Fast, Robust and Accurate Detection of Cache-based Spectre Attack Phases

Arash Pashrashid
National University of Singapore
pashrashid.arash@u.nus.edu

Ali Hajiabadi
National University of Singapore
ali.hajiabadi@u.nus.edu

Trevor E. Carlson
National University of Singapore
tcarlson@comp.nus.edu.sg

ABSTRACT

Modern processors achieve high performance and efficiency by employing techniques such as speculative execution and sharing resources such as caches. However, recent attacks like Spectre and Meltdown exploit the speculative execution of modern processors to leak sensitive information from the system. Many mitigation strategies have been proposed to restrict the speculative execution of processors and protect potential side-channels. Currently, these techniques have shown a significant performance overhead. A solution that can detect memory leaks before the attacker has a chance to exploit them would allow the processor to reduce the performance overhead by enabling protections only when the system is at risk.

In this paper, we propose a mechanism to detect speculative execution attacks that use caches as a side-channel. In this detector we track the phases of a successful attack and raise an alert before the attacker gets a chance to recover sensitive information. We accomplish this through monitoring the microarchitectural changes in the core and caches, and detect the memory locations that can be potential memory data leaks. We achieve 100% accuracy and negligible false positive rate in detecting Spectre attacks and evasive versions of Spectre that the state-of-the-art detectors are unable to detect. Our detector has no performance overhead with negligible power and area overheads.

ACM Reference Format:
Arash Pashrashid, Ali Hajiabadi, and Trevor E. Carlson. 2022. Fast, Robust and Accurate Detection of Cache-based Spectre Attack Phases. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD '22), October 30-November 3, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3508352.3549330

1 INTRODUCTION

Modern general-purpose processors provide high levels of performance and efficiency through different strategies like speculative execution with the state-of-the-art branch prediction techniques. These processors share resources like caches with other cores and different processes running on the same core. While general-purpose processors have evolved around these strategies in the past decades to achieve their current level of efficiency, recent speculative execution attacks, like Spectre [14], allow the malicious users to extract sensitive and private information from the system.

ICCAD '22, October 30-November 3, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9217-4/22/10.
https://doi.org/10.1145/3508352.3549330

Many defenses have been proposed in literature to mitigate speculative execution attacks in different ways [12, 26, 30] and most of them are focused on protecting the caches as the most common and practical side-channel to leak data [18]. However, all currently proposed mitigation strategies are always-enabled, operating even if no malicious activity is happening and the system is not at risk. To improve performance, researchers have proposed various detection mechanisms to turn on these protection mechanisms only if the system is at risk. Most recent detectors exploit Machine Learning (ML) to detect malicious activities by either monitoring the Hardware Performance Counters (HPC) or monitoring the microarchitecture-level statistics [8, 20–22, 25, 28, 29, 32]. While these techniques achieve high detection accuracy with a low performance overhead, they become significantly less effective when evasive attacks manipulate the ML-based detectors to miss-classify malicious activities.

In this paper, we first propose two new evasive Spectre attacks that aim to reduce the detection accuracy of ML-based detectors: (1) Expanded-Spectre, and (2) Benign-Program-Spectre attacks (Section 3). The former attack tries to change the footprint of a Spectre attack in a way to classify the attack as benign while completing the attack successfully. The latter attack is completely constructed from benign programs. All gadgets used for the attack are labeled as benign but by connecting these benign gadgets the attacker is able to leak arbitrary and potentially sensitive information. Even optimizing ML-based attacks with evasion resilient techniques is ineffective against our attack. For example, RHMD [11] makes the detector's sampling frame unknown for the attacker but still will not be able to distinguish our evasive Spectre attacks.

To overcome these shortcomings, we propose the Spectify detection methodology. This technique relies on microarchitecture-level information to detect different phases of a Spectre attack and to detect the exact memory data leaks and problematic speculative branches (Section 5). Our methodology allows the system to determine the details about the data leaks before a potential attacker gets a chance to reconstruct the leaked data and employs an appropriate protection. Our detection methodology is based on the general steps of a speculative execution attack and all known variations of the attack can be tracked by this methodology.

We demonstrate that our detector achieves 100% detection accuracy for known speculative execution attacks and also our evasive Spectre attacks. Spectify has 0% false negatives on our dataset which is crucial for a detector in order to provide a strong security guarantee. Even a small percentage of false negatives can be problematic as a smart attacker might find a way to exploit this. ML-based techniques cannot guarantee 0% false negatives since adversarial attacks can always be a threat to trick an ML model to miss-classify malicious activities. Also, we have a negligible false

positive rate while running benign programs. As false positives reported by our detector are actual data leaks from benign programs the system should protect the leaked data. In other words, the system needs to protect all the data leaks reported by our detector in order to guarantee there are no data leaks.

Finally, we provide a microarchitectural implementation of our detector (Section 6). Our implementation shows no performance overheads since all the operations are off the critical path; we also show low overheads in terms of power and area consumption.

The main contributions of our work are as follows:

- We significantly reduce the accuracy of a state-of-the-art detector, PerSpectron, through the use of two new evasive variants of Spectre: (1) Expanded-Spectre, and (2) Benign-Program-Spectre (Section 3).
- We propose Specтify, a new detector to overcome the limitations of ML-based solutions. We exploit the low-level information in the microarchitecture to track the required sequence of activities to launch a successful attack (Section 5).
- We show that Specтify achieves 100% detection accuracy for known attacks and our proposed evasive attacks with negligible false positive rate while running benign programs. Specтify has no performance overhead and negligible power and area overheads (Section 8).

In the rest of the paper, we provide background on speculative execution attacks and their detection in Section 2. In Section 3 we introduce our evasive variants of Spectre attack and the limitations of ML-based detectors to detect these attacks. In Section 5 and Section 6 we describe our detection methodology and microarchitecture. In Section 7 we describe our experimental setup and in Section 8 we present our results. Finally, Section 9 summarizes the related works.

One can download the Specтify software infrastructure at: https://github.com/Spectify-Detector/Spectify.

