SecurityCloak: Protection against cache timing and speculative memory access attacks

Fernando Mosquera a,∗, Ashen Ekanayake b, William Hua a, Krishna Kavi a, Gayatri Mehta a, Lizy John b

a University of North Texas, United States of America
b University of Texas at Austin, United States of America

A R T I C L E I N F O

Keywords:
Side-channel attacks
Speculative execution
Victim cache
Spectre
Prime & probe
Evict & time
Flush & reload
Guard cache

A B S T R A C T

Microarchitectural innovations such as deep cache hierarchies, out-of-order execution, branch prediction and speculative execution in modern processors have made possible to meet ever-increasing demands for performance. However, these innovations have inadvertently introduced vulnerabilities that are exploited by cache-side channel attacks such as Flush & Reload, Prime & Probe, Evict & Time, and attacks such as Spectre and Meltdown that exploit speculative executions. These attacks can potentially leak information which should be secured.

Mitigating the attacks while preserving the performance of out-of-order execution has been a challenge. Previous hardware mitigation techniques against cache timing or side-channel attacks include complex cache indexing mechanisms, encrypting addresses, partitioning cache memories, assigning specific ways of a set for each process, or obfuscating cache accesses by using ghost threads. Previous techniques for preventing or at least mitigating attacks based on speculative executions include hiding speculative data accesses using separate buffers or caches, or undoing the effects of speculation throughout program execution. Most techniques address either attacks that exploit speculation such as Spectre or cache side-channel attacks but not both. In many cases, changes to the microarchitecture with additional hardware are needed to implement the security protection. In some cases the mitigations cause performance penalties. In contrast we present very simple designs aimed at preventing both timing based cache side-channel attacks and Spectre style attacks based on speculative executions. Our approach combines obfuscation of cache timing making it more difficult for side-channel attacks to succeed and delaying speculative data accesses that miss in cache until the speculation is resolved. We will show that these approaches prevent both timing attacks such as Flush & Reload, Prime & Probe, Evict & Time as well as speculative attacks such as Spectre. Our technique requires very minimal changes to hardware.

1. Introduction

Over the past few decades, computer systems have introduced many major microarchitectural innovations including deep cache hierarchies, out-of-order instruction execution, branch prediction and speculative execution, and issuing multiple instructions per cycle, just to name a few. These innovations were motivated by the need for higher performance at lower energy consumption. However, these innovations have inadvertently introduced vulnerabilities that can potentially leak information that should be secured. The information may be leaked either through a side-channel or direct (modification to application code) attacks.

Among the earliest attacks discovered is a side-channel to information stored in cache memories by observing memory access times, which in turn reveal if an access (to an address) is a hit or a miss in cache. An attacker can use this side-channel to deduce the memory addresses accessed by a victim and extract additional information such as keys used by encryption in AES [1,2]. Some common side-channel attacks include Evict & Time [3], Prime & Probe [3,4] and Flush & Reload [5].

Yet another modern hardware side-channel attack is made possible by the use of out-of-order and speculative execution of instructions [6]. In such systems, processors rely on predictions as to which execution path to proceed along, even before completing the branch instruction that determines the correct path. In most cases, the predictions are very accurate. When the prediction is wrong, the processor squashes (or undoes) the computations along the mispredicted path. This is achieved...
by the use of reorder buffers (ROB). However, when a mispredicted path contains memory accesses (particularly read access), and if the accessed data is not available in cache memories (i.e., read miss), the processing system will bring the data into cache memories. Even when the misprediction is detected and all instructions along the mispredicted path are squashed, the data brought into the cache is not invalidated. At a later time, the attacker can rely on a side-channel to access this data in the cache, which may contain secret information.

Computer architects have been investigating techniques to either prevent or at least mitigate (making it harder for an attacker to successfully launch attacks) these attacks. The solutions can be implemented in software or hardware. For example, ASLR [7] randomizes kernel address space to make it more difficult for the attacker to deduce secret information based on addresses accessed by a victim. This requires no hardware changes but works for Meltdown [8] but does not prevent many Spectre variants [9]. Likewise, preventing “deduplication” [10] eliminates some attacks since shared libraries will have multiple copies but can lead to significant performance penalties.

Hardware techniques to mitigate timing based cache side-channel attacks include partitioning caches so that only one process can access its partition (and no other process can evict data contained in those partitions), changing cache indexing mechanisms or encrypting how an address is mapped to a cache index so that cache misses to an entry cannot easily be translated into addresses, among others [2,4,11–15]. Mitigation of speculative attacks can be categorized into three broad groups: (i) prevent speculation altogether or at least prevent speculative execution of certain instructions, (ii) make speculative execution “invisible” either by buffering all memory accesses performed until speculation is resolved or delaying memory accesses (at least accesses that cause cache misses) until speculation is resolved and (iii) “clean-up” all affects of speculative execution, including evicting any speculatively fetched data from caches.

It should be noted that most published mitigation or prevention techniques address either speculation based attacks or timing based attacks but not both. In this work, we present very simple hardware mitigation techniques that together can address both timing based cache-side channel attacks and illegal or unauthorized memory accesses during speculative execution. We explore various techniques to obfuscate micro-architectural information making it more difficult for attackers.

