SPECULARIZER: Detecting Speculative Execution Attacks via Performance Tracing

Wubing Wang[1], Guoxing Chen[1], Yueqiang Cheng[3], Yinqian Zhang[2](✉), and Zhiqiang Lin[1]

[1] The Ohio State University, Columbus, OH 43210, USA
wang.11488@osu.edu, chen.4329@osu.edu, zlin@cse.ohio-state.edu
[2] Southern University of Science and Technology, Shenzhen, Guangdong 518055, China
yinqianz@acm.org
[3] NIO Security Research, San Jose, CA 95134, USA
yueqiang.cheng@nio.io

Abstract. This paper presents SPECULARIZER, a framework for uncovering speculative execution attacks using performance tracing features available in commodity processors. It is motivated by the practical difficulty of eradicating such vulnerabilities in the design of CPU hardware and operating systems and the principle of defense-in-depth. The key idea of SPECULARIZER is the use of Hardware Performance Counters and Processor Trace to perform lightweight monitoring of production applications and the use of machine learning techniques for identifying the occurrence of the attacks during offline forensics analysis. Different from prior works that use performance counters to detect side-channel attacks, SPECULARIZER monitors triggers of the critical paths of the speculative execution attacks, thus making the detection mechanisms robust to different choices of side channels used in the attacks. To evaluate SPECULARIZER, we model all known types of exception-based and misprediction-based speculative execution attacks and automatically generate thousands of attack variants. Experimental results show that SPECULARIZER yields superior detection accuracy and the online tracing of SPECULARIZER incur reasonable overhead.

1 Introduction

Speculative execution attacks exploit micro-architectural design flaws and side channels in modern processors and enable unprivileged processes to exfiltrate sensitive information across security boundaries. These attacks have seriously undermined the fundamental security assumptions made in the design of the operating systems and have been in the spotlight since their very first public disclosure in early 2018. The most prominent examples of speculative execution attacks are Meltdown [27] and Spectre [23], and later variants, such as Foreshadow [41], Micro-architectural Data Sampling (MDS) [31,37,43], Load Value Injection (LVI) [42] are also well-known examples of such attacks.

In this paper, we apply the principle of defense-in-depth and propose SPECULARIZER[1], a software framework for uncovering speculative execution attacks using hardware performance tracing features available in commodity processors, *i.e.*, hardware

[1] SPECULARIZER is a portmanteau of "Speculative" and "Polarizer".

Y. Cheng—This work was mainly done at Baidu Research.

performance counters (HPC) and processor trace (PT). SPECULARIZER complements existing defenses against speculative execution attacks, by offering a capability of logging both architectural and micro-architectural behaviors of the monitored software to enable forensic analysis and offline attack detection.

In contrast to prior work that detects cache side channels to identify speculative execution attacks [19], which can be easily circumvented by attacks using alternative side channels, SPECULARIZER is inspired by the following key observations: Although speculative execution attacks may leverage a variety of micro-architectural side channels (*e.g.*, TLBs, caches) to leak secrets from speculatively executed instructions, the invariant of these attacks is the method with which the speculative execution can be triggered. In exception-based attacks, speculative execution is triggered by exceptions, which are either handled or suppressed; in misprediction-based attacks, speculative execution is triggered either by control-flow misprediction or by misprediction in the memory disambiguation. Therefore, SPECULARIZER utilizes the inevitable execution patterns of exceptions and mispredictions as signatures.

We identify PT packets and HPC events that can reveal crucial information necessary for attack detection, such as control-flow transfers for exception handling and TSX aborts, mispredicted branch instructions, machine clears due to memory order conflicts, *etc.* While each type of PT or HPC record alone is insufficient for reconstructing all attack activities, collectively they offer greater insight into the micro-architectural level behavior of the monitored applications. Therefore, we develop techniques to combine HPC and PT data to construct execution traces. With these traces, we build classification models using the Long Short Term Memory (LSTM) network to perform the classification of attack and benign programs.

SPECULARIZER consists of two components: an online trace collection component that is integrated into the operating system of a production machine, on which the monitored application runs, and an offline attack detection component that performs HPC and PT records parsing, trace processing, and trace classification, which are time-consuming and hard to finish in real-time. In fact, rarely do HPC or PT-based monitoring systems perform real-time analysis [14,51]. As such, SPECULARIZER is best suited for VM or container-based cloud systems, where suspicious workloads from untrusted cloud tenants are monitored on cloud servers and forensic analyses are performed on separate servers to detect attack activities. While deferred attack detection does not prevent the attacks from happening, it can trigger further investigation of attacks to identify their sources and assess their consequences.

We have implemented a prototype of SPECULARIZER and evaluated its effectiveness and efficiency in a lab setting. Specifically, to evaluate SPECULARIZER, we develop parameterized models for each type of the speculative execution attacks we aim to detect, and then automatically generate thousands of attack variants by tuning the parameters of these models. With the data sets collected from both benign and attack samples, the evaluation of SPECULARIZER suggests it has promising detection accuracy while inducing reasonable performance overhead. The evaluation results also indicate that SPECULARIZER significantly raises the bar for performing speculative execution attacks even if the attackers understand the detection mechanism.

Contributions. The paper makes the following contributions: ① SPECULARIZER is the software tool that detects speculative execution attacks, by their triggers of speculative execution rather than specific covert channels. ② SPECULARIZER provides new insights of combined use of multiple performance tracing hardware features, *e.g.*, PT and HPCs, in the context of offline attack detection. ③ The paper presents parameterized models of speculative execution attacks and methods to automatically generate attack variants with varying attack success rates. ④ The paper presents a prototype implementation of SPECULARIZER and empirically evaluates its selection of parameters, its effectiveness, and performance overhead.

2 Background

Speculative Execution Attacks. A speculative execution attack contains the following components [3]: *Speculation primitive* triggers speculative execution of instructions. *Disclosure gadget* transmits information through a side channel. *Disclosure primitive* reads the side-channel information that was transmitted by the disclosure gadget. As such, a speculative execution attack can be performed in the following steps: ① executes the *speculation primitive* to trigger speculative execution of instructions. ② utilizes the speculative instructions (including the *speculation primitive* itself) to access secrets across the security boundary; ③ speculatively executes the *disclosure gadget* to encode the secret value into the cache states; ④ uses the *disclosure primitive* to decode the secret data from cache states.

According to the *speculation primitives*, we classify speculative execution attacks into the following three categories [11]. Misprediction-based attacks leverage branch, Store-To-Load (STL), and memory-order buffer mispredictions as the *speculation primitive* and performs attacks before the correct target is resolved. Exception-based attacks and assistance-based attacks use exceptions (*e.g.* Page fault, General Protection fault, *etc.*) and microcode assists (*e.g.* line-fill buffer, store buffer, and load port conflict [1], *etc.*) as *speculation primitive*, respectively, and speculatively execute instructions before they are handled by the processor.

Performance Tracing Hardware. Intel PT is a hardware feature available in Intel processors since Broadwell. It is designed to record the information regarding the control-flow transfers of software programs with very low performance overhead. The PT hardware generates PT packets to reconstruct the timestamped control flow for a program [17,45]. HPCs are a set of model-specific registers that can be used to count user-selected processor architectural or micro-architectural events. Each HPC register can be configured to count a specific event supported by the processor. At runtime, when the specified event happens, the corresponding HPC counter will be incremented.

The HPCs have two different approaches for software to collect event samples. First, when the performance monitor interrupt (PMI) is enabled in a specific counter, a PMI will be triggered when the counter overflows, which provides the software with an opportunity to handle the HPCs data [7]. However, the large volume of interrupts dramatically increases the performance overhead. Second, to address the performance issues, Intel introduces Precise Event-Based Sampling (PEBS), which can store the events in a buffer (dubbed *Debug Store* (DS) area). Only one interrupt is triggered when the buffer is almost full (determined by a threshold).