2 BACKGROUND

In this section, we will discuss relevant background information to understand how speculative execution attacks work and why it is important to have a detection mechanism that is accurate, robust, fast, and efficient.

2.1 Speculative Execution Attacks

Modern CPUs utilize aggressive instruction scheduling to execute instructions out-of-order as soon as their operands are ready. The processor avoids stalling execution in situations where a branch value of a jump target is unknown by predicting the branch target and speculatively executing instructions. Speculatively executed instructions will be squashed in case of a misprediction. The processor aims to roll-back the system's state into a safe state but some speculatively executed instructions can leave traces of execution in system components. Recent attacks like Spectre [14] and Meltdown [17] exploit speculative execution of modern processors to leak sensitive information from the system. Listing 1 shows the steps of a Spectre attack. Below, we walk through the main four steps of a successful attack:

Step 1 (branch mistraining). As the first step, the attacker mistrains the branch predictor that the victim is going to make and access memory (see line 13 in Listing 1). This step ensures that this branch is always taken and the branch prediction unit (BPU) will learn to predict it as taken with high confidence.

```
1  unit8 A[10];
2  uint8 B[256*64]; //side-channel (SC)
3
4  void victim(size_t addr){
5      if (addr < 10){ //safety check - mispredicts
6          uint8 val = A[addr]; //accesses secret
7          uint8 x = B[val*64]; //transmits secret to SC
8      }
9  }
10
11 void attack(){
12     //Phase 1: branch mistraining
13     for (i = 0; i < 10000; i++) victim(0);
14
15     //Phase 2: initializing the side-channel
16     for (i = 0; i < 256; i++) clflush(B[i*64]);
17
18     //Phase 3: running the victim
19     size_t secret_offset = secret_address - A;
20     victim(secret_offset);
21
22     //Phase 4: recovering the secret
23     for (guess = 0; guess < 256; guess++){
24         t1 = rdtscp(); //read timer
25         temp = B[guess*64]; //accessing the guess
26         t2 = rdtscp(); //read timer
27         if (t2-t1 <= CACHE_HIT_THRESHOLD)
28             results[guess] += 1; //potential secret
29     }
30 }
```

Listing 1: Spectre attack based on Flush+Reload

Step 2 (initializing a side-channel). In this step, the attacker initializes a side-channel into a known state. For example, the attacker can use a Flush+Reload cache-based side-channel to flush 256 cache lines[1] (see line 16 in Listing 1). This allows the attacker to recover the data that the victim has speculatively brought to the cache at a later time.

Step 3 (speculative access). After the side-channel is initialized by the attacker, the victim will run and speculatively access the secret, leaking the data. This is the mispredicted branch path, but since the branch is trained to be taken, it will speculatively run the code (see line 6 in Listing 1). Next, the secret is transmitted to the side-channel (i.e., encoded in one of the flushed cache lines; see line 7 in Listing 1).

Step 4 (recovering the secret). The processor will squash the speculative instructions executed by the victim, but the transmitted information via the side-channel will not change. In the final step, the attacker will probe all of the initialized states of the side-channel and try to reconstruct the secret information based on the changes that occurred to the initial state of the side-channel. In the example of Flush+Reload in Listing 1, all the cache lines will miss in the cache except for the cache line that is associated with the secret byte. This line will be a hit in the cache and the difference in access latency will enable the attacker to recover the secret byte.

2.2 Cache-based Side-Channels

As demonstrated in Section 2.1, the attacker needs to initialize a side-channel and later probe the same side-channel in order to recover the secret information. The most important feature of an effective

[1] To extract one byte of secret, there are 256 possible guesses and the attacker needs to initialize 256 states and later evaluate these 256 states and recover the secret based on the changes that happened to them.

side-channel is that all the changes during speculative execution need to be persistent and not rolled back after the core handles the misspeculation. Currently, the most common and practical side-channel is the cache [27]. In this paper, we focus on the most practical forms of cache-based side-channels: (1) Flush+Reload, (2) Flush+Flush, and (3) Prime+Probe.

During a Flush+Reload attack, both victim and attacker share same address space, for example, by sharing libraries. To begin, the data is flushed from the cache. Afterward, the attacker goes into an idle state until the victim runs and accesses the data. A final step would be to reload the cache line and measure the access latency. The victim can access the data speculatively, the line in the cache containing the data will appear, and the reload operation will be faster for that line, revealing the victim's behavior. The attacker can observe a longer access time if the victim hasn't accessed the cache line.

In Flush+Flush attacks, the attacker does not access memory directly and no cache misses are caused for the attacker program. This method exploits the duration of the flush instruction: in case a cached address already exists, flushing takes longer. Because Flush+Flush does not cause any cache hits or misses, it cannot be detected by cache contention detection mechanisms.

With Prime+Probe attacks, the attacker attempts to detect which cache sets the victim's program uses. Attackers launch spy programs to monitor a victim's cache contention. As a first step (Prime), the attacker's program fills multiple cache sets. The attacker then waits, letting the victim's program run. Then, during the Probe phase, the attacker monitors its own filled cache sets. As this occurs, the attacker observes how much time it takes to load each set of its data that was already primed. When cache sets are evicted by the victim, the attacker will be able to observe this when data is delayed in being fetched from the cache. Throughout the paper, we use the term *initialization* to describe the cache lines that are either primed by Prime+Probe or flushed by Flush+Reload and Flush+Flush techniques.