The main contributions of our work (SecurityCloak) are:

- **Delaying speculative load misses**: Many of the attacks rely on fetching data into the cache during speculation, including the data along mispredicted paths or along unauthorized access paths. Existing techniques allow such accesses but either hide the fetched data using additional caches or buffers [16], or evict such data from cache hierarchies when the misprediction is detected [17]. In our approach, we permit speculative executions and memory accesses that hit in (L1-D) cache but delay memory accesses that miss in the cache until the speculation is resolved. We refer to this technique as SafeLoadOnMiss. A very similar approach was previously proposed in [18]. We make the following modifications to the previous approach:

  - we hold the delayed cache misses in a separate Safe Queue and use instruction sequence numbers2 to order the misses when the speculation is resolved instead of dropping the load misses and allowing the LSQ to retry later as done in [18] - this reduces performance penalties;
  - we use random replacement in primary cache instead of LRU replacement, potentially mitigating some speculative interference attacks [19], as stated in [20].

1 The instruction sequence numbers in Gem5 can be viewed as time stamps.

Our experiments show that this will result in very minimal performance loss and requires minimal hardware: only 5.4% geometric mean performance loss for SPEC2017 benchmarks, and only 160B of additional storage for safe queue.

- **Obfuscating cache timing with false hits**: We use a small fully associative Cache (labeled Guard Cache) which is used both as a Victim cache and a Miss cache [21] to create false hits. Any data item evicted from the primary cache is saved in this Guard Cache. If the evicted item (or victim) is accessed, it can be retrieved from Guard Cache, making the access appear as if it was a hit in the primary cache. To further obfuscate timing, we also rely on random replacement policy when entries in Guard Cache need to be replaced. Additional obfuscations can include randomly not placing “victims” in Guard Cache, thus causing false hits more randomly. While larger Guard Caches can provide more protection since victims can be held for longer periods of time but can lead to higher silicon area and power consumption, we found that even a small Victim Cache (1 Kib to 4 Kib), may be sufficient to prevent several types of attacks, particularly those that rely on Flush & Reload side-channels.

- **Obfuscating cache timing with false misses**: We randomly evict cache lines to cause false misses. On every cache hit, we choose a random cache line for eviction and evict those items based on how often a cache line should be evicted (eviction frequency). While such evictions can cause performance penalties, we have shown that even 10% eviction frequencies can prevent attacks that are based on Prime & Probe side-channels.

- **Safe-mode execution**: The microarchitecture enters a preventive state to mitigate known hardware security attacks such as Spectre and Meltdown as well as a variety of cache side-channel attacks. The safe mode can be turned-on by the user to assure secure execution of critical sections of their programs. Or the system can be endowed with the capability to detect the presence of an attack and turn-on the safe mode. In this way, the system will experience performance impacts only during the safe mode execution. It may also be possible to randomly switch between safe and unsafe modes of execution to make it more difficult for an attack to succeed. This may also make it more difficult for speculation-interference attack [19] to succeed.

The rest of the paper is organized as follows. Section 2 describes the background and motivation for our work. Section 3 describes our SecurityCloak framework design and implementation. Section 4 provides performance results as well as the experimental framework used for the evaluation. Section 5 includes relevant research that is related to our work and Section 6 summarizes our conclusions about SecurityCloak framework.

2. Background and motivation

In pursuit of meeting ever-increasing demands for processor performance, computer systems implemented many major microarchitectural innovations including deep cache hierarchies, out-of-order instruction execution, branch prediction, and speculative executions, issuing multiple instructions per cycle, just to name a few. However, these innovations have inadvertently introduced vulnerabilities that can potentially be exploited to leak information that should be secured. The information may be leaked either through a side-channel or direct (modification to application code) attacks. The two types of attacks that have been widely discussed are cache side-channel attacks (for example, [3–5,22,23]) and speculative attacks (like Spectre [9,24] and Meltdown [8]). Computer architects have been investigating techniques to either prevent or at least mitigate (making it harder for an attacker to successfully launch attacks) these attacks. The solutions can be implemented in software or hardware.

Among the hardware techniques against cache side-channel attacks include (i) partitioning caches so that only one process can access
Fig. 1 shows a slightly modified version of the attack graph. The key to understand speculation-based attacks such as Spectre and its variations (e.g., mcf) cause significantly higher than average performance losses completely. Although not shown in Table 1, we observe that memory access misses cause different amounts of performance loss and the hardware complexity also varies with these solutions. The performance loss depends on application behavior. For example, an application that has higher cache miss rates will likely see higher performance losses when “Delay Load Misses” [18] technique is used. Likewise, “Undo” [17] technique needs to invalidate more cached entries. Obviously, fewer conditional branches result in fewer speculatively executed instructions and smaller amounts of performance losses even if we disable speculation completely.