3 Threat Model and SPECULARIZER Overview

Threat Model. All misprediction-based and exception-based attacks are in-scope of this paper. Our method detects these two types of attacks by monitoring its execution of the *speculation primitives*, which are either a branch instruction that takes time to resolve its target address or a memory load that accesses data across the security boundary. We consider MDS and LVI attacks that are triggered by exceptions, which are the most common cases in current state-of-the-art attack examples, as exception-based attacks, and hence SPECULARIZER will detect those attacks.

SPECULARIZER Architecture. The overall architecture of SPECULARIZER is shown in Fig. 1, which consists of two components: *Online Trace Collection* and *Offline Attack Detection*. *Online Trace Collection* is an online component that runs on a production system, running as system programs, which produces execution traces collected using PT and HPC. *Offline Attack Detection* is a component that runs offline that includes two parts (*i.e. Trace Processing* and *Attack Detection*), possibly on a separate machine, and performs analysis of the collected traces to identify speculative execution attacks.

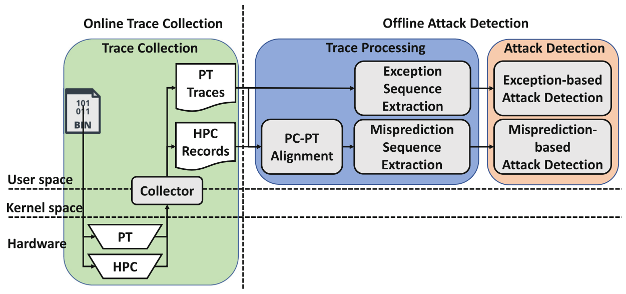

Fig. 1. Architectural of SPECULARIZER.

Online Trace Collection. To detect exception-based speculative execution attacks, SPECULARIZER monitors exceptions using PT. When the attacks use TSX to suppress exceptions, PT packets can record TSX aborts; when the attacks handle the exceptions directly, PT packets can record control-flow transfers that correspond to exception handling.

To detect misprediction-based attacks, SPECULARIZER needs to monitor the pattern of mispredictions, which includes misprediction in control-flow predictors (branch prediction units like BTB, PHT, and RSB) and data-flow predictors (the memory disambiguator in load/store buffers). However, PT is insufficient to monitor these microarchitectural events. HPCs are utilized instead. The limitation of using HPCs to monitor misprediction is that they are asynchronous with execution context, which is insufficient for detecting misprediction-based attacks. To address this problem, SPECULARIZER utilizes Intel PT to provide the execution contexts.

Offline Attack Detection. During the execution of benign programs, exceptions, TSX transaction aborts, and misprediction in control-flow and data loading is normal. There-

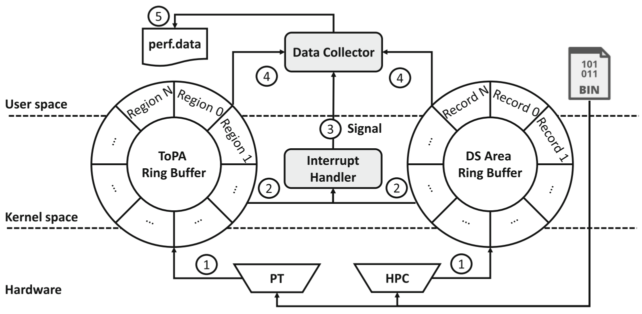

Fig. 2. Architectural and workflow of trace collection.

fore, we cannot simply detect speculative execution attacks using exception handling/-suppressing and branch/data misprediction as signatures. Instead, patterns of exceptions and mispredictions must be learned from both benign and attack programs and utilized to detect attacks in the program to be monitored.

4 Trace Collection

The overall workflow of trace collection (shown in Fig. 2) is as follows: ① SPECULARIZER enables PT and HPC to monitor the execution of the target program and specify the memory buffer to record the execution traces. ② When the memory buffer is full, an interrupt is triggered. ③ After replacing the full buffer with an empty one for the hardware to continue recording data, the interrupt handler sends a signal to the userspace data collector. ④ Upon receiving the signal, the data collector reads data from the full buffer. ⑤ Finally, the data collector saves the collected data into files.

Collecting Traces from HPC. To gain visibility into micro-architectural events, SPECULARIZER activates HPCs to monitor branch mispredictions (*e.g.* direct, indirect branches) and machine clear events caused by *memory order conflicts* by activating the events `BR_MISP_RETIRED.ALL_BRANCHES` and `MACHINE_CLEARS.MEMORY_ORDERING`. SPECULARIZER uses the PEBS to monitor the branch misprediction event and PMI to monitor the memory order conflict event, as the memory order conflict event is not available in PEBS mode. These two events are monitored simultaneously on different HPCs. When monitoring in the PMI mode, the overhead mainly comes from PMI handling. When monitoring in the PEBS mode, although the interrupts are significantly reduced, there are still two sources of overhead: First, writing each PEBS record into the DS area takes about 200 ns [8]. Second, DS-area-overflow interrupts need to be generated when the DS area is full (maximum size is 4 MB). Thousands of interrupts need to be generated during trace collection for one application.

Two performance optimization were implemented: *First*, SPECULARIZER implements a ring buffer [51] to cache the data in the DS area. Specifically, SPECULARIZER allocates two buffers for the DS area to reduce the overhead introduced by dumping data inside the interrupt handler. When the data in one of the buffers reaches the threshold, Specularizer switches the buffer used by the DS area upon receiving the interrupt. A signal is sent to the user-space component of Specularizer to dump data from the full buffer. *Second*, to reduce the number of generated PEBS records, Specularizer tunes the PEBS sampling rate (ρ), which indicates the fraction of events ($1/\rho$) sampled by PEBS to create PEBS records. $\rho > 1$ means PEBS are sampled less frequently with a higher performance overhead and hence some branch mispredicition information is missing. We will evaluate the impact of ρ on detection accuracy in Sect. 8.

Collecting Traces from PT. To collect control-flow transfer and timestamp information, Specularizer activates Intel PT by setting the following control bits of the MSR `IA32_RTIT_CTL`: `TraceEn` (to enable PT), `BranchEn` (to generate control-flow related packets, *e.g.*, TNT, TIP & FUP), `OS & User` (to monitor both user-mode processes kernel threads), `TSCEn, MTCEn & CYCEn` (to generate timestamp related packets, *e.g.* TSC, MTC & CYC).

The overhead incurred in generating PT packets is negligible. The main overhead comes from handling the memory buffer that stores the PT packets when it is full. Unlike PEBS's DS area, which has a fixed size (*i.e.*, 4 MB), the memory buffer used by PT can vary. Specifically, PT uses a Table of Physical Addresses (ToPA) to store all generated packets, which is a linked list that links multiple output regions. Therefore, the total size of the ToPA is flexible, and the number of generated interrupts can be controlled to decrease the runtime performance overhead.

5 Trace Processing

Specularizer processes HPC events and PT packets offline, possibly on a machine that is different from the host that implements the Specularizer monitors. The exception-based output sequences are generated using PT traces only, and the misprediction-based output sequences are extracted with information from both PT traces and HPC records.

5.1 Processing Exceptions

Among the three approaches to tackling exceptions in exception-based speculative execution attacks, namely handling exceptions, suppressing exceptions with TSX, and suppressing exceptions with branch misprediction, the first two cases trigger an indirect control-flow transfer. Therefore, Specularizer extracts exception-triggered control flow transfers in collected PT records. The third case is categorized as misprediction-based and discuss later.