2.3 Detection of Speculative Execution Attacks

In the previous section we discussed how speculative execution attacks are able to leak arbitrary memory locations from the victim in order to extract secret information. These attacks exploit one of the most important features of modern processors, speculative execution, which means that the vast majority of modern processors in high-performance settings have this vulnerability. To address these issues, a number of defense techniques have been proposed to mitigate speculative execution attacks [2, 12, 19, 30, 31]. The majority of these mitigations result in a high overhead in terms of performance and power consumption. As these techniques are typically enabled for the entire workload execution, they will cause a slowdown even when the system is not at risk. Hence, detecting potential attacks and malicious activities during execution allows the mitigation mechanisms to enable and protect the system only if it is necessary and there is a threat. An ideal detector should satisfy a specific set of conditions in order to be effective:

- *Condition 1*: The detector needs to be **accurate** with a low false positive rate and 0% false negatives in order to provide a strong security guarantee.
- *Condition 2*: Attackers can use a variety of methods to break the system and detection mechanisms deployed on a system [15]. An ideal detector needs to be **robust** to evasive attacks and detect data leaks.
- *Condition 3*: An ideal detector raises an alert before the attacker has a chance to extract secret information. A **fast** detection mechanism enables the system to protect sensitive information without any risks of data reconstruction by the attacker.
- *Condition 4*: An **efficient** detector is crucial in order to motivate its deployment in a system along with the mitigation techniques. An ideal detector has low overheads in terms of performance, power and area. Also, a low false positive rate helps the system to avoid unnecessary restrictions and maintain high efficiency.

3 MOTIVATION AND EVASIVE SPECTRE ATTACKS

In this section, we introduce two new ways to break PerSpectron [20], a state-of-the-art detector using Machine Learning (ML) and microarchitectural features of the execution. PerSpectron collects samples of microarchitectural features that are highly correlated with the malicious activities. The PerSpectron methodology starts by building a dataset of fixed-interval samples (e.g., each 10K instructions) and feeds this dataset to a heuristic algorithm like neural networks to classify malicious and benign programs. We use the same setup and methodologies used by the authors of PerSpectron (see Section 7 for more details). In this work, we aim to show that ML-based detectors can be vulnerable to evasive attacks. We propose two new attacks using for this purpose: (1) Expanded-Spectre, and (2) Benign-Program-Spectre attacks.

3.1 Expanded-Spectre Attack

PerSpectron exploits ML to detect the footprints of malicious activity by monitoring microarchitectural statistics. One way to defeat this methodology is to change the footprint of a Spectre attack without compromising the attack's success. We designed a new version of Spectre by expanding the first phase of Spectre, *branch mistraining*, since the changes of this phase are more persistent compared to other phases. We use benign software structures to construct a branch mistraining phase without disrupting the changes of this phase. There are two variants of this attack: (1) insertion of NOPs, and (2) insertion of memory delay instructions. In former variant, NOPs are inserted into the branch mistraining code to stretch the execution in a way that tricks the detector to classify the attack as benign. In the latter version, we insert memory accesses instead of additional NOPs. We use the Fisher-Yates shuffle algorithm [7] to determine the order of memory accesses in order to increase the chances of cache misses and changing the footprint of the attack. Our evaluation shows that PerSpectron is not robust against our Expanded-Spectre attacks, as the accuracy drops from 99% to 14.34% and 54.89% for NOP insertion and memory-delay insertion, respectively (See Table 4 for details). Also, we show that retraining the model with Expanded-Spectre does not help PerSpectron to reach an acceptable detection accuracy.

3.2 Benign-Program-Spectre Attack

As previous works classify programs as benign or malicious, one counterexample that could demonstrate the limited benefits of ML-based detectors would be to build a scenario that constructs a malicious program from components of benign programs. If a malicious attack could be built with these pieces, then a detector might be fooled by this new application. To demonstrate this, we detail the construction of another variant of evasive Spectre in this section. We design Benign-Program-Spectre attack by linking all phases of a Spectre attack by finding regions of the code in benign programs with the same functionality of Spectre and Prime+Probe:

- **Branch Mistraining Phase**: For this phase, we require a gadget which has a loop with a large number of iterations. We also need to make sure this branch has a collision with the victim's branch in Pattern History Table (PHT). The collision can be achieved either by the insertion of NOPs to change PC of the branch, or by moving the gadget inside a buffer and repeating the attack until a collision is found [13].
- **Cache Initialization Phase**: For this phase, we need a gadget in a benign program that primes at least two cache sets (in order to leak one bit of the secret). Table 5 shows that there are many regions in the benign programs priming at least 2 cache sets.
- **Secret Recovery Phase**: For this phase we use the same gadget found for the initialization phase, however, we need to measure the latency of accesses to different cache lines in order to reconstruct the data leak. The measurement can be achieved either by inserting `rdtscp()` to the gadget or using another thread with spin-loop with a shared variable and counting the latency of the gadget execution.

After extracting all these gadgets we can link them to launch a successful attack. We use ELFies [24] to extract the attack gadgets from MCF application in SPEC CPU2006 [9] and build our Benign-Program-Spectre. Our evaluation shows that PerSpectron's accuracy to detect Benign-Program-Spectre is 12.27%, which means that 77.73% of the time, they are unable to detect malicious attacks.

Our evasive Spectre attacks show a strong indication that ML-based detectors might not be sufficiently robust (Detection *Condition 2*, Section 2.3). Also, these detectors do not have a 0% false negative rate which means the attackers have the opportunity to exploit the small percentage of false negatives and break the system (Detection *Condition 1*). Another item to note is that most of the detectors do not provide strong guarantees to detect data leaks fast enough before the attackers can recover the secret (Detection *Condition 3*). In this paper, we propose Spectify that provides an efficient detection mechanism (Detection *Condition 4*) that is robust against evasive Spectre attacks and fast enough to enable the system to employ appropriate protections in time. Also, Spectify detects the source of memory data leaks instead of only classifying the execution as benign or malicious. This method demonstrates 100% detection accuracy and negligible false positive rate. Hence, Spectify fulfills all conditions of an ideal detector.

4 THREAT MODEL

The main goal of Spectify is to *detect* the speculative execution attacks that aim to leak the memory contents that they are not typically allowed to access. We assume the attacker uses cache based side-channels to leak the information using existing cache based attacks like Flush+Reload, Flush+Flush, and Prime+Probe. Mainstream attacks prime an array (a consecutive range of memory locations) and probe the same locations to extract the secret. However, our detector assumes these locations do not need to be consecutive, as long as the attacker knows which memory locations have been primed. Spectify raises an alert and reports the actual data leaks before the attacker gets a chance to probe the side-channel, unlike other works that use machine learning to just warn about the malicious activities without a definitive report of whether a memory data leak has happened or not [20]. The characteristics of the memory data leaks for Spectify are discussed in Section 5.