The performance losses caused by these different techniques are shown in Table 1. As can be expected, different mitigation techniques cause different amounts of performance loss and the hardware complexity also varies with these solutions. The performance loss depends on application behavior. For example, an application that has higher cache miss rates will likely see higher performance losses when “Delay Load Misses” [18] technique is used. Likewise, “Undo” [17] technique needs to invalidate more cached entries. Obviously, fewer conditional branches result in fewer speculatively executed instructions and smaller amounts of performance losses even if we disable speculation completely. Although not shown in Table 1, we observe that memory intensive benchmarks and benchmarks with very high MPKI rates (e.g., mcf) cause significantly higher than average performance losses in all the known mitigation techniques.

In a recent paper [25], the authors describe a graphical model to understand speculation-based attacks such as Spectre and its variations. Fig. 1 shows a slightly modified version of the attack graph. The key observation for our purpose is that there is a delay from the time a conditional branch (or authorization) is executed, initiating speculative execution until when the branch is resolved (or authorization resolved). The speculatively executed instructions during this interval either commit or are squashed based on the resolution. One observation that can be drawn from the model presented in [25] is that the longer the delay in resolving a branch decision (or resolving authorization), the more likely an attack will be successful. In one of the Spectre-like attacks (see Listing 1) that rely on Flush & Reload technique, the array bounds variable (Asize) is flushed from cache. This causes delays in resolving array bounds check (line 5) since the bounds variable needs to be fetched into L1-D cache. The speculative load instructions (lines 6 and 7) along the mispredicted (or unauthorized) path will more likely complete fetching the data (secret) into cache, which will be accessed by attacker at a later time. This delay will also correlate to the performance impacts of the various solutions outlined above. For example, the “Undo” technique may need to invalidate more cached data if the delay is longer; or “Delay Load Misses” [18] or our variation SafeLoadOnMiss may delay several load accesses resulting in higher performance penalties.

Our SafeLoadOnMiss technique is similar to [18]. However, our implementation results in smaller performance losses. Also, we collected data for SPEC 2017 benchmarks and not SPEC 2006.

The performance losses caused by these different techniques are shown in Table 1. As can be expected, different mitigation techniques cause different amounts of performance loss and the hardware complexity also varies with these solutions. The performance loss depends on application behavior. For example, an application that has higher cache miss rates will likely see higher performance losses when “Delay Load Misses” [18] technique is used. Likewise, “Undo” [17] technique needs to invalidate more cached entries. Obviously, fewer conditional branches result in fewer speculatively executed instructions and smaller amounts of performance losses even if we disable speculation completely. Although not shown in Table 1, we observe that memory intensive benchmarks and benchmarks with very high MPKI rates (e.g., mcf) cause significantly higher than average performance losses in all the known mitigation techniques.

In a recent paper [25], the authors describe a graphical model to understand speculation-based attacks such as Spectre and its variations. Fig. 1 shows a slightly modified version of the attack graph. The key observation for our purpose is that there is a delay from the time a conditional branch (or authorization) is executed, initiating speculative execution until when the branch is resolved (or authorization resolved). The speculatively executed instructions during this interval either commit or are squashed based on the resolution. One observation that can be drawn from the model presented in [25] is that the longer the delay in resolving a branch decision (or resolving authorization), the more likely an attack will be successful. In one of the Spectre-like attacks (see Listing 1) that rely on Flush & Reload technique, the array bounds variable (Asize) is flushed from cache. This causes delays in resolving array bounds check (line 5) since the bounds variable needs to be fetched into L1-D cache. The speculative load instructions (lines 6 and 7) along the mispredicted (or unauthorized) path will more likely complete fetching the data (secret) into cache, which will be accessed by attacker at a later time. This delay will also correlate to the performance impacts of the various solutions outlined above. For example, the “Undo” technique may need to invalidate more cached data if the delay is longer; or “Delay Load Misses” [18] or our variation SafeLoadOnMiss may delay several load accesses resulting in higher performance penalties.

<table>
<thead>
<tr>
<th>Mitigation technique</th>
<th>Average performance loss (%)</th>
<th>Hardware complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>InvisiSpec-Spectre [16]</td>
<td>5.0</td>
<td>Medium</td>
</tr>
<tr>
<td>InvisiSpec-Futuristic [16]</td>
<td>17.0</td>
<td>Medium</td>
</tr>
<tr>
<td>CleanupSpec [17]</td>
<td>5.1</td>
<td>Medium</td>
</tr>
<tr>
<td>Delay on Miss [18]</td>
<td>11.7</td>
<td>Low</td>
</tr>
<tr>
<td>GhostMinion [20]</td>
<td>2.5</td>
<td>Medium</td>
</tr>
</tbody>
</table>

Our SafeLoadOnMiss technique is similar to [18]. However, our implementation results in smaller performance losses. Also, we collected data for SPEC 2017 benchmarks and not SPEC 2006.

<table>
<thead>
<tr>
<th>Mitigation technique</th>
<th>Average performance loss (%)</th>
<th>Hardware complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>InvisiSpec-Spectre [16]</td>
<td>5.0</td>
<td>Medium</td>
</tr>
<tr>
<td>InvisiSpec-Futuristic [16]</td>
<td>17.0</td>
<td>Medium</td>
</tr>
<tr>
<td>CleanupSpec [17]</td>
<td>5.1</td>
<td>Medium</td>
</tr>
<tr>
<td>Delay on Miss [18]</td>
<td>11.7</td>
<td>Low</td>
</tr>
<tr>
<td>GhostMinion [20]</td>
<td>2.5</td>
<td>Medium</td>
</tr>
</tbody>
</table>

Our SafeLoadOnMiss technique is similar to [18]. However, our implementation results in smaller performance losses. Also, we collected data for SPEC 2017 benchmarks and not SPEC 2006.