Extracting Addresses of Exceptions. When the exception is handled by exception handlers, the control flow will transfer from user space to kernel space. With the PT packets, we can extract all *kernel traces*—a sequence of instructions in the kernel space. Afterwards, by comparing those traces with the kernel symbol table, the *kernel traces* can be used to identify different types of exceptions.

When the exception is suppressed by TSX transactions, the exception type is not revealed through *kernel traces*. Nevertheless, the exception is recorded by the MODE packet which has a field called TXAbort, with its value as 1. The addresses of the instructions that trigger TXAbort are recorded by the FUP packet that follows.

Extracting Timestamps of Exceptions. PT can be used to recover the timestamp of exceptions, as PT records the following time-related packets: Timestamp Counter (TSC) packets provide the wall-clock time (`wc`); Mini Time Counter (MTC) packets are generated periodically based on the core-crystal clock (`ccc`); a TMA packet is generated immediately after each TSC packet, with a common timestamp copy (`ctc`) value in its payload; a Cycle Accurate (CYC) packet is generated immediately preceding TIP packets and provides the accurate `ctc` value since the last CYC packet. To extract exception timestamp, SPECULARIZER calculates `ccc` for each TIP packet based on the relationship between these time-related packets [7], as PT generates a TIP packet when an exception is raised.

Output. SPECULARIZER analyzes each PT trace offline, identifies and records all exceptions, the virtual address of the instruction that triggers it, as well as the timestamps of the identified exceptions. Two parameters, δ and μ, were involved in the data output: PT traces are segmented into windows of δ CPU cycles, and the attack detection algorithm runs over the traces in each window.

The output of this step is a set of sequences of two tuples, which is denoted as $X_{e_k} = [(c_1, t_1), (c_2, t_2), \cdots, (c_n, t_n)]$, where e_k is the virtual address of the instruction that triggers the exception, c_i indicates whether exists an exception of the i^{th} occurrence of the virtual address e_k, t_i is the timestamp of its occurrence, and μ is the length of each sequence, which is the input of attack detection model in Sect. 6. For each δ-cycle window, one or multiple sequences are gathered: if the total number of exceptions is greater than μ, a new sequence is created; a sequence less than μ is padded to μ with $(0, 0)$. We will evaluate the impact of different values of δ and μ on the effectiveness of the detection algorithm in Sect. 8.

5.2 Identifying Branch and Data Misprediction

SPECULARIZER identifies branch and data misprediction from the recorded HPC events. Particularly, SPECULARIZER first extracts the timestamp of each misprediction event from the HPC records, then extracts the timestamp of each branch instruction from the PT traces. Finally, by aligning the timestamp information from the HPC records and PT traces, SPECULARIZER outputs traces of correctly predicted and mispredicted branches for attack detection.

HPC Records Parsing. SPECULARIZER parses the HPC records and identifies the records that are related to either branch misprediction or data misprediction, and then outputs a sequence of two tuples: $[(c_1, t_1), (c_2, t_2), \cdots, (c_n, t_n)]$, where c_i is the event (*i.e.*, the branch misprediction or data misprediction) of the ith occurrence of the misprediction and t_i is its timestamp. The accuracy of misprediction information could depend on the PEBS overflow threshold ρ discussed in Sect. 4.

PT Trace Reconstruction. SPECULARIZER first reconstructs the program execution trace and the timestamp value of each branch with packets generated by the PT hardware. Meanwhile, PT timestamp packets are used to reconstruct the timestamp of each branch instruction using the method described in Sect. 5.1. By combining program execution trace with the timestamp information, SPECULARIZER outputs a sequence of two tuples: $[(b_1, t_1), (b_2, t_2), \cdots, (b_n, t_n)]$, where b_i is the virtual address of the ith occurrence of the branch and t_i is the timestamp when the branch is executed.

HPC and PT Alignment. SPECULARIZER aligns HPC records with the control-flow transfer information collected from PT to attribute HPC records to a specific branch of the program. The alignment can be performed by matching the timestamp value t_i in the two sequences. Particularly, for each element (c_k, t_k) in the HPC sequence, we search the PT sequence to find an element with index i that satisfies $t_i \leq t_k < t_{i+1}$. Then we associate (c_k, t_k) with b_i.

Output. For each δ-cycle window, each branch instruction b_k, SPECULARIZER outputs a set of sequences of two tuples, which is denoted as $X_{b_k} = [(c_1, t_1), (c_2, t_2), \cdots, (c_n, t_n)]$, where t_i is the timestamp when the i^{th} execution of the branch b_k, c_i indicates whether there is a misprediction and the misprediction type (*i.e.* branch or data) in the i^{th} execution of this branch, and μ is the length of the sequences.

6 Attack Detection

Given the traces produced in the previous section, SPECULARIZER uses the LSTM to extract the temporal information of the traces for attack detection. SPECULARIZER uses four detection models to detect four different attack types, which are exception-based attacks, misprediction-based attacks exploiting BTB/PHT, RSB, and memory disambiguator, respectively. These four detection models share the same layout: one LSTM layer and one Dense layer. Particularly, the detection model inputs the traces to the LSTM layer and outputs the likelihood for the trace to be an attack (between 0 and 1) from the Dense layer.

An end-to-end construction of SPECULARIZER, therefore, works as follows: (1) for every program monitored, both HPC and PT traces are collected and processed; (2) all processed traces for the program are classified by all four models. If one of the models classifies any of the traces as "attack" with a likelihood higher than a threshold α, the program is labeled by SPECULARIZER as performing speculative execution attacks.

7 Attack Variants Generation

To systematically evaluate how accurate SPECULARIZER can detect speculative execution attacks, we produce a data set of attack variants. To do so, we first propose parameterized models for speculative execution attacks and then systematically tuning the parameters of these models to generate a set of attack variants.

7.1 Exception-Based Attack Variants

Modeling Attacks. The attack model of exception-based speculative execution attacks is described in Fig. 3 (a), which depicts the timestamps of exceptions that happened at

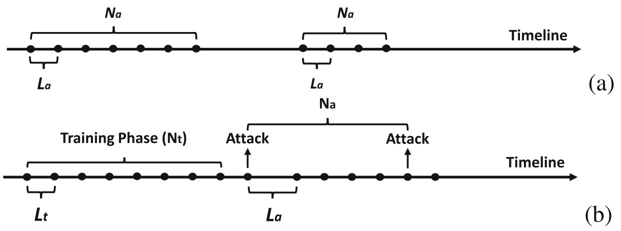

Fig. 3. Attack models for (a) exception-based attacks and (b) misprediction-based attacks.

Table 1. Relationship between N_a and p in exception-based attacks.

Success	Exception type								
rate (p)	US	RW	NM	BR	GP	P	LFB	LP	LVI
10%	1	1	17,000	1	10,000	300	3	1	4,000
30%	1	1	80,000	1	18,000	30,000	3	1	15,000
50%	2	1	130,000	1	58,000	60,000	3	1	28,000
80%	2	1	300,000	1	120,000	140,000	3	2	68,000
85%	3	1	400,000	1	140,000	180,000	3	2	86,000
90%	3	1	1,000,000	1	180,000	240,000	4	4	110,000
95%	4	1	1,300,000	1	300,000	400,000	8	6	140,000

a specific virtual address of the monitored program; each dot on the timeline represents the occurrence of an exception. N_a is the number of exceptions in a cluster that any two consecutive exceptions are no more than L_a cpu cycles apart.

To understand the practical implication of N_a and L_a, we performed an empirical evaluation of these two parameters using the Proof-of-Concept (PoC) code provided by Canella *et al.* [11]. We executed each of the PoC $10,000$ times when the system is idle and report the relationship between minimum N_a and the success rate (p) in Table 1. When the system is busy, the N_a increases for the same p. Therefore, we only present the data when the system is idle in Table 1. As we see from the result, when utilizing different *speculation primitives*, to have $p \geq 95\%$, N_a ranges from 1 to 1,300,000.