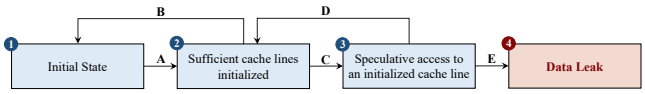

Figure 1: Spectify Detection Mechanism. Transition E makes sure that the reported data leaks are misspeculatively accessed by the victim and a potential attacker can reconstruct it later.

5 SPECTIFY DETECTION METHODOLOGY

In Spectify, we aim to track the required phases of the attack and detect the actual memory leaks. Spectify reports memory leaks with three main features:

- **Feature 1**. The data leak has occurred during the speculation window of a branch that has resulted in misprediction and squashing the memory access.
- **Feature 2**. The data leak is from a memory location that has been primed from previous processes on the same hardware (potentially an attacker).
- **Feature 3**. A potential attacker has the ability to reconstruct the state of the leaked data. The attacker needs to initialize a sufficient number of memory locations[2] and later probe the same locations in order to extract the secret. If more than one of the initialized memory locations are leaked during the process execution then the attacker will not have the ability to reconstruct the key later during the probe phase.

The data leaks that we report are unintended memory accesses that other processes might have some knowledge about their initial state (e.g., that location has been accessed or flushed before). Spectify reports theses data leaks as soon as possible (at the end of the execution of the current process before context switch).

Figure 1 shows an overview of Spectify's detection mechanism. Here, we explain the states and transitions of this technique:

(1) $\xrightarrow{A}$ (2): We start from an initial state and transition A happens if a sufficient number of cache lines are initialized and the system is ready from the attacker's perspective to run the victim. For example for a successful Spectre attack with Flush+Reload, the attacker needs to flush 256 memory locations in order to extract one byte of secret information. The threshold of the sufficient number of

[2]For example, Listing 1 flushes 256 cache lines to use Flush+Reload. However, flushing only two cache lines is enough to extract one bit of secret.

initialized cache lines is *configurable* in our design. We configure this threshold to 2 in our experiments (Section 8) which is the most conservative threshold since the attacker can leak one bit of secret information by initializing only two cache lines.

❷ $\xrightarrow{B}$ ❶: The transition B between state ❷ and state ❶ happens if the number of initialized number of cache lines falls below the configured threshold.

❷ $\xrightarrow{C}$ ❸: The transition C happens if a sufficient number of cache lines are initialized by previous processes and the current process (potentially a victim) speculatively accesses an initialized memory (the attacker will be able to reconstruct the access behavior).

❸ $\xrightarrow{D}$ ❷: The transition D between state ❸ and ❷ occurs when the speculative access resolves in correct the path (i.e., no branch misprediction and squash of the speculative access). This means that the speculative access could not be an access to a secret. Transition D can also happen when multiple speculative accesses are squashed in a way that multiple initialized cache lines are touched and the attacker loses the ability to reconstruct the individual accesses.

❸ $\xrightarrow{E}$ ❹: The transition E results in data leak detection and it happens if the speculative access in state ❸ was squashed (misprediction of a branch) and the next process (potentially an attacker) is able to reconstruct the access behavior. In other words, the speculative and squashed access has changed the state of only one of the initialized cache lines.

SPECTIFY reports the data leaks before the attacker can probe the initialized cache lines. This allows the system to enable the required protection to prevent recovering the misspeculatively accessed information by any potential attacker. Our detector provides the accurate information about the data leak, for example, the accessed memory location, the mispredicted branch, and the timing of the data leak. Note that, our detector does not aim to detect an end-to-end attack (unlike most other detectors [8, 22]). The goal of our detector is to find all the data leaks that are ready to be recovered from an attacker's perspective. Hence, even if our detector detects data leaks during the execution of benign programs, it means that these programs are leaking information unintentionally but that the system still needs to protect these data leaks.

6 SPECTIFY MICROARCHITECTURE

In this section, we discuss the microarchitecture implementation of our detection mechanism described in Section 5. We perform all the steps and transitions of the detector by monitoring the key components in the architecture that are involved in shaping an Spectre attack. Figure 2 shows an overview of the microarchitecture of SPECTIFY and the new structures added to the processor. As shown in Figure 2, there are three types of information flow that is tracked in our implementation: (1) the Initialization Flow happens during the normal execution of a workload and gathers the information about which cache lines and memory locations are initialized (Section 6.1), and (2) the Squash Flow tracks when branch misprediction and Reorder Buffer (ROB) squashes happen (Section 6.2). The (3) Context-Switch Flow happens only when a context-switch in the system occurs and the detector evaluates its current information to detect potential data leaks of the old process (Section 6.3).

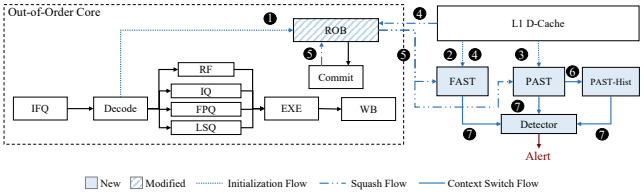

Figure 2: SPECTIFY Microarchitecture.

Modified ROB

PC	Branch	Unresolved	Set Number	Mem. Addr
462730	1	1	10	678350
462734	0	0	0	0
462738	1	1	5	0
…				

Flush and Access Status Table (FAST)

Mem. Addr	Valid	Current Frame	Squashed
678350	1	0	1
678430	0	0	0
679200	1	0	0
…			

Prime and Access Status Table (PAST)

Set	Way #1		…	Way #N		Squashed
	Accessed	Valid		Accessed	Valid	
0	1	0		1	0	0
1	0	0		0	0	1
2	1	0		1	0	0
…						
M						

Prime and Access Status Table History (PAST-Hist)

Set	Way #1	…	Way #N
	Primed		Primed
0	0		0
1	0		0
2	0		0
…			
M			

Figure 3: ROB is modified to store the information about branches (unresolved, and which cache sets and memory addresses they have accessed during their speculation window). FAST keeps the state of all memory addresses that have been flushed and whether they are still valid and squashed by a branch. PAST stores the state of all cache lines if they have been accessed by the current process and squashed. PAST-Hist is a shadow of PAST containing the state of PAST from previous processes. M is the number of L1D cache sets and N is the number of ways.

