HIDFIX: Efficient Mitigation of Cache-based Spectre Attacks through Hidden Rollbacks

Arash Pashrashid, Ali Hajiabadi, and Trevor E. Carlson

National University of Singapore

***Abstract*—Mitigating Spectre attacks in modern systems is a challenging task for CPU vendors as they need to provide comprehensive protection while maintaining high efficiency. One common solution is to adopt always-on mitigation strategies to prevent all speculative data leaks. However, these solutions incur prohibitive performance overheads as they limit the benefits of speculative execution, the main performance enabler of modern processors. Additionally, recent attacks have demonstrated the limitations of many existing defenses.**

Combining side-channel attack (SCA) detectors with mitigation strategies is a promising direction to achieve efficient and selective mitigation of Spectre attacks. In this work, we enumerate the combinations of state-of-the-art detection and mitigation strategies and present both new attacks as well as the potential risks of such detection/mitigation combinations. The result is the HIDFIX methodology, an efficient mitigation for cache-based Spectre attacks, that addresses the security limitations of prior work. We show that HIDFIX has a near-zero performance overhead for all evaluated applications. HIDFIX rollbacks the misspeculated data leaks in a timely manner, before an attacker has the chance to infer the victim's sensitive data. We demonstrate that HIDFIX is more secure compared to prior cache-based Spectre defenses, and moreover, it does not introduce new side effects that might enable an attacker to observe secret dependent changes in the system.zb

I. INTRODUCTION

Modern processors have been equipped with numerous techniques to speed up performance over the past few decades. The main performance enabler of these processors is *speculative execution* [1]. Speculative execution increases the performance in cases where the processor faces long latency control flow decisions, as it predicts the future path to execute while waiting for the correct path to resolve. However, Kocher et al. revealed the Spectre vulnerability in 2018 [2], as a fatal security flaw in modern processors which affects the majority of computing devices. Spectre attacks convince privileged victim applications to speculatively transmit sensitive data into a persistent side-channel, the most common being the data cache.

There have been extensive, recent studies on a variety of methods to mitigate Spectre-style attacks [3]–[7]. The majority of these solutions propose **always-on** defenses in an attempt to comprehensively mitigate speculative execution attacks. However, these defenses sacrifice performance to gain high levels of security. The most comprehensive defenses restrict the speculative execution of instructions to prevent data leaks through any potential side-channel. However, restriction-based solutions incur prohibitive performance overheads, diminishing the benefits of speculative execution [3], [4], [8]–[11]. To reduce performance overheads, some solutions focus on securing the cache side-channel, as it is the most practical and visible channel [12]–[14]. Cache randomization [5], invisible speculation [7] (*i.e.*, applying changes to the cache only if the instructions become non-speculative), and undoing misspeculated cache changes [6] are examples of protecting caches against Spectre attacks. Unfortunately, these always-on cache protections can incur performance overheads of up to 80%, and moreover, they are either proven to be vulnerable to recent attacks [15], [16], or provide limited protection (*e.g.*, cache randomization cannot protect against memory address contention attacks like Flush+Reload). Isolation (*e.g.*, cache partitioning [17]) and cache access obfuscation (*e.g.*, randomizing the cache access delays [18]) are other solutions to prevent attackers from extracting the victim's sensitive information. While always-on isolation and obfuscation of all running applications are secure, they can result to high performance overheads.

A promising direction to achieve a balanced trade-off between performance and security is deploying side-channel attack (SCA) detectors to selectively enable an appropriate defense. Existing methodologies that detect cache-based Spectre attacks can be categorized into three different classes: (1) machine learning (ML) based detectors collect runtime characteristics of the benign and malicious applications to train a model to classify benign and malicious programs [18]–[24]. (2) Another class of detectors rely on detecting a common property found in existing cache attacks – cyclic interference events on a specific cache location or memory address [25]–[27] (*i.e.*, existing cache attacks result in Attacker$\rightarrow$Victim$\rightarrow$Attacker interference patterns for a specific resource as a common behavior). In addition, (3) prior work [28] proposed a direct analysis of microarchitectural changes of a successful Spectre attack to detect potential data leaks through data caches. Among these solutions, ML-based detectors have been shown to be vulnerable to evasive attacks [28]. However, the later approaches are assumed to be viable options to provide the level of security required, since no vulnerability is reported by prior work. In this work, we will demonstrate the risks of using these solutions.

The goal of this work is to provide stronger security guarantees compared to prior work with minimal performance loss. To achieve this goal, we co-design detection and mitigation strategies and allow modern processors to safely continue to benefit from full potentials of speculative execution.

First, we enumerate the combinations of current detection and mitigation strategies. During this process, we demonstrate the limitations of Cyclone [26], as the state-of-the-art cyclic interference detector, and launch a successful attack without performing cyclic interference events (BENIGNINTERFERE ATTACK). Moreover, we show that even an improved version of the cyclic interference detectors is not timely enough to combine with any mitigation strategy to prevent successful secret extraction (SINGLEPROBE ATTACK). The final option that we consider is using the state-of-the-art detectors [28] or even an ideal detector (being accurate, robust, and timely) to selectively enable/disable isolation and obfuscation mitigations; as the only existing defenses to be secure and flexible enough to selectively enable their mitigation operations (as adopted in [18], [29]). We present a possible attack (SINGLEPRIME ATTACK) on such designs when protecting against cache-based Spectre attacks; ultimately demonstrating the vulnerabilities of isolation and obfuscation techniques when combined with a detector.

As a solution to these newly discovered attacks, we propose a novel detection and mitigation strategy, called HIDFIX, that directly tracks the accesses to cache locations and memory addresses, and effectively mitigates the misspeculated data leaks; we rollback the misspeculated changes that have been initialized by a potential attacker before the attacker has a chance to recover the victim's sensitive information. We demonstrate that our approach is more secure compared to prior cache-based Spectre mitigations. HIDFIX shows near-zero performance overhead for all tested applications (both benign and malicious applications). In addition, we show that our mitigation does not have any potential side effects that might introduce new side-channel attacks.

The main contributions of this work are:

- Providing an in-depth analysis of combining state-of-the-art detection and mitigation strategies, and demonstrating their potential vulnerabilities through our proposed attacks;
- Proposing a secure detection/mitigation strategy for cache-based Spectre attacks;
- Demonstrating near-zero performance overhead for the SPEC CPU2006 benchmark suite [30] and a variety of Spectre attacks; including our new proposed attacks and evasive attacks proposed by prior work [28];
- Presenting a comprehensive security analysis of our methodology against different potential attacks and showing that HIDFIX does not introduce new side-channels.

Step①: Side-channel initialization

```
 1  for (i = 0; i < 256; i++)
 2      clflush(B[i * 64]); //flushing array B as side-channel
```

Step②: Victim leak and transmit

```
 3  if (x < A_size){ //the branch mistrained to always take the branch
 4      secret = A[x]; //speculative access to secret
 5                     // (x = secret_address - A)
 6      temp = B[secret * 64]; //transmit secret to side-channel
 7  }
```

Step③: Secret extraction

```
 8  for (guess = 0; guess < 256; guess++){
 9      t1 = rdtsc();
10      temp = B[guess * 64]; // reloading array B
11      if (rdtsc() - t1 < CACHE_HIT_THRESHOLD)
12          results[guess] += 1; // the secret hits in the cache
13  }
```

Fig. 1. Spectre-V1 (array bound check bypass) attack via Flush+Reload.