We also measured the relationship between L_a and p. When $FLUSH+RELOAD$ is selected as the *disclosure primitive*, it takes at least $150,000$ CPU cycles to finish reloading 255 elements (the minimum for encoding one byte). Therefore, with $N_a = 100$, we select L_a from 150K, 250K, 350K, 450K, 550K, 650K, 750K cycles. The experiment results suggest that the variation of L_a does not have an observable effect on p.

Generating Attack Variants. For each type of *speculation primitives* (*e.g.* #PF, #GP, *etc.*), we generate one attack variant for each of the following 23 value ranges for N_a: {[1, 1], [2, 2], [3, 3], [4, 4], [5, 5], [6, 6], [7, 7], [8, 8], [9, 9], [10, 10], [11, 20], [21, 30], [31, 40], [41, 50], [51, 60], [61, 70], [71, 80], [81, 90], [91, 100], [1,000, 10,000], [10,001, 100,000], [100,001, 1,000,000], [1,000,001, 2,000,000]}. In all attack variants,

L_a was chosen from 150K, 250K, 350K, 450K, 550K, 650K, 750K cycles. For each variant, N_a was chosen uniformly at *random* within the corresponding range.

Therefore, in total $9 \times 7 \times 23 = 1449$ attack variants were generated. Then we created 3 separate data sets from these samples. Specifically, we first selected three thresholds (*i.e.*, 85%, 90%, and 95%) for the attack success rate p, as attack variants with low p are meaningless, which will be discussed in Sect. 8. Second, for each p and each type of *speculation primitives*, we determine the minimum N_a such that attack variants with equal or greater N_a yield attack success rates larger than the corresponding p (from Table 1). As such, the three data sets have 476, 448, and 399 attack variants, respectively.

7.2 BTB/PHT Misprediction Variants

Modeling Attacks. To perform a successful misprediction-based speculative execution attack against BTB (*e.g.*, *Spectre-BTB*) and PHT (*e.g.*, *Spectre-PHT*), one needs to train (poison) the prediction unit in a loop multiple times before performing the attack to retrieve one byte of data [23]. This training can be performed either from the same address space or cross different address spaces [11]; moreover, the training can be performed either in-place or out-of-place [11]. Our detection target is the process that per-

Table 2. The success rate of misprediction-based attacks.

BTB/PHT											
N_t	L_t					N_a	L_a				
	350	450	550	650	750		150K	250K	350K	450K	550K
1	0.13	0.46	0.52	0.65	0.38	**1**	0.20	0.21	0.20	0.21	0.25
2	0.72	0.95	0.99	0.99	0.98	**3**	0.48	0.87	0.90	0.86	0.89
3	0.81	0.99	0.99	0.99	0.98	**5**	0.46	0.87	0.89	0.91	0.92
4	0.81	0.99	0.98	0.99	0.98	**10**	0.49	0.90	0.91	0.94	0.95
5	0.83	0.99	0.98	0.98	0.98	**30**	0.52	0.96	0.96	0.95	0.95
6	0.80	0.99	0.99	0.98	0.97	**50**	0.57	0.96	0.95	0.96	0.95
7	0.81	0.99	0.99	0.99	0.98	**100**	0.51	0.95	0.94	0.96	0.96
RSB						STL					
N_a	L_a					N_a	L_a				
	150K	250K	350K	450K	550K		150K	250K	350K	450K	550K
10	0.00	0.00	0.00	0.00	0.00	**10**	0.00	0.00	0.00	0.00	0.00
100	0.00	0.00	0.00	0.05	0.06	**100**	0.00	0.00	0.01	0.01	0.02
1,000	0.13	0.25	0.30	0.33	0.33	**1,000**	0.07	0.17	0.14	0.05	0.05
10,000	0.47	0.81	0.94	0.96	0.96	**10,000**	0.19	0.30	0.62	0.67	0.74
20,000	0.83	0.96	0.97	0.98	0.98	**20,000**	0.72	0.86	0.90	0.90	0.91
30,000	0.91	0.97	0.98	0.98	0.98	**30,000**	0.90	0.95	0.97	0.97	0.96
50,000	0.97	0.98	0.98	0.98	0.97	**50,000**	0.98	0.99	0.99	0.98	0.98

forms the training, regardless of whether it aims to perform same-address-space/cross-address-space or in-place/out-of-place attacks.

Therefore, the attack model of different types of misprediction-based speculative execution attacks is described in Fig. 3(b), which depicts the timestamps of branch/data prediction happened at a specific virtual address of the monitored program; each dot on the timeline represents the occurrence of one prediction. There are four parameters: N_a is the total number of attack attempts, L_a is the time interval between an attack attempt and the next training phase (in cpu cycles), N_t is the number of training attempts in each training phase, and L_t is the time interval between two consecutive training attempts.

To measure the parameters of the model, we used the PoC from Kocher *et al.* [23] and Canella *et al.* [11]. First, we tested the relationship between the occurrence of branch misprediction when the "attack" is performed and the success rate of the attack by leveraging the HPC event. The result shows that whenever the branch misprediction occurs, the attack can always have a 100% attack success rate. This is because the speculative window caused by BTB/PHT misprediction is large enough to load secret into the microarchitecture [52]. Therefore, an occurrence of a branch misprediction is equivalent to a successful attack.

Next, we evaluate how N_t and L_t affect the success rate of triggering branch misprediction. In the experiments described below, $N_a = 1$ and $L_a = 150$K cycles, and the result is shown in the BTB/PHT portion of Table 2. Each L_t is the CPU cycles (starting from the minimum value 350) and N_t enumerates each integer between 1 and 7 (inclusive). Each number in the table is the attack success rate in $10,000$ trials. As we see from the table, when $L_t \geq 450$ and $N_t \geq 2$, p is greater than 95%.

Finally, we evaluated how N_a and L_a affect the attack success rate (p). In these experiments, we set $N_t = 1$ and $L_t = 350$ cycles, because this pair of N_t and L_t has the worst p, which is the best scenario for analyzing the effects of N_a and L_a. The result in Table 2 shows that larger N_a has greater p. L_a has very little impact on p: For L_a between 250k and 550k CPU cycles, p is greater than 95% when $N_a > 30$. When L_a is large enough (*e.g.* $L_a >$ 450k CPU cycles), L_a has no observable effect on p.

Generating Attack Variants. For each type of *speculation primitives* (*e.g.*, BTB sa-ip, PHT ca-ip, *etc.*), we generate one attack variant for each combination of N_t, N_a, L_t, and L_a. The values of N_t and N_a are sampled uniformly at random from the following 14 value ranges: {[1, 1], [2, 2], [3, 3], [4, 4], [5, 5], [6, 6], [7, 7], [8, 8], [9, 9], [10, 10], [11, 20], [21, 30], [31, 50], [51, 100]}; The values of L_t are chosen from {350, 450, 550, 650, 750} CPU cycles; and the values of L_a are chosen from: {150K, 250K, 350K, 450K, 550K} CPU cycles. Therefore, in total $14 \times 14 \times 5 \times 5 \times 2 \times 4 = 39,200$ attack variants were generated. With the similar approach described in Sect. 7.1, we created 3 separate data sets with 37,904, 37,544, and 36,968 attack variants, respectively.

7.3 RSB and STL Misprediction Variants

Modeling Attacks. Spectre-RSB [11] and spectre-STL [11] exploits RSB and the memory disambiguator to trigger misprediction. In these two attacks, because RSBs can be poisoned by *push* and *pop* instructions, which is difficult to monitor using HPC and PT, and the memory disambiguator can be triggered simply by *load* instructions, which does not need training phase. Therefore, we use the model described in Fig. 3(a) to model these attacks.