Figure 3 shows an example state of the new structures in our implementation, and Table 1 contains the details of the events and the actions processed in our proposed design.

6.1 Initialization Flow

The initialization flow of the detector gathers the essential information from the execution of the running process to realize if a process is performing the initialization step of a speculative execution attack (See Listing 1). When a branch is inserted to the ROB, we mark extra bits in the ROB to show if the instruction is a branch and yet unresolved (step ❶ in Figure 2). For the `Trained` bit in the ROB, we lookup the Branch Prediction Unit (BPU) for its confidence in the prediction (for example, the saturating counter assigned to each entry being above a threshold). We reset the `Unresolved` bit whenever the correct path of the branch is known.

In step ❷ and ❸, we update two new tables that are responsible to keep track of the initialized cache lines and memory locations:

- *Flush and Access Status Table (FAST).* For the initialization step of FLUSH+RELOAD and FLUSH+FLUSH, the attacker needs to flush a sufficient number of memory locations. Each flush request to L1 D-Cache inserts a new entry to FAST and

Event	Action
❶ Branch `B` entering ROB	ROB[B].Branch = 1 ROB[B].Unresolved = 1
❷ Cache flush with `addr` happens	FAST[addr].Valid = 1 FAST[addr].CurrentFrame = 1
❸ Cache access to set `M` and way `N` happens	PAST[M][N].Valid = 1 PAST[M][N].Accessed = 1

❹ Cache miss with `addr` and set `M` happens	if FAST[addr] exists:
	  FAST[addr].Valid = 0
	for all entries B in ROB:
	  if ROB[B].Branch == 1 && ROB[B].Unresolved == 1:
	  ROB[B].MemAddr = addr
	  ROB[B].SetNumber = M

❺ Squash happens	for all entries B being squashed in ROB:
	  if ROB[B].Branch == 1 && ROB[B].Unresolved == 1:
	  PAST[ROB[B].SetNumber].Squahed = 1
	  if ROB[B].MemAddr exists in FAST:
	  FAST[ROB[B].MemAddr].Squashed = 1

❻ After context switch	for all entries E in FAST:
	  if FAST[E].Valid == 1:
	  FAST[E].CurrentFrame = 0
	for all set X and way Y entries in PAST-Hist:
	  A = PAST[X][Y].Accessed
	  V = PAST[X][Y].Valid
	  P = PAST-Hist[X][Y].Primed
	  PAST-Hist[X][Y] = (P && !A) || (V && A)
	Reset PAST

❼ Before context switch	FlushedCNT = 0, AccessedCNT = 0
	for all entries E in FAST:
	  if FAST[E].Valid == 0 && FAST[E].CurrentFrame == 1
	  && FAST[E].Squahed == 1:
	  AccessedCNT++
	  else if FAST[E].Valid == 1 && FAST[E].CurrentFrame == 0:
	  FlushedCNT++
	if FlushedCNT >= Min_Flushed && AccessedCNT == 1:
	  Alert = 1
	PotentialCNT = 0, PrimedCNT = 0
	for all set entries S in PAST and PAST-Hist:
	  wayCNT = 0
	  for all way entries W in PAST[S]:
	  if PAST[S][W].Accessed == 1 && PAST[S][W].Valid == 1:
	  wayCNT++
	  if wayCNT == 1 && PAST[S].Squashed == 1:
	  PotentialCNT++
	  wayCNT = 0
	  for all way entries W in PAST-Hist[S]:
	  if PAST-Hist[S][W].Primed == 1:
	  wayCNT++
	  if wayCNT == TotalNumWays:
	  PrimedCNT++
	if PrimedCNT >= Min_Primed && PotentialCNT == 1:
	  Alert = 1

Table 1: SPECTIFY events and actions. `Min_Flushed` and `Min_Primed` are the minimum number of locations to be flushed or primed for a successful attack.

sets the *Valid* and *CurrentFrame* bits. Each entry of FAST is invalidated whenever an access to its memory location happens (step ❹). Note that, the *CurrentFrame* bit shows the flush is done by the current process or previous processes.

- *Prime and Access Status Table (PAST).* For the initialization step of PRIME+PROBE, the attacker needs to prime a sufficient number of cache sets. This means that she needs to access all the cache lines in each cache set in order to be considered as primed. All the accesses to the cache are monitored by the PAST table and the accesses are reflected in the appropriate cache set and cache way. Each access can be invalidated whenever a context switch occurs (see Section 6.3).

L1d Cache	64KB, 64B line, 8-way	LQ/SQ size	32/32 entries
L1i Cache	32KB, 64B line, 4-way	ROB size	192 entries
L2 Cache	2MB, 64B line, 8-way	RF (INT/FP) size	256/256 entries
Branch Predictor	TAGE	FAST size	512 entries
F/D/I/C width	8/8/8/8	PAST/PAST-Hist size	128 entries

Table 2: System Configuration.

Also, whenever a L1 D-Cache miss occurs we update all the unresolved branches in the ROB with the memory address and the cache set number of the cache miss (step ❹). We need this information later in the Squash Flow of the design (Section 6.2). If multiple misses happen during the speculation window of an unresolved branch, we reset the *SetNumber* and *MemAddr* of the branch in the ROB. We do this because the attacker will lose the ability to reconstruct the secret (Feature 3 of a data leak, Section 5).

6.2 Squash Flow

When a branch misprediction occurs the Commit stage sends a signal to the ROB to squash all the instructions after the branch (step ❺). For any branch being squashed, if the *SetNumber* and *MemAddr* are set then we set the *Squashed* bit for the corresponding entries in PAST and FAST tables. The detector will use this bit to make sure that it only reports squashed memory accesses (see Section 6.2).