If it is possible to turn-on the mitigation only when needed (either when an attack is detected, or while executing certain critical code segments), even the simplest technique (such as preventing speculative execution completely, or preventing speculative memory accesses) will be acceptable since performance penalties occur only when the mitigation is on.

Although we focus on side-channel and speculation-based attacks, the concept of on demand protection is applicable to any known or potentially new attacks. And the on demand security protection can be based on attack detection techniques proposed by researchers or new techniques that may be proposed against new attacks. The techniques should be amenable to turning-on and turning-off on demand. Some techniques may require extensive overheads when the system transitions between safe state and unsafe state, including activating and deactivating certain structures or copying data between hidden buffers (or GhostMinion) and primary cache memories.

To be able to turn-on protection automatically, it is necessary to investigate how an attack can be detected. Side-channel attacks rely on cache accesses, thus most existing methods for detecting side-channel attacks rely on the use of hardware performance counters to extract data of the cache utilization and use machine learning techniques to detect attack patterns.

To summarize, we feel that On Demand security can provide a trade-off between performance and security, by using protection or mitigation against attacks only when an attack is detected (or suspected), or to protect critical sections of applications. One may also Turn On and Turn off security protection randomly to make attacks significantly more difficult to succeed.

---

2 We understand this technique may not prevent Speculative Interference attacks [19]. However, as indicated in [20], such attacks can be made more difficult by relying on random replacement policy. We will discuss this in a later section.

3 The performance numbers and hardware complexity is based on the data from the cited publications.
3. SecurityCloak framework

Our goal is to facilitate On Demand protection against hardware security attacks. In our approach, we explore supporting the ability to protect critical code sections by entering safe modes. For example, this can be achieved using `Enter_Safe_Mode()` and `Exit_Safe_Mode()` system calls as shown in Listing II. Only the process that activated the safe mode can deactivate the protection.

```c
Enter_Safe_Mode();
......
Critical_code
......
Exit_Safe_Mode();
```

Listing II: On Demand Protection

While any security mitigation technique can be relied upon for protection, the techniques should be easy to be turned on and turned off, incurring minimal overheads in switching between safe and unsafe modes. It may also be possible to include multiple mitigation techniques for different types of attacks and users can select a mitigation technique based on the level of security desired, for example, as an argument to `Enter_Safe_Mode()` call.

It may be possible to automatically enable safe mode when an attack is detected (or suspected) - the mitigation selected can be based on the type of attack detected. SecurityCloak framework can use any known detection approaches for some known detection techniques. It is also necessary to determine when it is okay to exit the safe mode. More advanced invocation of safe modes in the presence of persistent or repeated attacks can include techniques similar to "exponential back-off" whereby the amount of time spent in safe mode increases every time the safe mode is invoked. With these goals, we will describe our SecurityCloak design using Gem5 [29]. While in Safe Mode we can use either our Guard Cache to mitigate cache timing attacks or use Safe Queue to delay load misses during speculative execution, or both. To make Speculative Interference attacks [19] more difficult, we use random replacement policy in our caches while in Safe Mode.

3.1. Creating false hits and false misses

Cache side-channel attacks rely on measuring memory access times to determine if an access to a specific cache line (or set) is a hit or a miss: a miss causes longer access times. This observation can be used by an attacker to obtain information regarding which memory addresses a victim accessed, and possibly retrieve data from those addresses.

Victim Caches are generally very small and fully associative caches and were originally used to improve performance by eliminating cache thrashing in direct mapped caches [21]. In a more recent work [30], victim caches were used to house data evicted by speculative load accesses (for example, ReViCe [30]). We feel this requires complex bookkeeping since one needs to distinguish between speculative and non-speculative load accesses, as well as remove misspeculated data from victim cache. We use Guard Caches similar to Victim Caches to create false hits — any data item evicted from the primary cache is saved in the Guard Cache. If the evicted item is accessed, it can be retrieved from the Guard Cache, making the access appear as if it was a hit in the primary cache, since the access times to a Guard Cache and primary cache are comparable. We also rely on random replacement policy when entries in the Guard Cache need to be replaced, to further obfuscate information leak. Also, not every data evicted from primary cache is placed in the Guard Cache but treated as a normal cache miss. We will also use the Guard Cache as a Miss Cache [21] – the missing data is brought into the Guard Cache, unlike in the case of a victim cache where the missing data is brought into the primary cache and the evicted data is stored in the Guard Cache. Such data items are likely to be short lived in Guard Cache unlike when the data is brought to primary cache since Guard Cache is very small compared to primary caches. This can add to additional obfuscation to cache timing. These different uses of the Guard Cache make it difficult for an attacker to discover the presence of a Guard Cache, its size or when it is used or not used. We saw negligible performance gains or losses with Guard Caches: larger Guard Caches can provide more protection since victims...
can be held for longer periods of time but can lead to higher silicon area and consume more power. We found that even a small Guard Cache (1 KiB or 2 KiB at L1 level and 2 KiB to 4 KiB at L2 or LLC levels), is sufficient to prevent several types of side-channel attacks.