To measure the impact of the parameters of the model on the success rate of the attacks, we used the PoC released with the published paper [11]. Using HPC events, we tested the relationship between p with N_a and L_a, respectively. Tested L_a and N_a start from the minimum ones, 150K and 1, respectively. The results are shown in Table 2. For Spectre-RSB, the value of N_a must be greater than $10,000$ for p to be larger than 90%. For Spectre-STL, the value of N_a must be greater than $20,000$ to achieve a similar success rate. For both attacks, L_a does not seem to play a significant role.

Generating Attack Variants. For each of RSB and memory disambiguator, we generated one attack variant for each of the following 15 value ranges for N_a:{[1, 10], [11, 100], [101, 1,000], [1,001, 2,000], [2,001, 3,000], [3,001, 4,000], [4,001, 5,000], [5,001, 6,000], [6,001, 7,000], [7,001, 8,000], [8,001, 9,000], [9,001, 10,000], [10,001, 20,000], [20,001, 30,000], [30,001, 50,000]} and 5 values for L_a: {150K, 250K, 350K, 450K, 550K}. In each variant, N_a was chosen uniformly at random during run-time with the corresponding value range. Therefore, in total $15 \times 5 \times 2 = 150$ attack variants were generated. With the similar approach described in Sect. 7.1, we created 3 separate data sets with 21, 20 and 14 attack variants, respectively.

8 Evaluation

In this section, we evaluate the detection accuracy and performance of SPECULARIZER. The data sets used in the evaluation are collected in the following approaches: The benign programs are selected from GNU Binutils[2] and SPEC benchmark 2006. The attack samples are drawn from the attack variants discussed in Sect. 7. The experiments were conducted on desktops with Intel Core i7-7700 Processors and 32 GB RAMs. 64-bit Ubuntu 16.04.6 LTS operating systems with the kernel version 5.4.0 were installed on the desktops.

8.1 Evaluation of SPECULARIZER's Parameters

There are a few parameters that can be tuned for SPECULARIZER: ① In the collection phase, the PEBS sampling rate (ρ) specifies the accuracy of branch misprediction records. We collected traces with 4 different ρ values: 1, 3, 5, 10. ② In the trace processing phase, the window size δ and trace length μ determine how the collected HPC and PT data are segmented for the LSTM algorithm to work on. We particularly picked two window sizes δ, 10 million CPU cycles and 100 million CPU cycles, and two trace lengths, 500 and 1000 data points. ③ The parameter we use to select *training data set* is the success rate p of the attack variants, which can be chosen from 85%, 90%, and 95%.

[2] https://www.gnu.org/software/binutils/.

Table 3. Data sets for parameter evaluation.

Index	Window size δ	Trace length μ	Sample ρ	Success rate p
1	10M	1000	1	95%
2	100M	1000	1	95%
3	100M	500	1	95%
4	100M	1000	3	95%
5	100M	1000	5	95%
6	100M	1000	10	95%
7	100M	1000	1	90%
8	100M	1000	1	85%

In this section, we analyze how these parameters affect the detection results. We created 8 data sets, whose parameter configuration is shown in Table 3. Each data set contains four groups of traces; each group is used to evaluate one LSTM model, as specified in Sect. 6. In each group, around $30,000$ benign traces and $30,000$ attack traces were collected. Then the traces in each group are randomly split into the training set (80%) and a testing set (20%).

By running the LSTM classification, the algorithm outputs a class label ("benign" or "attack") for each trace together with the likelihood between 0 and 1. We selected a threshold α of the likelihood, such that SPECULARIZER alerts the detection of an "attack" trace when the LSTM classifier outputs "attack" with a likelihood greater than α. Two values were selected for α, 0.5 and 0.75. The accuracy is evaluated using the F1 scores when $\alpha = 0.5$ and when $\alpha = 0.75$. A high F1 score suggests a balanced precision and recall. Here recall is defined as the percentage of detected attack traces in all attack traces and precision is defined as the percentage of correctly detected attack traces in all detected attack traces.

PEBS Sampling Rate ρ. We only evaluated ρ for misprediction-based attacks, because the detection of exception-based attacks does not need HPC. The data sets we used in this test are (2), (4), (5), and (6) (as shown in Table 3), where the window size (δ) is selected as 100 million cycles, trace size (μ) is selected as 1000, and $p > 95\%$ for attack variants selected in the training/testing set. The result is shown in Table 4. We see from the table that ρ only affects the detection of BTB/PHT-based attacks. Specifically, only when $\rho \leq 3$, F1 scores yield good detection accuracy (F1 score greater than 90%). In contrast, regardless of the ρ value, the detection accuracy for RSB and STL-based attacks is high. This is because losing branch misprediction information due to larger ρ values is more critical to detecting attacks that require training.

Window Size δ. To evaluate the effect of δ, we used data set (1) and (2), where for both data sets $\rho = 1$, $p > 95\%$, and $\mu = 1000$. For each window size (δ), we evaluate the F1 score when the threshold is 0.5 and 0.75 and the result is shown in Table 5, which suggests δ does not have a strong impact on the detection accuracy.

Table 4. Impact of PEBS sample rate ρ.

Misprediction-based	PEBS sampling rate ρ							
	1		3		5		10	
	F1 (0.5)	F1 (0.75)	F1 (0.5)	F1 (0.75)	F1 (0.5)	F1 (0.75)	F1 (0.5)	F1 (0.75)
BTB/PHT	0.977	0.977	0.910	0.909	0.716	0.715	0.593	0.593
RSB	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
STL	0.993	0.991	0.974	0.971	0.955	0.954	0.913	0.911

Table 5. Impact of window size δ, trace length μ and threshold of attack success rate p.

		Window Size δ (CPU Cycles)				Trace length μ (elements)				Attack Success Rate Threshold p					
		10M		100M		500		1000		0.85		0.90		0.95	
		F1 (0.5)	F1 (0.75)	F1 (0.5)	F1 (0.75)	F1 (0.5)	F1 (0.75)	F1 (0.5)	F1 (0.75)	F1 (0.5)	F1 (0.75)	F1 (0.5)	F1 (0.75)	F1 (0.5)	F1 (0.75)
Misprediction-based	BTB/PHT	0.963	0.962	0.977	0.977	0.969	0.970	0.977	0.977	0.976	0.976	0.976	0.975	0.977	0.977
	RSB	0.992	0.992	1.0	1.0	0.997	0.997	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	STL	0.992	0.993	0.993	0.991	0.994	0.995	0.993	0.991	0.991	0.991	0.990	0.989	0.993	0.991
Exception-based		0.945	0.937	0.960	0.955	0.936	0.938	0.960	0.955	0.961	0.960	0.959	0.956	0.960	0.955

Trace Length μ. The evaluation utilized data set (2) and (3), with $\rho = 1$, $p \geq 95\%$ and $\delta = 100$M cycles. The result is presented in Table 5, which means $\mu = 500$ or $\mu = 1000$ does not affect the detection accuracy dramatically.

Success Rate Threshold p. The data sets used in this evaluation are (2), (7), and (8), with $\rho = 1$, $\mu = 1000$ and $\delta = 100$M cycles. The result shown in Table 5 suggests that p does not have much impact on the detection accuracy.

Classification Likelihood Threshold α. The result shown in Table 4 and Table 5 suggests that $\alpha = 0.5$ or $\alpha = 0.75$ does not affect F1 score. Thus, we chose $\alpha = 0.5$ for the following evaluation.

Summary: In parameters ρ, δ, μ, p, and α; only ρ has a significant impact on detection accuracy of attacks that exploit BTB/PHT.