6.3 Context-Switch Flow

When a context switch happens, the system changes the state of flushed and primed locations in FAST and PAST tables [4] and affects the initializations that a potential attacker has done [2]. After a context switch, for all the entries in FAST that are still flushed ($Valid == 1$) we reset the *CurrentFrame* bit, which shows that the flush is done by the previous processes (a potential attacker). Also, we add a new table, called PAST-Hist, which is a shadow of PAST and contains the primed and access status of previous processes. PAST-Hist has a *Primed* bit for each cache line that shows if it has been primed. PAST-Hist is updated after each context switch. It will keep its *Primed* bit state for each line if it has not been accessed ($Accessed == 0$ in PAST), but if the cache line is accessed then we set the *Primed* bit to 1 only if the access is valid and has not occurred during the context switch ($Valid == 1$ in PAST) (step ❻).

In step ❼ and before a context switch, the detector evaluates the state of FAST, PAST, and PAST-Hist tables to report memory data leaks. The detector reports data leaks that satisfy all features explained in Section 5. If there is an entry in FAST or PAST that its *Squashed* bit is set to (*Feature 1*) and it is a valid flushed location ($Valid == 1$ in FAST, *Feature 2*) or a primed cache set ($Primed == 1$ for all its ways in PAST-Hist, *Feature 2*) then we report it as data leak only if it is the only entry satisfying these conditions (*Feature 3*). Table 1 shows the details of the detector's logic.

7 EXPERIMENTAL SETUP

Simulation. We implement SPECTIFY in gem5 [1] in syscall emulation mode. We use CACTI 6.5 [16] to estimate the power and area overheads of SPECTIFY over our baseline out-of-order core. Table 2 shows the detailed system parameters used for evaluation.

Scenario	Description
Benign	SPEC CPU2006 programs (list of SPEC programs in Table 5)
Spectre	Spectre V1 + Spectre V2
Expanded-Spectre-N1	Expanded-Spectre with insertion of 50 NOPs during each iteration in branch mistraining
Expanded-Spectre-N2	Expanded-Spectre with insertion of 100 NOPs during each iteration in branch mistraining
Expanded-Spectre-M	Expanded-Spectre with insertion of 30 memory delay instructions during each iteration in branch mistraining
Benign-Program-Spectre	Spectre constructed by MCF slices based on Prime+Probe
Training-Set#1	Benign + Spectre
Training-Set#2	Training-Set#1 + Expanded-Spectre-N1

Table 3: Scenario Summary.

Test Scenario	PerSpectron	Specify	
	Accuracy	Avg. #Alerts	Accuracy
Benign	99.10%	11	99.98%
Spectre V1	99.61%	1	100.0%
Spectre V2	98.67%	1	100.0%
Expanded-Spectre-N1	14.34%	1	100.0%
Expanded-Spectre-N2 ‡	61.25%	1	100.0%
Expanded-Spectre-M	54.89%	1	100.0%
Benign-Program-Spectre	12.27%	1	100.0%

‡ in this case, PerSpectron is retrained with the Training-Set#2 dataset (see Table 3 for details).

Table 4: Comparison of detection accuracy for PerSpectron and Specify.

Application	#frames	avg. primed	#min. 2 sets primed ❶ $\xrightarrow{A}$ ❷	#SANS ❷ $\xleftarrow{D}$ ❸	#SASNP ❸ $\xleftarrow{D}$ ❸	#SASMA ❷ $\xleftarrow{D}$ ❸	#Data Leaks ❸ $\xrightarrow{E}$ ❹
401.bzip2	38136	1.04	6667	5122	81	5035	4
403.gcc	151771	2.29	53186	43097	1322	675494	11
410.bwaves	55255	15.05	46278	5800	10	6205	3
416.gamess	26720	0.28	1205	83	291	12016	10
429.mcf	217673	7.23	106053	6509	4246	9330631	52
434.zeusmp	32763	6.51	19760	10757	700	215863	40
436.cactusADM	60729	0.54	7407	7367	0	52	0
444.namd	277321	0.01	244	235	3	42	0
445.gobmk	48742	1.14	4074	719	668	384957	1
450.soplex	128519	5.60	39411	4745	2223	2734828	10
462.libquantum	72327	9.37	26315	16847	0	140	0
471.omnetpp	85982	1.25	22715	1258	2164	1275119	2

Table 5: Detailed statistics from running Specify. Three statistics showing different reasons for transition D between state ❸ and state ❷: (1) SANS: Speculative Access but Not Squashed, (2) SASNP: Speculative Access and Squashed but Not Primed, and (3) SASMA: Speculative Access and Squashed but Multiple Accesses.

Benchmarks. For benign programs, we use SPEC CPU2006 benchmark suite [9]. We generate ELFies [24] as executable representative with a region size of 100M instructions. For malicious programs, we run Spectre V1 and Spectre V2 alongside our evasive Spectre attacks. Table 3 shows different scenarios including the details of our evasive Spectre attacks.

Running PerSpectron. We implement PerSpectron with the FANN C library [23] using microarchitectural features, a 10k instruction sampling rate and the single-layer perceptron neural network originally used [20]. Datasets from Table 3 were used for evaluation, where 66% of the data is used for training and the rest for testing.

8 EXPERIMENTAL RESULTS

8.1 Security Evaluation

To compare our work with the state-of-the-art ML based detection, we use the scenarios in Table 3 to evaluate the accuracy of Specify and PerSpectron. We do not report the false positive rates since the test scenarios consist of only benign or only malicious programs. For all cases we used the Training-Set#1 dataset to train PerSpectron's model. However, we retrained the model with Training-Set#2 only when testing the Expanded-Spectre-N2 scenario.

Table 4 shows the accuracy of Specify compared to PerSpectron. We split the execution of programs into multiple *frames* where each frame represents a period of execution between two context switches. We assume that context switches occur every 10ms [6]. Also, we emulate the impact of a context switch by randomly clearing the primed state of 50% of entries in PAST since more than half of the cache is accessed by the system during a context switch [4]. At the end of each frame we evaluate whether there has been a data leak in the previous frame or not. We consider the most conservative situation where the attacker only needs to prime two cache sets or flush two memory locations to leak one bit of secret information [18].