II. BACKGROUND

A. Cache-based Spectre Attacks

Speculative execution attacks, popularized by Spectre-V1 [2], exploit one of the main performance features of modern CPUs – speculative execution. Speculative execution, especially using branch prediction, allows a processor to continue to make progress even though it is waiting for branch instructions to resolve (*i.e.,* determining the correct path). Spectre attacks mistrain the branch predictor to force the victim to execute an incorrect path of the program. Figure 1 shows a vulnerable code. In this example, the attacker convinces the victim's branch predictor to bypass the array bounds check (line 3) and speculatively access an unauthorized memory location, potentially sensitive information (line 4). While the processor discards changes of the incorrect path upon misprediction, various channels can contain persistent changes of misspeculated memory accesses.

The most vulnerable, and practical, side-channel used to conduct Spectre attacks has been the processors' private data caches, while the victim and the attacker execute on the same core. For example, Figure 1 explains a Flush+Reload cache side-channel attack that the attacker first flushes all the cache lines associated with array B (step 1, line 2). After the victim execution and data leak (step 2), it will reload all those cache lines (line 10) and evaluates which cache line has been loaded in the core by the victim (line 11). This evaluation allows the attacker to extract the secret value since it will hit the cache, instead of the expected cache miss. Other cache side-channel mechanisms that we consider in this paper are Prime+Probe and Flush+Flush. Prime+Probe initializes the cache side-channel by priming the cache by its known data. Later the attacker probes all the primed cache lines and extracts the secret value if one of the primed values has been speculatively evicted by the victim. Flush+Flush deploys a similar strategy to Flush+Reload, but for the secret extraction it flushes the cache lines again and extracts the secret value by detecting the timing difference of flushing the secret dependent cache line that has been accessed by the victim.

B. Mitigation and Detection of Spectre Attacks

There have been extensive studies to protect modern processors against speculative execution attacks. The most comprehensive defenses prevent branch-level speculation and restrict the execution of any instruction that might leak sensitive data (*restriction-based defenses*) [3], [4], [8]–[11]. However, these defenses tend to introduce very high-performance overheads (up to 2.3× overhead for some applications in the state-of-the-art work [3]), and still continue to leak data [3].

More efficient defenses aim to only secure the cache side-channel. A class of cache-based defenses implements *invisible speculation* which prevents the speculative changes to the cache until they become non-speculative [7], [31], [32]. However, prior work has demonstrated the vulnerability of invisible speculation against Speculative Interference attacks [15]. Another class of defenses tries to undo the effects of misspeculated cache changes (*undo-based defenses*) in an attempt to improve the performance since the misspeculation events are rare [6]. They are, however, shown to be vulnerable since their undo operations introduce secret dependent timing differences that allow attackers to create new attacks based on new inferred behavior [16]. *Cache randomization* defenses [5], [33] also make it harder for attackers to probe the same initialized cache sets, but they are still vulnerable to memory address contention attacks (*e.g.*, Flush+Reload that initializes the side-channel through flushing a set of specific memory addresses, not priming cache sets).

More resistant cache protections partition and isolate the caches (*isolation-based defenses*) [17] to prevent the attacker to initialize and probe the cache locations touched by the victim. However, the cache isolation techniques introduce high performance overheads since the applications can utilize a smaller partition of the cache. Finally, another potential mitigation is *access delay obfuscation* that randomizes the system frequency to obfuscate the data access timings (*i.e.*, making it harder for the attacker to differentiate between the cache hits and misses [18]). This technique provides very high performance overheads as it limits the benefits of accessing the data that is already present in data caches. Moreover, prior work shows that obfuscating the system frequency only lowers the success rate of the attack, and does not completely block the side-channel [18].

All discussed defenses provide **always-on** protection, even if no data leak or malicious activity happens in the system, resulting in unnecessary overheads. Hence, some prior work focused on the detection of malicious activities to enable proper mitigation only if necessary. A large class of detectors trains machine learning (ML) models to detect the Spectre and cache attacks [18]–[24], [26]. For example, PerSpectron [19] uses the microarchitectural features of the processor to detect various Spectre variants. However, prior work has demonstrated a lack of robustness of ML-based detectors against evasive attacks [28]. Another class of detectors proposes to exploit the common behavior of cache attacks that cause a cyclic interference in processor resources and cache locations [25]–[27]. In this work,

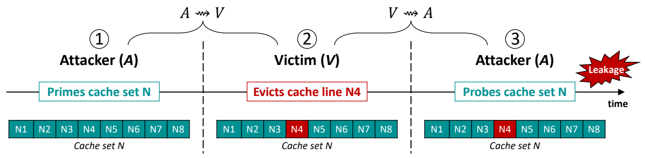

Fig. 2. Cache cyclic interference in Prime+Probe. A and V create two consecutive directional interference events on cache line N_4.

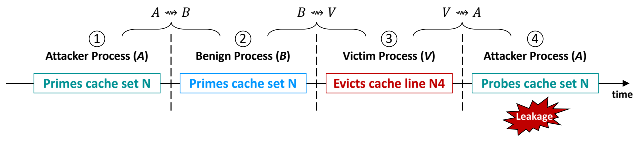

Fig. 3. BENIGNINTERFERE ATTACK using a third party and benign application to launch a Prime+Probe attack, bypassing Cyclone [26] detection.

we show the ineffectiveness of cache cyclic interference detectors and demonstrate they are not sufficient to be used to mitigate Spectre attacks. We provide more details in Section II-C. Finally, prior work has demonstrated, with very high accuracy, the ability to detect the phases of a Spectre attack through direct analysis of the microarchitectural changes of a successful attack [28]. We demonstrate that combining this detector, or even an ideal detection methodology, with existing mitigation strategies can potentially introduce new vulnerabilities, ultimately enabling the attacker to recover sensitive information.

C. Cache Contention Cyclic Interference Detection

We provide a detailed background of Cyclone [26], as the state-of-the-art cyclic interference detector, since we intend to have an in-depth investigation of the limitations of such detectors. We are the first to reveal the risks of using this class of detection.

A directional interference on a resource (*e.g.*, cache line N) is defined as $X \rightsquigarrow Y$, in which X and Y are two processes with different security domains (*i.e.*, they need to be isolated from each other) which both touch the same resource. Cyclic interference detectors exploit the common property of known cache contention leaks: recurring and consecutive interference on resources and memory addresses (*e.g.*, $X \rightsquigarrow Y$ and $Y \rightsquigarrow X$ occurring consecutively on cache line N). Figure 2 shows an example of cyclic interference performed by the Prime+Probe attack (A is the attack process, and V is the victim). As it can be seen, there are two consecutive directional interference events on cache line N_4: (1) $A \rightsquigarrow V$, and (2) $V \rightsquigarrow A$, creating a $A \rightsquigarrow V \rightsquigarrow A$ cyclic interference. The Cyclone methodology uses a local detector to determine the cyclic interference events of resources and memory addresses completed by processes with different security domains, and not interfered with third party applications. Cyclone includes *all* cache contention-based attacks in their threat model, and in addition, it claims to detect Spectre attacks through cache contention. We will present a new version of Spectre attacks that can bypass the Cyclone local detector (Section III-A). Note, that Cyclone can only detect cache contention-based attacks and does not cover attacks like Flush+Flush.

III. MOTIVATION

The goal of this work is to propose a highly efficient and robust protection for cache-based Spectre attacks. We aim to enable our proposed mitigation strategy only when a potential attacker is present in the system. To achieve this, we require a detector that is *accurate* (zero false negatives and low false positives), *robust* (comprehensive security against existing attacks), and *timely* (reporting the leaks before the attacker has a chance to recover it) in order to selectively and efficiently protect the potential data leaks [18], [28]. In this section, we will explore different viable and existing options to use detection mechanisms to provide mitigation, and evaluate their effectiveness. ML-based detectors have already been shown to be vulnerable to evasive attacks [28]. Hence, we start with another class of detection, cyclic interference detectors. We first demonstrate the limitation of the Cyclone [26], as the state-of-the-art cyclic interference detector, through the first of a number of new attacks presented in this work, BENIGNINTERFERE ATTACK. Second, we discuss that even an ideal and patched version of cyclic interference detectors is not timely enough to be used for mitigation (SINGLEPROBE ATTACK). Third, we discuss the limitations of isolation and obfuscation techniques, as the only current viable mitigations to benefit from detectors. We show that such defenses are still vulnerable (SINGLEPRIME ATTACK) even using an ideal detection mechanism.