8.2 Evaluation of Detection Accuracy

We evaluated the detection accuracy of the LSTM models trained using data set (4) in Table 3. Using these parameters, in the following experiments, we evaluate the models' capability of detecting various attack variants. Because the traces that are classified as benign all have a precision that is close to 100%, the F1 score does not provide more information than recall, or true positive rate (TPR). Therefore, we use TPR as the metric for evaluating detection accuracy, which is defined as the percentage of correctly classified traces among all traces that are classified as attacks. The results are represented in Fig. 4, while the blue line is the TPR and the red line is the attack success rate p. In the cases where TPR $> p$ means the probabilistic to detect the attack is higher than the secret been leaked.

Exception-Based Attacks. We collected $11,700$ traces from all types of exception-based variants we generated and split them into separate groups according to their N_a

value. Then we perform classification on each of the groups and show the results in Fig. 4(a). In this figure, the X-axis is the value of N_a, the red line is the attack success rate p and the blue line is the TPR. When $N_a = 4$, TPR $= 99.1\%$; when $N_a > 10$, TPR $\geq 99.9\%$; but when $N_a \leq 3$, TPR drops to 0, which means we were not able to detect exception-based attacks with fewer than 4 attempts within a time window of 100 million CPU cycles.

Misprediction-Based Attacks on BTB/PHT. We collected $980,000$ traces from the BTB/PHT attack variants. To evaluate SPECULARIZER with varying N_t, N_a, L_t, and L_a values, we split the traces accordingly. The results are shown in Fig. 4 (b), Fig. 4 (d), Fig. 4 (c) and Fig. 4 (d), respectively. As we see from these figures, SPECULARIZER can detect attack variants with $N_t \geq 2$, $N_a \geq 3$, $350 \leq L_t \leq 750$ CPU cycles, and 150K $\leq L_a \leq$ 550K CPU cycles with TPR $\geq 90\%$.

Misprediction-Based Attacks on RSB and Memory Disambiguator. We collected $7,500$ traces from attack variants exploiting RSB and memory disabmiguators. To evaluate SPECULARIZER with varying L_a and N_a values, we split the traces accordingly. The results are presented in Fig. 4 (g), Fig. 4 (i), Fig. 4 (f), and Fig. 4 (h), respectively. As we see from these figures, SPECULARIZER can detect attack variants with 150K $\leq L_a \leq$ 550K cpu cycles with TPR $> 80\%$. TPR increases almost monotonically when N_a increases. SPECULARIZER can detect attack variants when $N_a > 3000$ with TPR $> 80\%$. It is worth noting that when TPR $< 80\%$ for both attacks, the success rate of these attacks goes below 30%, which suggests that the adversary needs to balance the attack efficiency with the risk of detection.

Summary: With the selected parameters, SPECULARIZER can detect most of the attack traces we collected from the generated attack variants with high recalls. However, In cases where the detection is less accurate, p of these attack variants is also low.

8.3 End-to-End Evaluation

In practice, SPECULARIZER monitors the execution of a program and raises alarms if any of the traces collected from the program is detected as "attacks". To perform end-to-end evaluation, we use the same model as trained using data set (4).

The data set we used has 26 benign programs collected from GNU Binutils and SPEC benchmark 2006, and randomly selected 160 attack variants we generated (*i.e.* 40 variants for each attack type). Each of the 186 programs was examined using all the four LSTM models. Among the 41 benign programs, only one (*gobmk*) is falsely classified as BTB/PHT misprediction attacks and four benign programs (*i.e.*, *ld*, *perlbench*, *sophlex*, and *gobmk*) were misclassified as exception-based attacks. However, in all these misclassified cases, less than 3 traces (out of over 1000 traces) extracted from each program were indeed misclassified, which means these false detections can be prevented if SPECULARIZER only raises alarms when multiple traces (*e.g.*, > 3) were detected as attacks, which can be another parameter the user of SPECULARIZER could tune. Nevertheless, all attack variants are successfully detected by their corresponding LSTM classifier. The BTB/PHT classifier also detects 117 out of 120 other attack variants, because these attack variants also exhibit this type of branch misprediction.

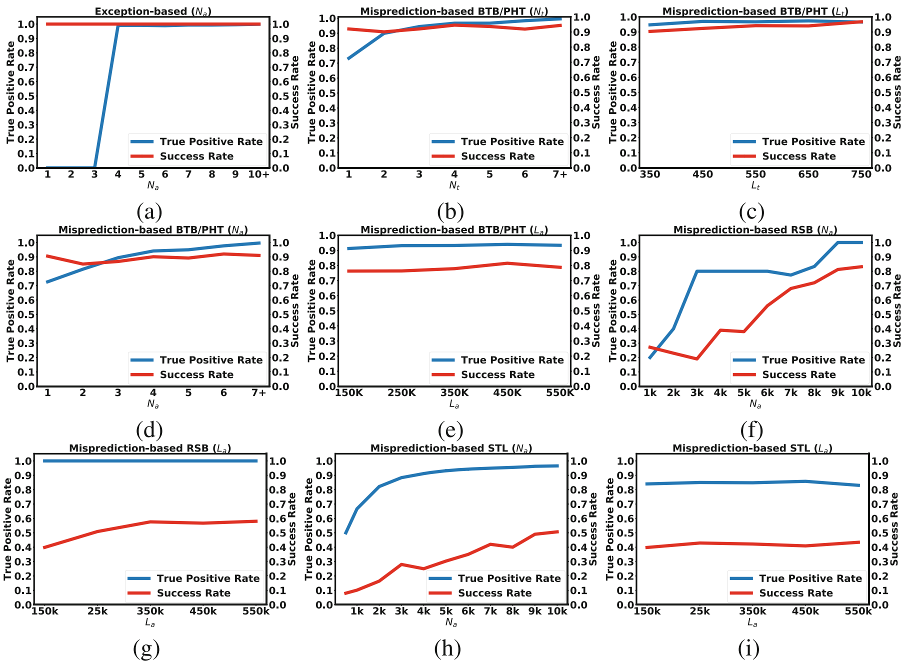

Fig. 4. Accuracy of attack detection (recall). (a) N_a in exception-based attacks, (b) N_t in BTB/PHT misprediction attacks, (c) L_t in BTB/PHT misprediction attacks, (d) N_a in BTB/PHT misprediction attacks, (e) L_a in BTB/PHT misprediction attacks, (f) N_a in RSB misprediction attacks, (g) L_a in RSB misprediction attacks, (h) N_a in STL attacks, (i) L_a in STL attacks. (Color figure online)

8.4 Performance Analysis

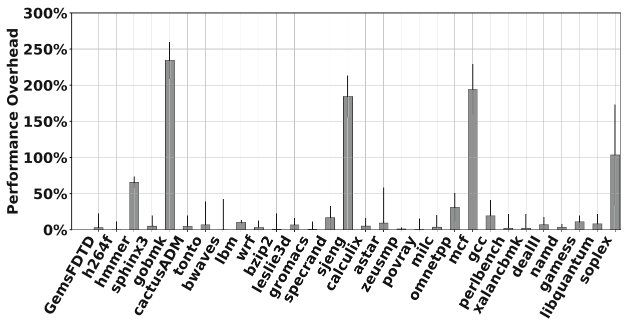

Fig. 5. The performance overhead of online trace collection.