We create false misses by randomly evicting cache lines. On every L1-D (or L2) cache access that is a hit, we select a cache line randomly and evict the selected data based on the eviction frequency but do not place it in the Guard Cache. We varied the eviction frequency from 5% to 40%. The random evictions lead to performance losses but if the percentage of evictions is kept around 10% (which is sufficient to prevent currently known side channel attacks), the loss is reasonable (in the order of 33%) compared to turning off speculation completely. Additionally, if the random evictions are used only when in Safe Mode the performance losses can be reasonable. For example, if the random evictions are in place only for 10% of an application execution, assuming an attack will only last that long or critical sections constitute only 10% or execution times, the performance loss is only 2% (see Fig. 11).

The false misses will make attacks using such techniques as Evict & Time [3] and Prime & Probe [3,4] more difficult since the attacker will need to capture every evicted data, an attacker may be able to deduce the size of the Guard Cache. That is why we propose to randomly change the frequencies of evictions can be varied to achieve desired levels of protection. Section 4.3 provides a detailed analysis of the effectiveness of Guard Cache while Section 4.4 describes the effectiveness of random evictions.

We can deploy both false hits and false misses together to increase the randomization of cache timing. Fig. 3 shows the results from a simulated Prime & Probe attack. The left column shows the normal mode indicating what attacker sees as cache misses caused by the victim’s code (evicting attackers primed data). The middle column shows that the attacker sees additional cache misses caused by false hits. The right column shows the case when the Guard Cache is used to cause false hits, and it can be seen that the attack is not successful (the characters of the secret key (The Magic Words) are visible). The right-hand side of the figure shows the case when a Guard Cache is used to cause false hits, and it can be seen that the attack is not successful (the characters of the secret key (The Magic Words) are visible). The right-hand side of the figure shows the case when a Guard Cache is used to cause false hits, and it can be seen that the attack is not successful (the characters of the secret key (The Magic Words) are visible). The right-hand side of the figure shows the case when a Guard Cache is used to cause false hits, and it can be seen that the attack is not successful (the characters of the secret key (The Magic Words) are visible).

The use of the Guard Cache causing false hits may prevent attacks that rely on Prime & Probe. The left-hand side of Fig. 4 shows a successful attack using a proof-of-concept code from [17]: characters of the secret key (The Magic Words) are visible. The right-hand side of the figure shows the case when a Guard Cache is used to cause false hits, and it can be seen that the attack is not successful (the characters of the secret key (The Magic Words) are visible).

Speculative attacks are based on flushing array bounds variables from caches, leading to delays in checking for out-of-bounds accesses (since the array bounds variables are not in the cache) and the attacker can rely on speculative execution to bring large amounts of out-of-bounds data to the cache during this delay. Our Guard Cache prevents such attacks since it will capture the flushed array bounds variable in Guard Cache, reducing the time for bounds check, and limiting the accesses to out of bounds data.

For attack models based on cache timing analyses, our Guard Cache and random evictions will make attacks significantly more difficult as the number of hits and misses will change. If the Guard Cache is used to capture every evicted data, an attacker may be able to deduce the size of the Guard Cache. That is why we propose to randomly change the fraction of evicted data that is stored in the Guard Cache, making it difficult for the attacker to observe the size of the Guard Cache.

As shown in the table at the bottom of Fig. 4, even a 1 KiB Guard Cache at L1-D level obscures data during a Spectre attack and prevents the attack, if the attacker is using Flush & Reload side-channel. False misses are more effective against Prime & Probe side-channel attacks. While 6% random evictions are less likely to prevent this attack, 8% will partially mitigate the attack by obfuscating some characters, and 10% or higher rates of evictions will prevent the attack. However, even 6% random evictions will significantly vary cache timing, making it more difficult for the attack to be successful. These numbers are based on the specific proof of attack codes, but the sizes of Guard Caches and the frequencies of evictions can be varied to achieve desired levels of protection. Section 4.3 provides a detailed analysis of the effectiveness of Guard Cache while Section 4.4 describes the effectiveness of random evictions.
3.2. Delay speculative load misses or SafeLoadOnMiss

While False Hits and False Misses may be used to mitigate speculative attacks using proper Guard Cache sizes and random eviction frequencies, we also present a technique that delays any speculatively issued loads that miss in L1-D cache. This is similar to the one proposed in [18]; however, we queue speculative load misses in a separate queue and process the misses as soon as these loads become non-speculative, without reissuing the load request as done in [18].