We will conclude by showing that blindly connecting detection and mitigation strategies can introduce new vulnerabilities. Hence, an efficient and robust solution requires to carefully co-design the detection and mitigation mechanisms in order to protect all potential speculatively leaked data.

A. BENIGNINTERFERE ATTACK*: Bypassing Cyclone Detection*

As we discussed in Section II-C, Cyclone detection mechanism considers $X \rightsquigarrow Y \rightsquigarrow X$ interference events on a resource as cyclic and potentially malicious, only if it is not interfered with a third party process. We demonstrate that this assumption is not always true through our new attack that uses benign applications as a third party to interfere with the target cache location and still the attacker can successfully recover the victim's secret data. Figure 3 shows how BENIGNINTERFERE ATTACK exploits the fact that the attacker can share data with benign applications (*e.g.,* shared libraries). There are three directional interference events in this attack (A stands for Attacker, B for interfering Benign application, and V for Victim): (1) $A \rightsquigarrow B$, in which both A and B prime the cache line N with shared data. (2) $B \rightsquigarrow V$ interference happens when the victim speculatively evicts the data in cache line N. Finally, (3) $V \rightsquigarrow A$ interference occurs due to the attacker probing cache line N to recover the secret. As you can see, the $A \rightsquigarrow B \rightsquigarrow V \rightsquigarrow A$ sequence does not have any two consecutive directional interference events creating a cyclic contention on cache line N, hence, bypassing the Cyclone local detector. The general sequence of our attack is $A \rightsquigarrow .^* \rightsquigarrow B \rightsquigarrow V \rightsquigarrow A$, which $.^*$ means that any number of applications interfering with the cache line N can execute in between.

While it is possible to improve the Cyclone local detector to catch all cases of BENIGNINTERFERE ATTACK (*i.e.*, counting all $A \rightsquigarrow .^+ \rightsquigarrow A$ interference sequences as potentially malicious), the performance penalty and false positive rate of such modification is significant. We tested a set of SPEC CPU2006 applications with the original and the patched version of the Cyclone[1], and we observe 164% increase in the false positive rate of cyclic events detected by the local detector, ultimately making it less practical to be used for efficient mitigation because of more false alerts.

B. SINGLEPROBE ATTACK*: Bypassing Isolation/Obfuscation + Ideal Cyclic Interference Detection*

Among all mitigation strategies discussed in Section II-B, restriction based, isolation, and access obfuscation solutions provide strong protections (as the other techniques are proven to be vulnerable or

[1]Section VII-A provides the details of our simulation and applications setup.

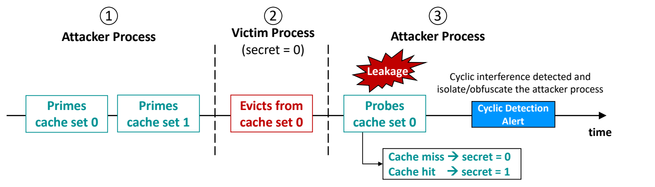

Fig. 4. SINGLEPROBE ATTACK bypassing an ideal cyclic interference detection when combined with isolation/obfuscation (Prime+Probe example).

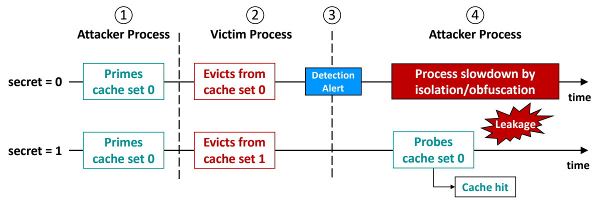

Fig. 5. SINGLEPRIME ATTACK bypassing an isolation based mitigation using an ideal detector (Prime+Probe example).

provide limited protections). However, restriction-based mitigations cannot benefit from detection mechanisms since they need to be enabled by default in order to prevent data leaks; enabling such mitigations after the detector's alert is not timely enough to mitigate the data that is already leaked.

Isolation-based and obfuscation-based strategies are viable candidates to be combined with an SCA detector and prevent potentially malicious processes to extract the secret when detectors raise an alert (*e.g.*, [29] deploys Cyclone to isolate malicious processes for LLC attacks). We present SINGLEPROBE ATTACK that shows the ineffectiveness of combining isolation/obfuscation with an ideal version of cyclic interference detectors (*i.e.,* resistant against BENIGNINTERFERE ATTACK with a low false positive rate). The key insight of our attack is to show that *cyclic interference detectors are not timely enough to protect data leaks and prevent secret recovery*.

Figure 4 shows the steps of SINGLEPROBE ATTACK. First, the attacker primes two cache lines to infer one bit of the secret. The victim evicts the data in the second step (the secret bit is 0 in this example, so the victim evicts from cache set 0). In the third step, the attacker probes the first cache set (cache set 0) and then the detector detects a cyclic interference on cache line 0 and raises the alert. While the attacker's process is isolated or obfuscated afterward, but only probing the first cache line before the alert, enables the attacker to recover the secret bit (observing a cache miss means that the secret is 0, otherwise the secret is 1). Hence, we conclude that even improving the cyclic interference detectors is not secure and not timely enough to raise an alert before the data leak.

C. SINGLEPRIME ATTACK*: Bypassing Isolation/Obfuscation + Ideal Detection*

In this section, we go one step further and present SINGLEPRIME ATTACK to show the limitation of isolation/obfuscation mitigation strategies when combined even with an ideal detector (*i.e.,* accurate, robust, and timely). The key insight of our attack is that the protections are enabled or disabled in a secret dependent way. Figure 5 depicts the four steps of SINGLEPRIME ATTACK for a Prime+Probe approach. In the first step, the attacker primes only one cache set (cache set 0). Then the victim evicts from cache set 0 or cache set 1 depending on the secret bit (step 2). In step 3, if the primed cache set and the evicted cache set match then the detector raises an alert and activates isolation or obfuscation for the next process, resulting in a slowdown of the process. However, if the evicted cache set is not among primed cache sets then the detectors will not raise an alert, hence, the attacker can successfully probe the primed cache line and experience no slowdown. Since both isolation and access obfuscation strategies slow down the execution the attacker can infer the secret bit by monitoring the time difference between an enabled and disabled isolated/obfuscated mode.

We conclude that blindly enabling existing mitigation strategies selectively by using detectors (*e.g.,* the state-of-the-art detector [28] or even an ideal detector) will not provide a robust solution. Hence, in this work, we discuss that we need to co-design the detection and mitigation. We propose HIDFIX that aims to accurately detect the speculatively leaked data and rollback all these potential leakages before the attacker gets a chance to extract them. In Section V, we will discuss the details of our methodology and the required changes on the microarchitectural level.

IV. THREAT MODEL

The threat model of HIDFIX includes all Spectre-style attacks [2], [34]–[36] that use the cache as the channel to leak the victim's secret data. In this work, we demonstrate the effectiveness of the HIDFIX methodology for the Prime+Probe [13], Flush+Reload [14], and Flush+Flush [12] cache primitives. However, our approach can be extended to other cache primitives used for Spectre attacks.