Overhead of Online Trace Collection. In our experiments, SPECULARIZER enabled trace collection of both HPC and PT, with the HPC events and MSR configurations

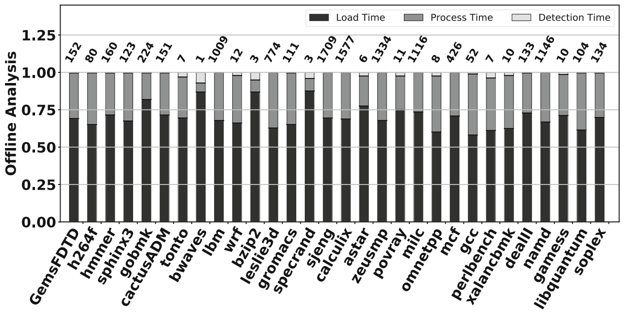

Fig. 6. Running time of offline attack detection.

specified in Sect. 4. The ρ was set to 3. The experiments on LMbench [29] show the runtime overhead on I/O is negligible. The results of the SPEC benchmark are shown in Fig. 5. The runtime overhead was introduced from 0.038% to 231.42%, with a geometric mean of 14.36%. Some of the benchmark programs (*e.g.* mcf, gobmk, and sjeng) had high performance overhead; as their execution triggers a lot of branch misprediction. We note that the performance can be reduced with Intel's new feature that redirects PEBS's sampling output to PT packets [7], as PT packet generation introduces much less overhead [16]. We leave this evaluation to future work.

Running Time of Offline Attack Detection. Figure 6 shows the running time of offline attack detection. The number above each bar is the average running time (of 1000 trials) of the offline analysis for each SPEC benchmark (in seconds), which ranges from 3 s to 1709 s with PT trace files between 13M bytes and 13G bytes. More specifically, the offline analysis includes three phases: trace loading, trace processing, and attack detection. On average, they account for 70.01%, 29.85%, and 0.14% of the entire running time. One reason for the long-running time for trace loading/processing (99.86%) is that PT generates a large number of packets, which takes a long time to parse and analyze. The attack detection phase typically takes less than 1s. Finally, it is worth noting our offline analyses were performed within a single thread with limited memory, which can be further optimized using multi-threading and larger memory. And for applications such as forensics, the overhead of offline analysis is not critical.

9 Discussion

Detecting Assistance-Based Attacks. Microcode assist was exploited in some recent works [31,37,42,43]. However, there is no systematic study of these microcode assists yet. It is not clear how many methods can trigger microcode assists and how many of them can be exploited in speculative execution attacks by unprivileged programs. Without such systematic exploration, an ad-hoc detection technique is likely to be bypassed. We leave the detection of assistance-based attacks to future work.

Completeness of the Attack Data Sets. We could hardly claim that our generated attack data sets cover all possible attack variants. However, as the models used for attack variants generation only specify the patterns of misprediction and exception, they are general enough for modeling attacks that use different types of disclosure gadgets and disclosure primitives. Moreover, the parameters in the attack models can be tuned to alter specific properties of an attack variant, which in combination can be used to approximate most attack methods one could think of.

Using Simpler Classification Models for Attack Detection. One might think deep learning algorithms like LSTM are too heavyweight for our scenarios. In fact, we have also tested multiple alternatives, such as decision trees, K-means, random forest, *etc.* However, we found those models very fragile for any practical use. In contrast, LSTM offers an automated selection of parameters and thresholds, greatly reducing the subjectivity in the selection of classification models.

Adversarial Machine Learning (AML). SPECULARIZER is vulnerable to AML-based techniques that generate carefully crafted attack variants to evade detection. As shown in Fig. 4, in general, attack code that evades detection is likely to have a lower success rate. In that sense, SPECULARIZER makes speculative execution attacks harder to perform, but may not eliminate the threats. However, we note this arms race is common in all machine-learning-based defense systems [14,32].

Real-Time Attack Detection. Ideally, attack detection should be performed in real-time and for all programs. However, as parsing PT packets and processing the traces are time-consuming (as shown in Fig. 6), it is very challenging to do so in practice. Moreover, enabling whole system monitoring with PT will drastically increase the overhead of trace parsing and analysis. These are common issues for PT/HPC-based monitoring systems [39,51].

10 Related Work

Detecting Speculative Execution Attacks. Prior works on detecting speculative execution attacks mainly focus on the detection of *disclosure primitives*, such as the Flush+Reload cache side channels [19]. In contrast, SPECULARIZER detects the speculative execution attacks by monitoring its root cause—the *speculation primitives*. Close to our work is due to [25,44] who also leverages HPC to detect speculative execution attacks. However, as their approach only uses HPC, it omits the context of program execution in the detection of attacks. Therefore, their approach is less accurate and only applicable to simple proof-of-concept attacks.

Mitigating Speculative Execution Attacks. Software solutions provide temporary mitigation of the threats, which are reactive to only known attacks and ad hoc. For instance, page table isolation (*e.g.*, KPTI of Linux) PTE inversion, and L1d flush [5], compiler-based mitigation [12,23,30,40] provides generic solutions for exception-based and misprediction-based speculative execution attacks. *SPECCFI* [24], *ConTExT* [36] mitigates a specific type of speculative attack. Furthermore, many works focus on detecting the code gadget of speculative execution attacks [13,18,20,28,33,47].

Proposals from the computer architecture research community mitigate speculative execution attacks with more dramatic revision on the micro-architectural level [2,4, 6,9,10,15,21,22,26,34,35,38,43,46,48–50]. While these approaches may be efficient in addressing the targeted problems, however, it may take a longer time before these academic proposals can be adopted by the industry.

11 Conclusion

In this paper, we present Specularizer, a software tool for uncovering speculative execution attacks using performance tracing hardware features (PT and HPCs). Specularizer monitors the execution of the inspected applications in an online mode, introducing modest runtime performance overhead, and then performs attack detection in an offline analysis using LSTM networks. Empirical evaluation of Specularizer suggests that the proposed approach leads to high detection accuracy with reasonable overhead, particularly suitable for offline forensic analysis.

References

1. Deep dive: Intel analysis of microarchitectural data sampling (2019). https://software.intel.com/security-software-guidance/insights/deep-dive-intel-analysis-microarchitectural-data-sampling
2. Intel analysis of speculative execution side channels. Revision 4.0. Accessed July 2018
3. Mitigating speculative execution side channel hardware vulnerabilities (2018). https://msrc-blog.microsoft.com/2018/03/15/%20mitigating-%20speculative-%20execution-%20side-%20channel-%20hardware-%20vulnerabilities/
4. Speculative execution side channel mitigations (2018). http://kib.kiev.ua/x86docs/SDMs/336996-001.pdf. Revision 1.0, January 2018
5. Deep dive: Intel analysis of l1 terminal fault (2019). https://software.intel.com/security-software-guidance/insights/deep-dive-intel-analysis-l1-terminal-fault
6. Engineering new protections into hardware (2019). https://www.intel.com/content/www/us/en/architecture-and-technology/engineering-new-protections-into-hardware.html
7. Intel® 64 and IA-32 architectures software developer's manual (2019). https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf. January 2019
8. Akiyama, S., Hirofuchi, T.: Quantitative evaluation of intel PEBS overhead for online system-noise analysis. In: 7th International Workshop on Runtime and Operating Systems for Supercomputers (ROSS). ACM (2017)
9. Barber, K., Bacha, A., Zhou, L., Zhang, Y., Teodorescu, R.: SpecShield: shielding speculative data from microarchitectural covert channels. In: 28th International Conference on Parallel Architectures and Compilation Techniques (PACT). IEEE (2019)
10. Bourgeat, T., Lebedev, I., Wright, A., Zhang, S., Devadas, S.: Mi6: secure enclaves in a speculative out-of-order processor. In: 52nd Annual IEEE/ACM International Symposium on Microarchitecture (2019)
11. Canella, C.: A systematic evaluation of transient execution attacks and defenses. In: 28th USENIX Security Symposium (2019)
12. Carruth, C.: Speculative load hardening (2018)