3.2.1. SafeLoadOnMiss

In modern processors, the Load Store Queue (LSQ) is responsible for keeping track of all the memory/cache requests (loads, stores, cache maintenance, etc.). The LSQ (see Fig. 5) can speculatively send a load request to the L1-D cache for each load instruction when the corresponding address generation is complete and the load is not aligned with a previous ready-to-commit store present in the LSQ. Even though the stores can be executed speculatively, the LSQ only writes back the store requests when they become non-speculative (ready-to-commit). When there is a misprediction and the pipeline is flushed, partially/fully executed instructions (loads and stores) in the LSQ are flushed. These load requests, store requests, and other cache maintenance requests like cache flush requests are queued before the L1-D cache until the cache controller serves them. From here onward we use the term Mandatory Queue (Fig. 5) for the queue which holds all the requests to the L1-D cache (as implemented by Ruby coherence protocol in Gem5 [29]). The cache controller fetches the requests from the Mandatory Queue and performs necessary actions. When there is a cache miss, load/store request is added to the Miss Status Holding Register (MSHR), and a new request is sent to the next level cache or the memory. The speculatively executed loads that miss the cache are responsible for bringing the secret information directly and/or indirectly when the processor is under attack.

We propose a new structure called Safe Queue (similar to the Mandatory Queue) to delay the load requests that miss in the cache. Initially,
the load request arrives at the Mandatory Queue \(^1\) and is read by the cache controller. When the cache controller identifies that the load is a miss, instead of adding the request into MSHR and sending a new request to the next levels of the memory hierarchy, it pushes the request to the Safe Queue \(^2\). This mechanism blocks all the load misses and lets the load hits to be served by the cache controller. Since the store requests are always non speculative, there is no need for them to be blocked even if it is a cache hit or miss. It is important to note that the Mandatory Queue is specific to Ruby coherence protocol in Gem5 \([29]\) but other micro-architectures may use similar structures. Safe Queue is the only additional buffer required in our design apart from the minor changes to the control logic.\(^4\)

Gem5 \([29]\) maintains a monotonically increasing unique sequence number \((1, 2, 3, 4 \ldots)\) for each instruction entering the pipeline (similar to the timestamp in Ghostminion \([20]\)). This sequence number is also added to the metadata of all the cache requests. When the processor determines a branch misprediction and squashes (or flushes) the pipeline, the LSQ generates a new cache request called squash request with the sequence number of the mispredicted branch instruction. When the cache controller receives the squash request, it drops all the load requests which have a higher sequence number than the mis-predicted branch from both Safe Queue and Mandatory Queue.

When there is a non-speculative load that is a miss (true program misses), the processor stalls because the load request is blocked in the Safe Queue. To release the load, we generate another new cache request called non-speculative request with the sequence number to communicate that the load is safe to be served. The LSQ generates the non-speculative request whenever the load instruction becomes the ROB head (i.e., becomes not speculative). When the cache receives the non-speculative request, it moves the corresponding load from Safe Queue to the top of the Mandatory Queue \(^3\). This will ensure that the load is served as quickly as possible to reduce the stalling time. The determination of the load request being non-speculative can also take advantage of Bell and Lipasti conditions \([31]\) rather than waiting for the load to reach the head of ROB. This helps moving requests from the Safe Queue to Mandatory Queue sooner and slightly improve the performance. Each entry in the Mandatory Queue maintains a flag to indicate whether it is speculative or not. The cache controller allows the requests to proceed if the flag is set to “non-speculative” even if it is a load miss. When a request is moved from Safe Queue to Mandatory Queue upon a non-speculative request, the flag is set to “non-speculative” to avoid possible deadlock.

By using the above mechanism, we avoid any changes to the cache state by the speculative load misses and block the attacker from bringing secret information to the cache. In \([18]\), the authors propose to drop the load misses and allow the LSQ to retry later. The repeated tries may lead to uneven delays in processing correctly predicted loads that miss in the cache. This may explain in part why the performance losses reported in \([18]\) are higher than those reported in this paper. In the rare case when the Safe Queue is full, load misses are dropped and the LSQ must retry when the resources are available (similar to the mechanism when the MSHR is full). As we show in Section 4.8, even a small 16 entry Safe Queue is more than adequate to track speculative load misses for SPECspeed 2017 benchmarks. Thus, it is very unlikely that the Safe Queue will be full.

3.2.2. Real-time attack detection

Speculative execution attacks rely on widely known cache side channels such as Flush & Reload \([5]\) and Prime & Probe \([3,4]\) to infer the secret information. The Flush & Reload side channel has a higher reliability and less noise compared to Prime & Probe and is often used in most speculative attacks \([32]\). We observed that the Flush & Reload uses an excessive number of cache flush instructions (CLFLUSH in X86, MCR in ARM) and it can be used as an indicator of an attack underway. Instruction address and permission bits can be used to distinguish a cache flush used by an attacker and system activities (e.g. TLB shootdown). Innocuous user programs rarely use cache flushes and if it does, the program will trigger a false positive. This detection technique can be improved using (off-line) machine learning algorithms to identify the nature and frequency of flush instructions occurring when an attack is underway, including the likelihood of the same cache line/ set being flushed repeatedly, and higher L2/L3 cache occupancy. In this paper, we utilize a more conservative and simple approach to detecting an attack whenever a cache flush request is present. Evaluating different detection techniques published in the literature for their suitability for real-time attack detection and accuracy of detection is left as future work.