As shown in Table 4, Specify has 100% detection accuracy for malicious scenarios and is able to detect the exact data leak of the attack before the recovery phase. Specify shows 99.98% accuracy in detection of benign programs. However, the data leaks detected by Specify are all actual memory data leaks that can be recovered later and it means the system needs to be cautious in these situations and employ appropriate protections. This also confirms the possibility of our Benign-Program-Spectre while running benign applications on the system. Table 4 shows the detection accuracy of PerSpectron as well. While PerSpectron shows high accuracy for normal benign programs and Spectre attacks, it fails to detect our evasive Spectre attacks in different scenarios. We also retrained PerSpectron's model with the Expanded-Spectre-N1 dataset, however, we were able to break the new model by adding more NOPs to the Expanded-Spectre attack (Expanded-Spectre-N2 test).

Table 5 shows the detailed statistics of running SPEC CPU2006 programs with Specify. We report the number of frames (#frames) and the number of frames that at least two cache sets are primed (i.e., ❶ $\xrightarrow{A}$ ❷ transition has been seen, Section 5) and the system is ready to leak information (i.e., ready for the ❷ $\xrightarrow{B}$ ❸ transition). We also show the average number of cache sets primed and the

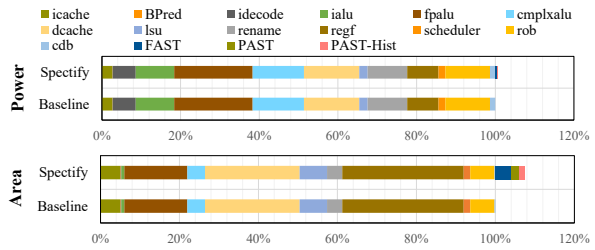

Figure 4: Total power and area consumption normalized to the out-of-order baseline processor. The new structures are in capital letters in the legend.

number of data leaks per frame. The other three statistics show the results for situations where a data leak has not occurred: (1) #SANS counter shows the number of speculative accesses that did not squash, and therefore cannot leak data. (2) The #SASNP counter shows the number of speculative accesses that have been squashed but have not resulted in a data leak since the accesses did not access primed cache sets. (3) The #SASMA counter shows the number of speculative accesses that have been squashed but did not result in a data leak since they accessed multiple cache sets during the speculation window (this means that a potential attacker will not be able to reconstruct the data at a later point).

8.2 Efficiency Evaluation

Since SPECTIFY operates in parallel with the core operations and is off the critical path of the execution, we do not expect a runtime performance reduction.

Figure 4 shows the total power and area consumption of SPECTIFY normalized to the out-of-order baseline processor. Overall, SPECTIFY has an average power overhead of 0.66% that comes from the implementation of FAST, PAST, and PAST-Hist structures, which are direct-mapped memory structures requiring little power. Also, the area overhead of new structures is 7.3% over the baseline core.

9 RELATED WORKS

ML-based detectors. Machine Learning is widely used in the literature to classify running applications into benign and malicious categories. These techniques train a classifier based on a labeled dataset from runtime characteristics of benign programs and attacks. Most detectors use hardware performance counters (HPCs) [5, 21, 22, 25, 28, 29, 32]. For example, [22] proposes a detector that uses three different trained models of HPC data and classifies applications based on a majority vote. While HPCs provide a basic dataset for ML-based anomaly detection, the limited number of HPCs result in a degradation of accuracy and less robustness of detectors. Only a few hardware events can be tracked in parallel due to the limitation of available physical registers (typically 4-8 registers). To overcome this limitation, PerSpectron [20] uses microarchitecture-level statistics directly to capture attack signatures through highly-correlated features. They propose an efficient microarchitecture for more accurate classification of running applications. Most detectors in this class can be vulnerable to evasive attack via bandwidth reduction [20]. However, they can address this issue by employing RHMD [11]. RHMD uses multiple independent classifiers and chooses one of the classifiers randomly to defend against reverse engineering attacks. While this technique is effective against bandwidth reduction evasion, it cannot address our evasive Spectre attacks. Also, it is challenging for ML-based detectors to detect our Benign-Program-Spectre since they use a labeled dataset of benign and malicious programs.

Cache contention based detectors. A class of detectors track cache contention and try to find malicious access patterns of the attacker to cache blocks [3, 8, 10]. Cyclone [8] introduces a common feature in contention-based attacks, like PRIME+PROBE and FLUSH+RELOAD, called cyclic interference to efficiently implement contention tracking. However, the cyclic interference feature can occur in benign programs as well, and Cyclone uses an ML technique to classify each cyclic interference to be benign or malicious. This means that this technique can be vulnerable to evasive attacks that can miss-classify malicious activities. Also, Cyclone is not as fast as our detector since the probe phase of the attack needs to occur in order to form a cyclic interference, while we report data leaks before the probe phase. Note that since FLUSH+FLUSH attacks do not form a cyclic interference pattern Cyclone will not be able to detect this class of attacks.

Mitigations. Many defenses have been proposed to mitigate the speculative execution attacks targeting caches. SafeSpec [12] and InvisiSpec [30] hide transient loads from the cache hierarchy and use a separate buffer for speculative accesses before they become non-speculative and commit changes to the cache. CEASER [26] is another mitigation that keeps the cache in a randomized and obfuscated state that prevents the attacker from reconstructing the secret. This technique significantly reduces the attack's success via cache side-channels. STT [31] and DOLMA [19] try to mitigate Spectre attacks from the source of the leak and prevent data leaks via any side-channels. As our detector reports data leaks before the recovery phase of attack, it allows these defenses to be enabled only if it is required. This can be more efficient to protect the actual data leaks and problematic speculative branches.

10 CONCLUSION

Modern CPUs are vulnerable to speculative execution attacks and it is crucial to adopt effective and efficient protections against these attacks. But, the performance overheads of recent always-on mitigations slows down the execution of CPUs. An effective detection mechanism can help the system to employ the appropriate protections whenever there is a risk and potentially a memory leak.