13. Chen, G., Chen, S., Xiao, Y., Zhang, Y., Lin, Z., Lai, T.H.: Stealing intel secrets from SGX enclaves via speculative execution. In: IEEE European Symposium on Security and Privacy, June 2019
14. Chen, L., Sultana, S., Sahita, R.: HeNet: a deep learning approach on intel® processor trace for effective exploit detection. In: IEEE Security and Privacy Workshops (SPW). IEEE (2018)
15. Fustos, J., Farshchi, F., Yun, H.: SpectreGuard: an efficient data-centric defense mechanism against Spectre attacks. In: 56th Annual Design Automation Conference (2019)
16. Ge, X., Cui, W., Jaeger, T.: GRIFFIN, guarding control flows using intel processor trace. ACM SIGPLAN Not. (2017)
17. Gu, Y., Zhao, Q., Zhang, Y., Lin, Z.: PT-CFI: transparent backward-edge control flow violation detection using intel processor trace. In: 7th ACM on Conference on Data and Application Security and Privacy (2017)
18. Guarnieri, M., Köpf, B., Morales, J.F., Reineke, J., Sánchez, A.: Spectector: principled detection of speculative information flows. In: IEEE Symposium on Security and Privacy (SP) (2020)
19. Gulmezoglu, B., Moghimi, A., Eisenbarth, T., Sunar, B.: FortuneTeller: predicting microarchitectural attacks via unsupervised deep learning (2019). arXiv preprint arXiv:1907.03651
20. Guo, S.: SpecuSym: speculative symbolic execution for cache timing leak detection (2019). arXiv preprint arXiv:1911.00507
21. Khasawneh, K.N., Koruyeh, E.M., Song, C., Evtyushkin, D., Ponomarev, D., Abu-Ghazaleh, N.: SafeSpec: banishing the spectre of a meltdown with leakage-free speculation (2018). arXiv preprint arXiv:1806.05179
22. Kiriansky, V., Lebedev, I.A., Amarasinghe, S.P., Devadas, S., Emer, J.: DAWG: a defense against cache timing attacks in speculative execution processors. IACR Cryptology ePrint Archive (2018)
23. Kocher, P., et al.: Spectre attacks: exploiting speculative execution. In: 40th IEEE Symposium on Security and Privacy (S&P) (2019)
24. Koruyeh, E.M., Shirazi, S.H.A., Khasawneh, K.N., Song, C., Abu-Ghazaleh, N.: SPECCFI: mitigating Spectre attacks using CFI informed speculation (2019). arXiv preprint arXiv:1906.01345
25. Li, C., Gaudiot, J.-L.: Detecting malicious attacks exploiting hardware vulnerabilities using performance counters. In: 43rd Annual Computer Software and Applications Conference (COMPSAC). IEEE (2019)
26. Li, P., Zhao, L., Hou, R., Zhang, L., Meng, D.: Conditional speculation: an effective approach to safeguard out-of-order execution against Spectre attacks. In: IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE (2019)
27. Lipp, M., et al.: Meltdown: reading kernel memory from user space. In: 27th USENIX Security Symposium (2018)
28. Mambretti, A., Neugschwandtner, M., Sorniotti, A., Kirda, E., Robertson, W., Kurmus, A.: Speculator: a tool to analyze speculative execution attacks and mitigations. In: 35th Annual Computer Security Applications Conference (2019)
29. McVoy, L.W., Staelin, C., et al.: lmbench: portable tools for performance analysis. In: USENIX Annual Technical Conference (1996)
30. Miller, M.: Mitigating speculative execution side channel hardware vulnerabilities. Microsoft Security Response Center (MSRC) (2018)
31. Minkin, M., et al.: Fallout: reading kernel writes from user space (2019)
32. Moosavi-Dezfooli, S.-M., Fawzi, A., Fawzi, O., Frossard, P.: Universal adversarial perturbations. In: IEEE Conference on Computer Vision and Pattern Recognition (2017)
33. Oleksenko, O., Trach, B., Silberstein, M., Fetzer, C.: SpecFuzz: bringing spectre-type vulnerabilities to the surface. In: 29th USENIX Security Symposium (2020)

34. Saileshwar, G., Qureshi, M.K.: CleanupSpec: an "undo" approach to safe speculation. In: 52nd Annual IEEE/ACM International Symposium on Microarchitecture (2019)
35. Sakalis, C., Kaxiras, S., Ros, A., Jimborean, A., Själander, M.: Efficient invisible speculative execution through selective delay and value prediction. In: 46th International Symposium on Computer Architecture. ACM (2019)
36. Schwarz, M., Lipp, M., Canella, C., Schilling, R., Kargl, F., Gruss, D.: ConTexT: a generic approach for mitigating Spectre. In: Network and Distributed System Security Symposium (NDSS) (2020)
37. Schwarz, M., et al.: ZombieLoad: cross-privilege-boundary data sampling (2019). arXiv:1905.05726
38. Taram, M., Venkat, A., Tullsen, D.: Context-sensitive fencing: securing speculative execution via microcode customization. In: 24th International Conference on Architectural Support for Programming Languages and Operating Systems. ACM (2019)
39. Thalheim, J., Bhatotia, P., Fetzer, C.: INSPECTOR: data provenance using intel processor trace (PT). In: 36th International Conference on Distributed Computing Systems (ICDCS). IEEE (2016)
40. Turner, P.: Retpoline: a software construct for preventing branch-target-injection (2018). https://support.google.com/faqs/answer/7625886
41. Van Bulck, J: Foreshadow: extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In: 27th USENIX Security Symposium (2018)
42. Van Bulck, J.: LVI: hijacking transient execution through microarchitectural load value injection. In: 41th IEEE Symposium on Security and Privacy (S&P) (2020)
43. van Schaik, S.: RIDL: rogue in-flight data load. In: IEEE Symposium on Security and Privacy (S&P) (2019)
44. Wang, H., Sayadi, H., Rafatirad, S., Sasan, A., Homayoun, H.: SCARF: detecting side-channel attacks at real-time using low-level hardware features. In: 26th International Symposium on On-Line Testing and Robust System Design (IOLTS). IEEE (2020)
45. Wang, W., Zhang, Y., Lin, Z.: Time and order: towards automatically identifying side-channel vulnerabilities in enclave binaries. In: 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID) (2019)
46. Weisse, O., Neal, I., Loughlin, K., Wenisch, T.F., Kasikci, B.: Nda: preventing speculative execution attacks at their source. In: 52nd Annual IEEE/ACM International Symposium on Microarchitecture (2019)
47. Xiao, Y., Zhang, Y., Teodorescu, R.: Speechminer: a framework for investigating and measuring speculative execution vulnerabilities. In: Network and Distributed System Security Symposium (NDSS) (2020)
48. Yan, M., Choi, J., Skarlatos, D., Morrison, A., Fletcher, C.W., Torrellas, J.: InvisiSpec: making speculative execution invisible in the cache hierarchy. In: International Symposium on Microarchitecture (2018)
49. Yu, J., Hsiung, L., Hajj, M.E., Fletcher, C.W.: Data oblivious ISA extensions for side channel-resistant and high performance computing. In: Network and Distributed System Security Symposium (NDSS) (2019)
50. Yu, J., Yan, M., Khyzha, A., Morrison, A., Torrellas, J., Fletcher, C.W.: Speculative taint tracking (STT) a comprehensive protection for speculatively accessed data. In: 52nd Annual IEEE/ACM International Symposium on Microarchitecture (2019)
51. Zhang, T., Jung, C., Lee, D.: ProRace: practical data race detection for production use. ACM SIGOPS Oper. Syst. Rev. **51**(2), 149–162 (2017)
52. Zhang, Z., Cheng, Y., Zhang, Y., Nepal, S.: GhostKnight: breaching data integrity via speculative execution (2020). arXiv preprint arXiv:2002.00524