We consider a trusted OS and a multi-core system in which the main assumption for the attacker is that it runs on the same core as the victim and they share the branch predictor and the private data cache (a common assumption of Spectre-style attacks). Note, that we do not consider cross-core Spectre attacks, as no prior work has demonstrated the practicality of such an adversary; The main challenges of cross-core Spectre attacks are that branch predictors are not shared among different cores, and also non-inclusive last-level caches in modern processors [37] make it challenging to create cross-core cache-based side-channel. However, HIDFIX methodology can be extended to all layers of the cache subsystem and is not fundamentally limited to private data caches.

V. HIDFIX DESIGN

The goal of our work is to provide efficient mitigation for speculative data leaks that occur via the private data caches of modern processors. We aim to accomplish this goal by achieving three key requirements:

RQ1 Accurately spotting the speculatively leaked data through the data cache;
RQ2 Reverting the data leaks before a potential attacker has a chance to extract the data;
RQ3 Minimizing performance and efficiency overheads, while comprehensively blocking all the leaks.

We present the HIDFIX methodology in Section V-A that aims to achieve these requirements. Section V-B provides the details of our microarchitecture to implement HIDFIX in the hardware.

A. Overview of Methodology

***Spotting Speculative Data Leaks*.** To comprehensively block any speculative data leaks, we first have to spot all the potential leaks before the attacker gets a chance to extract them (**RQ1**). As an overview, we first describe the common steps of all cache-based Spectre attacks:

Step 1 (*initialization*): Initializing the cache into a known state (*e.g.,* primed with known data in Prime+Probe, and flushing a specific set of memory addresses in Flush+Reload and Flush+Flush).

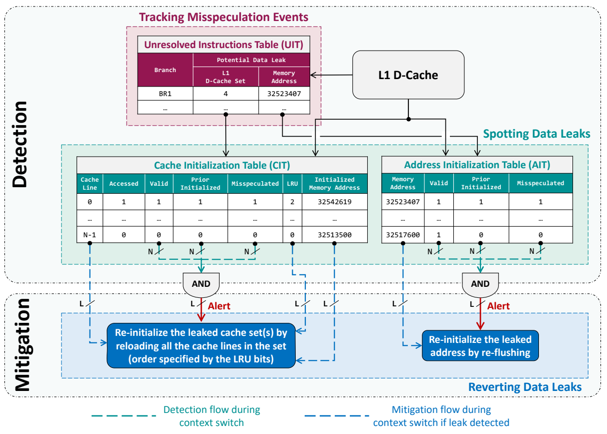
Tracking Misspeculation Events
Unresolved Instructions Table (UIT)
Branch | Potential Data Leak | L1 D-Cache Set | Memory Address
BR1 | 4 | 32523407
… | … | …
L1 D-Cache
Spotting Data Leaks
Detection
Cache Initialization Table (CIT)
Cache Line | Accessed | Valid | Prior Initialized | Misspeculated | LRU | Initialized Memory Address
0 | 1 | 1 | 1 | 1 | 2 | 32542619
… | … | … | … | … | … | …
N-1 | 0 | 0 | 0 | 0 | 0 | 32513500
Address Initialization Table (AIT)
Memory Address | Valid | Prior Initialized | Misspeculated
32523407 | 1 | 1 | 1
… | … | … | …
32517600 | 1 | 0 | 0
N
AND
L
Alert
Mitigation
Re-initialize the leaked cache set(s) by reloading all the cache lines in the set (order specified by the LRU bits)
Re-initialize the leaked address by re-flushing
Reverting Data Leaks
Detection flow during context switch
Mitigation flow during context switch if leak detected

Fig. 6. Overview of HIDFIX design. The detection part spots data leaks (N checks during the context switch) and the mitigation part rollbacks the leaks during context switch, before scheduling the next process. N is the size of CIT and AIT tables, and L is the number of leaks detected at context switches.

Step 2 (*victim leak*): The victim misspeculatively transmits potentially sensitive data to the initialized cache sets or memory addresses.

Step 3 (*data extraction*): The attacker probes the initialized part of the cache (*e.g.,* via accessing its known data or known addresses) and looks for the changes made by the victim.

Based on these steps, we need to determine all of the potential misspeculated leaks before the secret extraction step (step 3). We deploy a similar approach to [28] and directly track and spot potential data leaks. A cache location or a memory address is considered a potential data leak if it has been initialized by prior processes, either primed or flushed (*leak condition 1*); and it has been speculatively accessed by the current process and resulted in misspeculation (*leak condition 2*). The majority of prior work only relies on the second condition (*i.e.*, the misprediction events) to spot the potential data leaks [6], [7], however, we argue that the attacker will not be able to recover a misspeculated data leak if the leakage source (either cache set or memory address) has not been initialized by prior processes (*i.e.*, *leak condition 1*).

Our leak conditions capture all known cache-based Spectre attacks (including our new attacks; Section III), and still provide a more accurate approach (zero false negatives and low false positives), resulting in an efficient approach to block the leaks (**RQ3**). Prior work [28] attempts to be more selective by ignoring the cases that the number of initialized locations and addresses are less than a threshold (*e.g.*, at least two caches sets need to be primed if the attacker is leaking one bit of the secret). However, our SINGLEPRIME ATTACK demonstrates the possibility of data leak in some situations even by priming only one cache set. We do not consider any thresholds for the number of initialized locations and addresses.

Our microarchitecture uses two tables: (1) Cache Initialization Table (CIT) that tracks the initialization state of all cache lines (monitoring attacks like Prime+Probe), and (2) Address Initialization Table (AIT) that tracks all the initialized memory addresses (monitoring attacks like Flush+Reload and Flush+Flush). In addition, these tables store more information to track all the leak conditions we discussed (*e.g.*, there is a misspeculated bit that indicates the leak condition 2). Figure 6 shows the new structures of HIDFIX microarchitecture that we discuss the details of the tables and their operation in Section V-B.

Reverting the Speculative Data Leaks. To effectively revert all the potential data leaks before the attacker has the ability to extract them (**RQ2**), we deploy a rollback mechanism at the context switches, before running the next process, where we have spotted potential data leaks. The goal of a rollback mechanism is to re-initialize the misspeculatively leaked cache locations and memory addresses that had been initialized by previous processes (possibly a malicious process). In other words, a rollback mechanism prevents the attacker from inferring any secret dependent changes in the initialized part of the cache, hence, preventing the data extraction step of the attack.

One of the key aspects of our approach is that we only perform the rollback at the end of the context switches by a trusted OS, before scheduling the next process. Authors of [16] have demonstrated the vulnerabilities of mitigations that solve the data leaks by interfering with the running application [6] since it enables the attacker to observe secret dependent timing differences introduced by the mitigation operations. However, our approach does not have this vulnerability because it does not interfere with the running process, and prevents attackers from finding the exact timing of data leaks or creating customized gadgets that reveal secret dependent timings.

To rollback the leaks through cache initialization (*i.e.*, primed in case of Prime+Probe), we reload the affected and leaked cache sets; The cache lines are reloaded in the order specified by the initialized Least Recently Used (LRU) bits checkpointed in CIT, to preserve the same order that a potential attacker has used to initialize the cache set; Otherwise, the order of evictions will be different in the attack probe phase which reveals if the cache set has been touched by the victim. To rollback the leaks through address initialization (*i.e.*, flushed in case of Flush+Reload and Flush+Flush), we flush the leaked addresses again to ensure the attacker will not observe any unintended leak from the victim. In Section VI, we discuss the security analysis of our rollback mechanism and demonstrate that we hide all the potential data leaks, and moreover, we do not introduce new changes that might enable the attacker to bypass our defense.