In this work, we propose SPECTIFY that takes advantage of microarchitecture-level information to track the attack phases in order to find actual memory data leaks before a potential attacker finds a chance to reconstruct the data leak. We demonstrate 100% detection accuracy for known speculative execution attacks and evasive attacks that the state-of-the-art ML based detectors are unable to detect. SPECTIFY has no performance slowdowns with low overheads in terms of power and area.

ACKNOWLEDGMENTS

Support provided by Singapore NRF grant NRF2018NCR-NCR002.

REFERENCES

[1] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH computer architecture news* 39, 2 (2011), 1–7.

[2] Thomas Bourgeat, Ilia Lebedev, Andrew Wright, Sizhuo Zhang, and Srinivas Devadas. 2019. Mi6: Secure enclaves in a speculative out-of-order processor. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 42–56.

[3] Jie Chen and Guru Venkataramani. 2014. Cc-hunter: Uncovering covert timing channels on shared processor hardware. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 216–228.

[4] Yun Chen, Lingfeng Pei, and Trevor E Carlson. 2021. Leaking Control Flow Information via the Hardware Prefetcher. *arXiv preprint arXiv:2109.00474* (2021).

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

[6] Dror G Feitelson and Larry Rudolph. 1995. *Job Scheduling Strategies for Parallel Processing: IPPS'95 Workshop, Santa Barbara, CA, USA, April 25, 1995. Proceedings*. Vol. 949. Springer Science & Business Media.

[7] Ronald Aylmer Fisher and Frank Yates. 1953. *Statistical tables for biological, agricultural and medical research*. Hafner Publishing Company.

[8] Austin Harris, Shijia Wei, Prateek Sahu, Pranav Kumar, Todd Austin, and Mohit Tiwari. 2019. Cyclone: Detecting contention-based cache information leaks through cyclic interference. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 57–72.

[9] John L Henning. 2006. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News* 34, 4 (2006), 1–17.

[10] Casen Hunger, Mikhail Kazdagli, Ankit Rawat, Alex Dimakis, Sriram Vishwanath, and Mohit Tiwari. 2015. Understanding contention-based channels and using them for defense. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 639–650.

[11] Khaled N Khasawneh, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Lei Yu. 2017. RHMD: Evasion-resilient hardware malware detectors. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. 315–327.

[12] Khaled N Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2019. Safespec: Banishing the spectre of a meltdown with leakage-free speculation. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[13] Ofek Kirzner and Adam Morrison. 2021. An analysis of speculative type confusion vulnerabilities in the wild. In *30th USENIX Security Symposium (USENIX Security 21)*. 2399–2416.

[14] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. 2019. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1–19.

[15] Congmiao Li and Jean-Luc Gaudiot. 2020. Challenges in detecting an "evasive spectre". *IEEE Computer Architecture Letters* 19, 1 (2020), 18–21.

[16] Sheng Li, Ke Chen, Jung Ho Ahn, Jay B. Brockman, and Norman P. Jouppi. 2011. CACTI-P: Architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques. In *ICCAD: International Conference on Computer-Aided Design*. 694–701.

[17] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. 2018. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*. 973–990.

[18] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. 2015. Last-level cache side-channel attacks are practical. In *2015 IEEE symposium on security and privacy*. IEEE, 605–622.

[19] Kevin Loughlin, Ian Neal, Jiacheng Ma, Elisa Tsai, Ofir Weisse, Satish Narayanasamy, and Baris Kasikci. 2021. {DOLMA}: Securing Speculation with the Principle of Transient Non-Observability. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*.

[20] Samira Mirbagher-Ajorpaz, Gilles Pokam, Esmaeil Mohammadian-Koruyeh, Elba Garza, Nael Abu-Ghazaleh, and Daniel A Jiménez. 2020. Perspectron: Detecting invariant footprints of microarchitectural attacks with perceptron. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1124–1137.

[21] Maria Mushtaq, Ayaz Akram, Muhammad Khurram Bhatti, Vianney Lapotre, and Guy Gogniat. 2018. Cache-Based Side-Channel Intrusion Detection using Hardware Performance Counters. In *CryptArchi 2018-16th International Workshops on Cryptographic architectures embedded in logic devices*.

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

[23] Steffen Nissen et al. 2003. Implementation of a fast artificial neural network library (fann). *Report, Department of Computer Science University of Copenhagen (DIKU)* 31, 29 (2003), 26.

[24] Harish Patil, Alexander Isaev, Wim Heirman, Alen Sabu, Ali Hajiabadi, and Trevor E Carlson. 2021. ELFies: executable region checkpoints for performance analysis and simulation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 126–136.

[25] Mathias Payer. 2016. HexPADS: a platform to detect "stealth" attacks. In *International Symposium on Engineering Secure Software and Systems*. Springer, 138–154.

[26] Moinuddin K Qureshi. 2018. CEASER: Mitigating conflict-based cache attacks via encrypted-address and remapping. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 775–787.

[27] Chao Su and Qingkai Zeng. 2021. Survey of CPU cache-based side-channel attacks: systematic analysis, security models, and countermeasures. *Security and Communication Networks* 2021 (2021).

[28] Han Wang, Hossein Sayadi, Avesta Sasan, Setareh Rafatirad, and Houman Homayoun. 2020. Hybrid-shield: Accurate and efficient cross-layer countermeasure for run-time detection and mitigation of cache-based side-channel attacks. In *Proceedings of the 39th International Conference on Computer-Aided Design*. 1–9.

[29] Xueyang Wang, Charalambos Konstantinou, Michail Maniatakos, and Ramesh Karri. 2015. Confirm: Detecting firmware modifications in embedded systems using hardware performance counters. In *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 544–551.

[30] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. 2018. Invisispec: Making speculative execution invisible in the cache hierarchy. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 428–441.

[31] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W Fletcher. 2019. Speculative taint tracking (stt) a comprehensive protection for speculatively accessed data. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 954–968.

[32] Tianwei Zhang, Yinqian Zhang, and Ruby B Lee. 2016. Cloudradar: A real-time side-channel attack detection system in clouds. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 118–140.