3.2.3. Alternate between safe and unsafe modes

As explained in Section 3.2.2, when an attack is detected or when Enter_Safe_Mode() call is made, a register/flag is updated to indicate that the system is switched from UnsafeBaseline to SafeLoadOnMiss. The L1-D cache will start moving all load misses to the Safe Queue and the LSQ starts generating necessary squash and non-speculative requests to remove or release load misses. Our experiments with the Proof of Concept attack presented in \([17]\) (see Listing I, in Section 2) showed that waiting in the SafeLoadOnMiss mode for 1000 committed instructions and switching back to UnsafeBaseline after detecting an attack if no additional cache flushes occurred is sufficient to successfully avoid the attack.

When switching from SafeLoadOnMiss to UnsafeBaseline, L1-D cache stops any new load misses being placed in Safe Queue. However, the LSQ keeps generating squash and non-speculative requests until the Safe Queue becomes empty. A system register is maintained to indicate the number of valid entries in the Safe Queue. It is also possible to immediately remove all Safe Queue entries when switching from safe to unsafe and slightly improve the performance. However, this may increase the hardware overhead for the Mandatory Queue (see Section 4.8) and the complexity of the design. When the Safe Queue becomes empty, the system fully recovers from the Safe Mode and continues working under high performance (and unsafe) mode. This mechanism allows us to easily switch back and forth without significant overhead which is one of the main goals of the SecurityClock framework.

3.2.4. Random replacement policy

In the baseline UnsafeBaseline mode, we use the Tree-PLRU \([33]\) replacement policy (default policy of Gem5 \([29]\) Ruby caches) for all cache levels. However, the LRU based replacement variants are susceptible to attacks such as Speculative Interference \([19]\). The attacker can exploit the replacement state changes (e.g. loads reordering) made by the speculative execution to indirectly recover or infer secret information. We use random replacement policy at L1-D cache and L2 shared cache levels to mitigate such attacks. Table 2 shows the average performance loss for SPEC2017 benchmarks when the random replacement policy is used for the baseline (UnsafeBaseline) system compared to using Tree-PLRU replacement policy. As can be seen, the use of random replacement policy does not cause a significant performance loss and since we use the random replacement policy only in Safe Mode, the performance loss due to random replacement will be even less.

When the system detects an attack and switches from UnsafeBaseline to SafeLoadOnMiss, cache replacement policy is also changed from Tree-PLRU to random. When the system exits Safe Mode, the cache replacement reverts to Tree-PLRU. This switching between the replacement policies may obfuscate some side-channel attacks that depend on cache meta-data.

\(^{4}\) Although our implementation is based on Ruby system in Gem5, our approach of using a Safe Queue can be implemented in any microarchitecture.
Table 2: Average performance loss due to random replacement policy compared to Tree-PLRU.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Slowdown (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1-D Random replacement</td>
<td>0.3</td>
</tr>
<tr>
<td>L2 Random replacement</td>
<td>3.2</td>
</tr>
<tr>
<td>Both together</td>
<td>3.4</td>
</tr>
</tbody>
</table>

Table 3: System configuration.

<table>
<thead>
<tr>
<th>Core</th>
<th>1-core Out-of-order, no SMT, 2 GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipeline</td>
<td>192-entry ROB, 64-entry IQ, 32-entry EU, 32-entry LB, 256 Inst/256 FP registers</td>
</tr>
<tr>
<td>Tournament</td>
<td>2-bit, 2048-entry local, 8192 global, 8192 choice, 4096 BTB, 16 RAS</td>
</tr>
<tr>
<td>Predictor</td>
<td>8192 choice, 4096 BTB, 16 RAS</td>
</tr>
<tr>
<td>Caches</td>
<td></td>
</tr>
<tr>
<td>L1-I Cache</td>
<td>32 KiB, 4-way, 2-cycle latency</td>
</tr>
<tr>
<td>L1-D Cache</td>
<td>64 KiB, 8-way, 2-cycle latency</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>2 MiB, share, 16-way, 20-cycle latency</td>
</tr>
<tr>
<td>Coherency</td>
<td>Directory based MESI Ruby protocol</td>
</tr>
<tr>
<td>Rest of the System</td>
<td></td>
</tr>
<tr>
<td>Memory</td>
<td>DDR3_1600_8 x 8, 8 GiB</td>
</tr>
<tr>
<td>Safe queue</td>
<td>64 entries</td>
</tr>
<tr>
<td>Mandatory queue</td>
<td>64 entries</td>
</tr>
</tbody>
</table>

4. Experimental evaluation

In this section, we describe our experimental setup and evaluation of our techniques for mitigating side-channel attacks and attacks based on speculative execution. Our techniques were described in Section 3.

4.1. Experimental setup

We evaluate our design using Gem5 [29] System-call Emulation (SE) mode to accurately model the high performance X86 system showed in Table 3 (similar to prior works [16,17,20]). We implemented all the features described in Section 3 and in this section we will include an evaluation of our SecurityCloak framework. For this contribution, we focused on a single core. However, we are confident that the approach can be extended to multicore systems and multiple levels of caches. We will investigate the multicore implementations in our future work.