Walkthrough Example. Figure 7 shows the HIDFIX methodology when running two variants of Spectre: (a) via Prime+Probe, and (b) via Flush+Reload. Process P_{n-1} (a potentially malicious process) primes two cache sets by initializing with known data or initializes three memory addresses by flushing known addresses (step ❶). Tables CIT and AIT are updated with this information during the execution of process P_{n-1}. Then in step ❷, process P_n (potentially a victim) leaks one of the initialized cache sets (cache set 1) during a speculation window resulting in misspeculation. In addition, one of the initialized memory addresses is leaked misspeculatively as well (memory address 2). CIT and AIT tables mark the initialized cache sets and memory addresses that have leaked misspeculatively. In step ❸, we check the initialization tables to detect valid speculative data leaks during context switch CTX_{n+1}. Since a misspeculated data leak is spotted in CIT and AIT, HIDFIX triggers the rollback mechanism to re-initialize the leaked cache set (cache set 1) and memory address (address 2). When the next process, process P_{n+1}, starts executing the state of initialized caches sets and memory addresses will be the same as process P_{n-1} (step ❹), hence, there will be no evidence for a potential attacker to infer the victim's data.

B. *Microarchitecture*

Figure 6 depicts the new structures of HIDFIX and their interactions on top of a baseline out-of-order (OoO) processor. HIDFIX microarchitecture has three pieces to ultimately block the misspeculated data leaks: (1) tracking the initialization state of cache lines and memory addresses, (2) tracking misspeculation events, and (3) reverting the data leaks of misspeculated instructions to their previosuly initialized states.

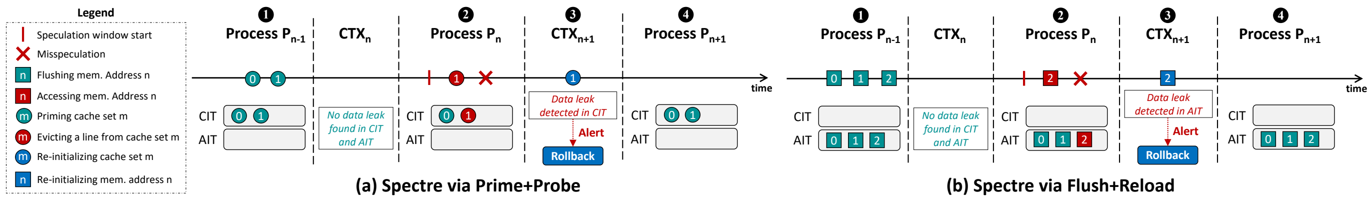
Legend
Speculation window start
Misspeculation
Flushing mem. Address n
Accessing mem. Address n
Priming cache set m
Evicting a line from cache set m
Re-initializing cache set m
Re-initializing mem. address n
Process P_{n-1}
CTX_n
Process P_n
CTX_{n+1}
Process P_{n+1}
time
CIT
AIT
No data leak found in CIT and AIT
Data leak detected in CIT
Data leak detected in AIT
Alert
Rollback
(a) Spectre via Prime+Probe
(b) Spectre via Flush+Reload

Fig. 7. Workflow of HIDFIX for detection and mitigation of Spectre via (a) Prime+Probe, and (b) Flush+Reload cache primitives (Flush+Flush have a similar initialization phase). P_{n-1} and P_{n+1} can be the same process, as the attack process and the common for Spectre. CTX_i is i^{th} context switch in the system.

(1) Tracking initialization states. We use two tables, the CIT and AIT, to store the initialization information of all cache lines and memory addresses (entries in the CIT are indexed by cache line numbers, and entries in the AIT are indexed by memory addresses). Each access to the L1 D-Cache updates the corresponding cache line entry in the CIT ($Accessed = 1$). In addition, the memory address of the data that has primed the cache line and its LRU bits are checkpointed at the CIT table; We need this information later to re-initialize the cache sets (with the same order) that have misspeculatively leaked data. Any flushing operation on L1 D-Cache will update the AIT with the flushed memory address (memory addresses have the cache line granularity; as used in x86 `clflush` instructions). We include a $Valid$ bit per entry to indicate the initializations that have been done by a user process, and not by OS during the context switches.

At the end of each context switch, the $Prior\_Initialized$ bit is set to 1 only if the entry has been primed or flushed by prior processes and not by the OS during the context switch (*i.e.*, $Valid = 1$). In addition, the $Accessed$ bits of all entries in the CIT are reset to 0 to capture the access patterns of the next process.

(2) Tracking misspeculation events. To capture data leaks during speculative execution of the core, we monitor all live unresolved branches in the ROB using a new structure called the Unresolved Instructions Table (UIT). Whenever a cache miss occurs during the speculation window of an unresolved branch, the corresponding entry in the UIT is updated with the affected cache set number and the memory address. If any of the branches in the UIT result in misprediction, the $Misspeculated$ bit of the affected entries in the CIT and AIT are set to 1. In other words, upon each misspeculation event, we update the initialization tables to mark the entries that have been misspeculatively touched (*i.e.*, the unintended cache changes).

(3) Reverting misspeculated data leaks. At each context switch, we check the status of the initialization tables to detect misspeculated data leaks. First, we look up the CIT and AIT to find such leaks. Cache data leaks, as defined in Equation 1, are all entries e in the CIT that have been initialized by prior processes ($prior\_initialized = 1$) and have not been interfered by context switches ($Valid = 1$). Moreover, it has been misspeculated ($misspeculated = 1$).

$$cache\_leaks = \{e \ \in \ CIT \mid e.Misspeculated = 1 \ \wedge \\ e.Prior\_Initialized = 1 \ \wedge \ e.Valid = 1\} \tag{1}$$

We detect the leaks through memory addresses in a similar way, as defined in Equation 2.

$$memory\_leaks = \{e \ \in \ AIT \mid e.Misspeculated = 1 \ \wedge \\ e.Prior\_Initialized = 1 \ \wedge \ e.Valid = 1\} \tag{2}$$

When the sets of data leaks have been determined, a rollback mechanism is initiated to re-initialize the affected cache sets and memory addresses. For the memory address leaks, we just flush those addresses and ensure that the next processes will not observe a cache hit for the leaked addresses. For the cache leaks, we reload all the cache lines of the leaked cache set. We issue n loads, as n is the number of cache lines per cache set (*i.e.*, the number of ways), with the checkpointed addresses in the CIT; the order of loads is specified by the checkpointed LRU bits in the CIT. It is important to issue the loads in the same order that a potential attacker has used for initialization to guarantee that the probing access times will be the same as if the victim has not leaked anything. The rollback mechanism is an atomic operation, for example by acquiring a lock to the target resource and freeing the lock when it is updated by the valid data. This ensures that the system is updated with the correct re-initializations before handing over the system to a potential attacker.

VI. SECURITY ANALYSIS

In this section, we discuss the comprehensiveness of our mitigation, both in spotting and reverting data leaks. Moreover, we aim to show that HIDFIX does not introduce additional changes enabling other side-channel attacks. We study an adversary that runs on the same core as the victim and there is a context switch when transitioning between the malicious and benign processes. Figure 8 depicts the timeline of the events occurring during an attack initializing the cache sets (*e.g.*, Prime+Probe) without the HIDFIX protection in a baseline OoO processor (No Protection), and the HIDFIX protection enabled. As it is shown, the target cache set is initialized (*i.e.*, primed) at the beginning ($Initial\ State$), then the victim misspeculatively evicts a cache line from the initialized cache set. HIDFIX detects such behavior in the next context switch (CTX_{n+1}), before the next process starts running, and initiates the rollback mechanism, while the unprotected baseline leaves the potential data leak as it is. Finally, the attacker will observe indistinguishable access timings for the target cache sets when HIDFIX is enabled ($Next\ Access$). However, an unprotected processor will show a timing difference that enables the attacker to recover the secret (*i.e.*, cache miss during $Next\ Access$ informs the attacker that the initialized cache set has been evicted by the victim). Note, that HIDFIX re-initializes cache lines in the leaking cache set in the same order that the attacker has initialized originally (the order is specified with the checkpointed LRU bits of the cache lines). This guarantees that the state of the cache set is exactly the same as the original state that the attacker has initialized.

