020.

[34] Nikolaos Foivos Polychronou, Pierre-Henri Thevenon, Maxime Puys, and Vincent Beroulle. Madman: Detection of software attacks targeting hardware vulnerabilities. In *2021 24th Euromicro Conference on Digital System Design (DSD)*, pages 355–362. IEEE, 2021.

[35] Minkyu Song, Taeweon Suh, and Gunjae Koo. Vizard: Passing over profiling-based detection by manipulating performance counters. *IEEE Access*, 2023.

[36] Zhongkai Tong, Ziyuan Zhu, Zhanpeng Wang, Limin Wang, Yusha Zhang, and Yuxin Liu. Cache side-channel attacks detection based on machine learning. In *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 919–926, 2020.

[37] Wubing Wang, Guoxing Chen, Yueqiang Cheng, Yinqian Zhang, and Zhiqiang Lin. Specularizer: Detecting speculative execution attacks via performance tracing. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 18th International Conference, DIMVA 2021, Virtual Event, July 14–16, 2021, Proceedings 18*, pages 151–172. Springer, 2021.

[38] Jason Zhang. Mlpdf: an effective machine learning based approach for pdf malware detection. *arXiv preprint arXiv:1808.06991*, 2018.

[39] Tianwei Zhang, Yinqian Zhang, and Ruby B Lee. Cloudradar: A real-time side-channel attack detection system in clouds. In *Research in Attacks, Intrusions, and Defenses: 19th International Symposium, RAID 2016, Paris, France, September 19-21, 2016, Proceedings 19*, pages 118–140. Springer, 2016.

Debopriya Roy Dipta is currently a third-year PhD student in Computer Engineering at Iowa State University. He completed his B.Sc. degree in Electrical and Electronic Engineering from Khulna University of Engineering and Technology, Bangladesh. His research interests include Micro-architectural Attacks, Side-Channel data analysis, Hardware Security, and Applications of Machine Learning. He is currently working on micro-architectural security with deep learning.

Jonathan Tan is currently a senior in Computer Engineering at Iowa State University and is working towards completing his undergrad and getting his master's in computer engineering. His area of interest is microarchitecture security and hardware accelerator design for Machine Learning. He presented at the National Conference on Undergraduate Research (NCUR) 2023. He also interned at Eaton as a firmware intern.

Berk Gulmezoglu is an Assistant Professor in the Electrical and Computer Engineering Department of Iowa State University. He received his PhD degree from Worcester Polytechnic Institute and his B.S. and M.S. degrees from Ihsan Dogramaci Bilkent University. His research interests include attacks on cryptographic implementations, microarchitectural attacks, Machine Learning applications on hardware security and side-channel attacks in the cloud.