In addition, we demonstrate that the updates of the cache coherence state of the affected cache lines during the rollback mechanism do not introduce new vulnerabilities. To provide background, each cache block in MESI coherence protocol has four possible states[2]: (1) *Modified* (M) state means that the cache block is present in only one private cache and is dirty. (2) *Shared* (S) state means that the cache block is present in multiple privates caches and is clean, and (3)

[2]As explained in prior work [38], different processor families might have more coherence states (*e.g.*, MESIF protocol in Intel Xeon processors), but they only serve as performance optimizations and do not add any fundamentally different coherence states.

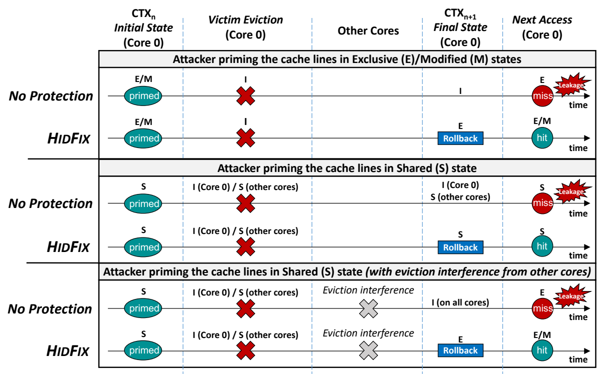
CTX$_n$ Initial State (Core 0)
Victim Eviction (Core 0)
Other Cores
CTX$_{n+1}$ Final State (Core 0)
Next Access (Core 0)
Attacker priming the cache lines in Exclusive (E)/Modified (M) states
No Protection
HIDFIX
E/M
primed
I
E
Rollback
miss
hit
Leakage
time
Attacker priming the cache lines in Shared (S) state
S
I (Core 0) / S (other cores)
I (Core 0)
S (other cores)
Attacker priming the cache lines in Shared (S) state (with eviction interference from other cores)
Eviction interference
I (on all cores)

Fig. 8. Timeline of the events with and without HIDFIX protection for cache initialization attacks (*e.g.*, Prime+Probe). The coherence state of the primed and leaked cache set is also indicated, and HIDFIX does not introduce any new leaks due to $E/M \rightarrow S$ transitions [6], [38].

Exclusive (E) state means that the cache block is present in only one private cache and is clean. (4) *Invalid* (I) state indicates that none of the private caches have a valid copy of the cache block. Prior work [6], [38] demonstrates that $E/M \rightarrow S$ state changes can be a vulnerability as they present an observable timing difference. Figure 8 indicates the coherence state of the target cache block on the top and with different initial states (as the attacker has primed the block in E/M state or S state). As one can see, HIDFIX operations do not introduce these vulnerabilities (*i.e.*, $E/M \rightarrow S$ state transitions) in any of the transitions happening during HIDFIX rollbacks, even if there are eviction interference during the victim execution from other cores (*i.e.*, the shared data has been evicted in remote private caches). Note, that HIDFIX does not aim to mitigate non-speculative coherence vulnerabilities [38] which are out of the scope of this work.

A. *Security of* HIDFIX *against Known Attacks*

We argue that the HIDFIX approach to detect and mitigate the data leaks is sound since it directly tracks the initialization and misspeculation state of all cache lines and memory addresses. Here, we discuss the protection of HIDFIX against known attacks.

Security of HIDFIX against our attacks (Section III). BENIGNINTERFERE ATTACK bypasses the Cyclone detection by a third party application breaking the cyclic interference pattern of the attack. However, HIDFIX is not vulnerable to such an attack since we consider a cache set or memory address to be initialized, even if it has been initialized multiple times by different processes. SINGLEPROBE ATTACK defeats the cyclic interference detectors since they are not timely in spotting the data leaks, and the attacker can leak one bit of the secret before an alert is raised. However, HIDFIX spots the data leaks before the system is handed over to a potentially malicious process. SINGLEPRIME ATTACK defeats the combination of isolation/obfuscation mitigations and detection methodologies. Since this combination results in slowing down the attacker's process based on the secret value and the attacker can infer the secret key by observing this behavior. HIDFIX is not vulnerable to such attacks because it does not cause a slowdown of the attacker's processes and it reverts all of the victim changes during the OS context switches as a trusted software. Moreover, the rollback delays are extremely small compared to the context switch delays. Hence, when the attacker runs there will be no footprints of the victim in the system (*i.e.*, misspeculated data leaks) and the attacker's process runs normally without any signs of secret dependent behavior.

TABLE I
PROCESSOR CONFIGURATION FOR GEM5 SIMULATION.

L1 D/I Cache	32KB, 8-way	LQ/SQ/IQ/ROB	32/32/96/192 entries
L2 Cache	256KB, 8-way	RF (INT/FP)	256/256 entries
L3 Cache	1MB, 16-way	Branch Predictor	TAGE-SC-L-64KB
F/D/I/C width	8/8/8/8	CIT/AIT/UIT	512/512/16 entries

Security of HIDFIX against attacks defeating current cache-based Spectre mitigations. Speculative Interference attacks [15] defeat invisible speculation defenses and exploit the fact these mitigations load the secret values if they hit in the cache. Hence, they create resource contention with the secret values to extract the secret. The UnXpec attack [16] defeats the undo-based defenses, like CleanupSpec [6], by creating gadgets that create secret dependent timing differences introduced by undo operations. HIDFIX is secure against both attacks as it does not interfere with the execution of any of the running processes and all rollback operations are handled at the end of context switches before scheduling the next process. Hence, an attacker cannot trigger and observe secret dependent timing differences during its execution.

Security of HIDFIX against attacks evading ML-based detectors. Prior work [28], [39] has demonstrated the limitations of ML-based detection methodologies either by changing the footprint of the attack or by connecting benign gadgets to launch an attack. However, all the known evasive attacks follow the same three steps that we discussed in Section V-A. Since HIDFIX directly tracks all cache lines and memory addresses following the steps of a successful Spectre attack, it can protect against all these attacks.

Table II experimentally confirms the ability of HIDFIX to detect and mitigate the attacks presented in this paper and the ML evasive attacks [28]. Table III in the Related Work section (Section VIII) provides an overview of prior mitigations and how their protection guarantees compare to HIDFIX.

VII. EVALUATION

A. *Experimental Setup*

We use gem5 simulator [40] to evaluate the performance impacts of HIDFIX. The simulations are done in Syscall Emulation (SE) mode using the DerivO3CPU core model. In addition, we use CACTI 6.5 [41] to estimate the area and power overheads. Table I shows the details of our gem5 configuration. Note, that the size of the CIT is equal to the number of cache lines in L1 D-Cache. Also, we use a 16-entries UIT table since our experiments show that the number of unresolved branches does not hit the limit of 16.

We test our design with the SPEC CPU2006 [30] benchmark suite and build 100M-instruction representative executables (SimPoint) for each application using the ELFies methodology [42]. Additionally, we communicated with the authors of Cyclone [26] to use their gem5 implementation to evaluate our new attacks against cyclic interference detectors (BENIGNINTERFERE ATTACK and SINGLEPROBE ATTACK). We used the ML evasive attacks from Specify [28] as well to evaluate HIDFIX against such attacks.

B. *Performance Evaluation*

Table II shows the performance impacts of HIDFIX on SPEC CPU2006 applications compared to an unprotected OoO core as the baseline. The execution of each program is split into different frames that represent the program execution between context switches and as the points that we examine the CIT and AIT tables to rollback the potential data leaks[3]. While SPEC CPU2006 applications are

[3]We consider the time quantum of task scheduling to be 10ms [43].

TABLE II
HIDFIX STATISTICS AND PERFORMANCE OVERHEAD.

	Application	#Leaks	Baseline OoO #cycles	HIDFIX #cycles	Performance Overhead (%)
Spectre PoC [2]	Spectre (Prime+Probe)	1	2,476,199	2,476,359	0.0065%
	Spectre (Flush+Reload)	1	1,606,505	1,606,665	0.0099%
	Spectre (Flush+Flush)	1	3,539,091	3,539,251	0.0045%
Specify [28]	Expanded-Spectre-nop	1	1,738,309	1,738,469	0.0092%
	Expanded-Spectre-mem	1	11,811,613	11,811,773	0.0013%
	Benign-Program-Spectre	1	17,736,531	17,736,691	0.0009%
This work	BENIGNINTERFERE	1	1,655,046	1,655,206	0.0096%
	SINGLEPROBE	1	115,091	115,251	0.1390%
	SINGLEPRIME	1	108,159	108,319	0.1479%
SPEC CPU2006	zeusmp	44	65,516,404	65,523,444	0.0107%
	bwaves	4	110,485,539	110,486,179	0.0006%
	bzip2	6	76,260,838	76,261,798	0.0013%
	cactus	0	121,449,812	121,449,812	0.0000%
	gamess	32	53,436,713	53,441,833	0.0096%
	gcc	15	303,419,510	303,421,910	0.0008%
	gobmk	7	97,271,448	97,272,568	0.0012%
	libquantum	0	144,772,205	144,772,205	0.0000%
	mcf	86	435,546,173	435,559,933	0.0032%
	omnetpp	8	171,908,584	171,909,864	0.0007%
	soplex	6	256,567,930	256,568,890	0.0004%
	Average (SPEC CPU2006)‡	18.45	135,986,161	135,989,670	0.0025%

‡ geomean is used for performance overhead, and arithmetic mean for the number of leaks.

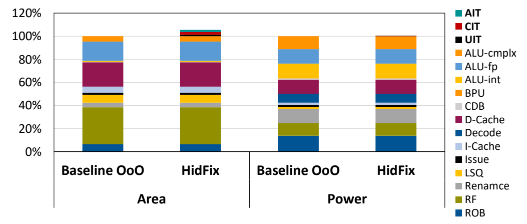

Fig. 9. Power and area overheads of HIDFIX compared to an unprotected baseline out-of-order (OoO) processor. The new structures are shown in bold.

benign, we detected 18.45 data leaks on average (up to 86 leaks in `mcf`) during the execution of 100M instructions as the points that the applications speculatively leaked their data unintentionally. HIDFIX can introduce performance overhead if the rollback mechanism evicts data from the cache and the evicted data is accessed in the next frames of the execution. However, our experiments show that the HIDFIX overheads are very negligible for SPEC CPU2006, only 0.0025% over the unprotected OoO processor.

In addition, we show that HIDFIX performance overhead is negligible under attack as well. We tested three Spectre Proof-of-Concept (PoC) attacks (Prime+Probe, Flush+Reload, and Flush+Flush), ML evasive Spectre attacks [28], and the new attacks that we present in this paper (Section III).

C. Security Evaluation and Penetration Tests

As Table II shows, we are able to detect and mitigate the data leaks occurring during all three tested Spectre PoC attacks. Note, that these attacks are extracting only one byte of the secret and only one data leak happens in the system. In addition, we tested our attacks presented in Section III with both HIDFIX and Cyclone gem5 implementations. While HIDFIX can successfully detect data leaks, the attacks can bypass the Cyclone detection technique. Finally, we have tested HIDFIX design against the evasive Spectre attacks bypassing ML-based detectors [28] and confirm that we can successfully detect the leaks and mitigate them.

D. Power and Area Overheads

Figure 9 shows the detailed power and area overheads of HIDFIX compared to an unprotected baseline OoO core. The average power consumption overhead of HIDFIX is 0.5%, which comes from the new three tables (CIT/AIT/UIT) as direct-mapped memory structures. The area overhead is 5.6% over the baseline OoO core.

TABLE III
RELATED WORK COMPARISON.

Defense	Approach	Protection for Cache-based Spectre		Max. reported perf. overhead	Flexibility‡
		Safe?	Vulnerability		
DOLMA [3]	Restriction	✗	Speculative Interference [15]	130%	○
NDA [4]	Restriction	✓	-	240%	○
MI6 [17]	Isolation	✓	-	34%	◐
InvisiSpec [7]	Invisible Speculation	✗	Speculative Interference [15]	80%	○
CleanupSpec [6]	Undoing Speculation	✗	UnXpec [16]	25%	○
CEASER [5]	Cache Randomization	✗	Address Contention (*e.g.*, Flush+Reload)	4%	○
HIDFIX	Selective Rollback	✓	-	∼ 0%	●

‡ ○ : cannot combine with a detector, ◐ : can combine with a detector, but vulnerable to SINGLEPRIME ATTACK, ● : can combine with a detector with no known vulnerabilities. SINGLEPRIME is our proposed attack.

VIII. RELATED WORK

Table III shows existing approaches to mitigate cache-based Spectre attacks. Restriction-based defenses attempt to comprehensively protect all potential channels, including the data caches. However, DOLMA [3], as a recent restriction-based defense, is vulnerable to attacks like Speculative Interference attacks [15]. NDA [4] is the most comprehensive solution which incurs prohibitive performance overheads of up to 240% for some applications. Invisible speculation and undo-based speculation defenses show better average performance overheads, however, all of them are vulnerable to recent attacks [15], [16]. Cache randomization techniques, like CEASER [5], incur low performance overhead, but they cannot provide protection for address contention attacks (*e.g.*, Flush+Reload and Flush+Flush). In addition, all the aforementioned defenses are not flexible to benefit from a detection methodology since they need to be enabled by default and it will be late for them to selectively enable their mitigations. In other words, they cannot combine with detection methodologies to decrease performance overheads. Isolation-based techniques, like MI6 [17], have the flexibility to benefit from detection methodologies. However, we have demonstrated that combining isolation methods with detectors can introduce new vulnerabilities (see SINGLEPRIME ATTACK in Section III-C). HIDFIX is the first solution providing the flexibility of taking advantage of detection methodologies, resisting all known cache-based Spectre attacks and their evasive variants [28], [44], and finally showing near zero performance overhead.

IX. CONCLUSION

In this work, we investigate existing detection and mitigation methodologies proposed for cache-based Spectre attacks, and how these methods can be combined to achieve higher efficiency against such attacks. We present three new attacks and vulnerabilities demonstrating the limitations of prior work in the direction of detection and mitigation of Spectre attacks. We propose HIDFIX as an efficient and robust detection/mitigation methodology. We spot all misspeculated data leaks that a potential attacker has prior knowledge about them (*i.e.*, initialized by the attacker). HIDFIX guarantees to rollback the cache state to a state that is indistinguishable for the attacker. We provide an in-depth discussion of HIDFIX protection against known attacks, and we argue HIDFIX does not introduce new side effects that might enable an attacker to observe secret dependent changes in the system. We show that HIDFIX incurs almost zero performance overhead, with negligible power and area overheads.

ACKNOWLEDGEMENTS

We would like to thank Austin Harris and the authors of Cyclone for sharing their gem5 implementation. We also thank the anonymous reviewers and Gururaj Saileshwar for feedback and discussion. Supported by Singapore NRF grant NRF2018NCR-NCR002.

References

[1] D. S. McFarlin, C. Tucker, and C. Zilles, "Discerning the dominant out-of-order performance advantage: Is it speculation or dynamism?" in *ASPLOS*, 2013.

[2] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," in *SP*, 2019, pp. 1–19.

[3] K. Loughlin, I. Neal, J. Ma, E. Tsai, O. Weisse, S. Narayanasamy, and B. Kasikci, "Dolma: Securing speculation with the principle of transient non-observability," in *USENIX Security*, 2021.

[4] O. Weisse, I. Neal, K. Loughlin, T. F. Wenisch, and B. Kasikci, "NDA: Preventing speculative execution attacks at their source," in *MICRO*, 2019, pp. 572–586.

[5] M. K. Qureshi, "CEASER: Mitigating conflict-based cache attacks via encrypted-address and remapping," in *MICRO*, 2018, pp. 775–787.

[6] G. Saileshwar and M. K. Qureshi, "CleanupSpec: An "undo" approach to safe speculation," in *MICRO*, 2019, pp. 73–86.

[7] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, "Invisispec: Making speculative execution invisible in the cache hierarchy," in *MICRO*, 2018, pp. 428–441.

[8] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher, "Speculative taint tracking (STT) a comprehensive protection for speculatively accessed data," in *MICRO*, 2019, pp. 954–968.

[9] M. Schwarz, M. Lipp, C. Canella, R. Schilling, F. Kargl, and D. Gruss, "Context: A generic approach for mitigating spectre." in *NDSS*, 2020.

[10] L.-A. Daniel, M. Bognar, J. Noorman, S. Bardin, T. Rezk, and F. Piessens, "Prospect: Provably secure speculation for the constant-time policy (extended version)," *arXiv preprint arXiv:2302.12108*, 2023.

[11] R. Choudhary, J. Yu, C. Fletcher, and A. Morrison, "Speculative privacy tracking (SPT): Leaking information from speculative execution without compromising privacy," in *MICRO*, 2021, pp. 607–622.

[12] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+Flush: a fast and stealthy cache attack," in *DIMVA*, 2016, pp. 279–299.

[13] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *SP*, 2015, pp. 605–622.

[14] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack," in *USENIX Security*, 2014, pp. 719–732.

[15] M. Behnia, P. Sahu, R. Paccagnella, J. Yu, Z. N. Zhao, X. Zou, T. Unterluggauer, J. Torrellas, C. Rozas, A. Morrison *et al.*, "Speculative interference attacks: Breaking invisible speculation schemes," in *ASPLOS*, 2021, pp. 1046–1060.

[16] M. Li, C. Miao, Y. Yang, and K. Bu, "unXpec: Breaking undo-based safe speculation," in *HPCA*, 2022, pp. 98–112.

[17] T. Bourgeat, I. Lebedev, A. Wright, S. Zhang, and S. Devadas, "MI6: Secure enclaves in a speculative out-of-order processor," in *MICRO*, 2019, pp. 42–56.

[18] H. Wang, H. Sayadi, A. Sasan, S. Rafatirad, and H. Homayoun, "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 ICCAD*, 2020, pp. 1–9.

[19] S. Mirbagher-Ajorpaz, G. Pokam, E. Mohammadian-Koruyeh, E. Garza, N. Abu-Ghazaleh, and D. A. Jiménez, "Perspectron: Detecting invariant footprints of microarchitectural attacks with perceptron," in *MICRO*, 2020, pp. 1124–1137.

[20] M. Mushtaq, A. Akram, M. K. Bhatti, V. Lapotre, and G. Gogniat, "Cache-based side-channel intrusion detection using hardware performance counters," in *CryptArchi 2018-16th International Workshops on Cryptographic architectures embedded in logic devices*, 2018.

[21] M. Payer, "Hexpads: a platform to detect "stealth" attacks," in *International Symposium on Engineering Secure Software and Systems*. Springer, 2016, pp. 138–154.

[22] X. Wang, C. Konstantinou, M. Maniatakos, and R. Karri, "Confirm: Detecting firmware modifications in embedded systems using hardware performance counters," in *ICCAD*, 2015, pp. 544–551.

[23] T. Zhang, Y. Zhang, and R. B. Lee, "Cloudradar: A real-time side-channel attack detection system in clouds," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2016, pp. 118–140.

[24] S. M. Ajorpaz, D. Moghimi, J. N. Collins, G. Pokam, N. Abu-Ghazaleh, and D. Tullsen, "EVAX: Towards a practical, pro-active & adaptive architecture for high performance & security," in *MICRO*, 2022, pp. 1218–1236.

[25] J. Chen and G. Venkataramani, "CC-Hunter: Uncovering covert timing channels on shared processor hardware," in *MICRO*, 2014, pp. 216–228.

[26] A. Harris, S. Wei, P. Sahu, P. Kumar, T. Austin, and M. Tiwari, "Cyclone: Detecting contention-based cache information leaks through cyclic interference," in *MICRO*, 2019, pp. 57–72.

[27] F. Yao, H. Fang, M. Doroslovacki, and G. Venkataramani, "Towards a better indicator for cache timing channels," *arXiv preprint arXiv:1902.04711*, 2019.

[28] A. Pashrashid, A. Hajiabadi, and T. E. Carlson, "Fast, robust and accurate detection of cache-based spectre attack phases," in *ICCAD*, 2022, pp. 1–9.

[29] Y. Verma and B. Panda, "Avenger: Punishing the cross-core last-level cache attacker and not the victim by isolating the attacker," in *SEED*, 2022, pp. 1–12.

[30] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[31] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Safespec: Banishing the spectre of a meltdown with leakage-free speculation," in *DAC*, 2019, pp. 1–6.

[32] C. Sakalis, S. Kaxiras, A. Ros, A. Jimborean, and M. Själander, "Efficient invisible speculative execution through selective delay and value prediction," in *MICRO*, 2019, pp. 723–735.

[33] G. Saileshwar and M. K. Qureshi, "MIRAGE: Mitigating conflict-based cache attacks with a practical fully-associative design." in *USENIX Security*, 2021, pp. 1379–1396.

[34] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution," in *EuroS&P'19*. IEEE, 2019, pp. 142–157.

[35] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. B. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer." in *WOOT@ USENIX Security Symposium*, 2018.

[36] G. Maisuradze and C. Rossow, "ret2spec: Speculative execution using return stack buffers," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2109–2122.

[37] Y. Guo, A. Zigerelli, Y. Zhang, and J. Yang, "Adversarial prefetch: New cross-core cache side channel attacks," in *SP*, 2022, pp. 1458–1473.

[38] F. Yao, M. Doroslovacki, and G. Venkataramani, "Are coherence protocol states vulnerable to information leakage?" in *HPCA*, 2018, pp. 168–179.

[39] C. Li and J.-L. Gaudiot, "Challenges in detecting an "evasive spectre"," *IEEE Computer Architecture Letters*, vol. 19, no. 1, pp. 18–21, 2020.

[40] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

[41] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "Cacti-p: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques," in *ICCAD*, 2011, pp. 694–701.

[42] H. Patil, A. Isaev, W. Heirman, A. Sabu, A. Hajiabadi, and T. E. Carlson, "ELFies: executable region checkpoints for performance analysis and simulation," in *CGO*, 2021, pp. 126–136.

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

[44] M. Mushtaq, J. Bricq, M. K. Bhatti, A. Akram, V. Lapotre, G. Gogniat, and P. Benoit, "Whisper: A tool for run-time detection of side-channel attacks," *IEEE Access*, vol. 8, pp. 83 871–83 900, 2020.