4.2. Workloads

In our experiments, we use 14 SPECspeed 2017 benchmarks with the reference data set. We forward the execution by 10 billion instructions to generate the checkpoint and simulate 500 million instructions. Important characteristics of the workloads are shown in Table 4. As discussed in Section 2, Branch Misprediction Rates and L1-D Cache Miss Rates play a significant role on the performance lost due to any security protection mechanism. For our mitigation technique of delaying speculative cache misses, higher L1-D miss rates correlate to higher performance loss since more load instructions will be delayed until the speculation is resolved. Likewise, higher number of branches and lower branch mispredictions can impact the performance lost since more branches cause more speculations and when the misprediction is low, delayed load instructions will need to be completed. To better understand the absolute number of branch mispredictions and load misses present in the workload, we include the MPKI (Misses/Misleveled Instructions) values in Table 4.

4.3. Analysis of false hits

In Section 3.1 we have shown that even a small Guard Cache (1 KiB or 2 KiB at L1 level) can prevent attacks that rely on Flush and Reload side-channels. Here we present the performance impacts caused by our Guard Cache for several different SPEC 2017 benchmarks. We
The benchmark lbm has very high L1-D miss rates (21.36 misses per 1000 instructions) and 10% of demand misses are brought into Guard Cache (used as victim cache). As can be seen, the results in Fig. 6 show very minimal impact on performance ranging between −0.2% to 3.0% performance loss. Negative bars indicate performance gains — LU replacement policy for primary caches results in performance gains than when Random Replacement is used. The use of Guard Cache as a Miss Cache results in slightly higher performance losses than when used as a victim cache.

Fig. 7 shows some memory access behaviors of applications including average number of data accesses per 1000 instructions and average number of cache misses per 1000 instructions (first four columns in the figure). The figure also shows the average number of L1-D and L2 misses that are satisfied by the Guard Cache per 1000 instructions executed. Guard Cache is likely to result in performance benefits when the application exhibits higher cache conflicts; this can be seen from a higher percentage of false hits. Consider cactus with 2.92 L1-D misses per 1000 instructions without a side channel attack. As can be seen, the results in Fig. 6 show very minimal impact on performance ranging between −0.2% to 3.0% performance loss. Negative bars indicate performance gains — LU replacement policy for primary caches results in performance gains than when Random Replacement is used. The use of Guard Cache as a Miss Cache results in slightly higher performance losses than when used as a victim cache.

The benchmark lbm has very high L1-D miss rates (21.36 misses per 1000 instructions) and 10% of demand misses are brought into Guard Cache (used as victim cache). As can be seen, the results in Fig. 6 show very minimal impact on performance ranging between −0.2% to 3.0% performance loss. Negative bars indicate performance gains — LU replacement policy for primary caches results in performance gains than when Random Replacement is used. The use of Guard Cache as a Miss Cache results in slightly higher performance losses than when used as a victim cache.

4.4.4.1. Using Prime & Probe side-channel

If Prime & Probe is used, attacker first “primes” (or fills) the 256 entries of his array. When victim accesses a character in the key, the attacker injects code uses the value of the character accessed by the victim. The attacker can use either a Prime & Probe or Flush & Reload side-channel to find the key.
Fig. 8. Random evictions (False misses) at different eviction frequencies in L1D cache. The minimal cache misses evicted are the randomly evicted cache lines that attacker encounters as misses which can mitigate the chance of a successful attack.

To understand the relationship between the random eviction frequency and the fraction of total cache that the attacker primes and later probes (we call this cache occupancy), consider an extreme case where the attacker primes only 2 cache lines, assuming that the key is one of two different characters. If the cache contains 1024 lines, there is only one-in-512 (or 0.002) chance that a random eviction removes one of the two attacker's cache lines. The probability that we can randomly evict both attacker cache lines so that the attacker will not be able to find the correct key is at most 1/(512*512) but likely much lower as a random eviction may evict the cache line multiple times. On the other hand, consider the case where the attacker primes and probes 1024 cache lines (the entire size of the cache). Every random eviction will evict an attacker's data making it very easy to prevent attacks.

Fig. 8 shows more details about the effectiveness of random evictions for different eviction rates. The column “Minimal Cache Evictions” indicates the minimum number of random evictions that fall within the attacker's 256 cache lines. The attacker repeats the Prime & Probe process 5 times for each character. Since only the cache line accessed by the victim will evict attacker data, repeating Prime & Probe process improves the confidence in the key detected. To prevent the discovery of the key, our random evictions must also repeatedly evict the same cache lines (besides the key). This is likely to happen if the “Minimal Cache Evictions” is high. As can be seen at 10% random eviction frequency, at least 83 of the attacker’s data is evicted on every attempt, and for the given Proof of Concept this appears to be more than sufficient to completely prevent the attack. However, at 8% frequency, attacker was able to discover some of the characters in the key but not all. It should also be noted that the attack may be successful even at 10% evictions if the Prime & Probe process is repeated more than 7 times for each character. At higher random eviction frequencies (say at 30% or 40%), the minimal cache misses encountered by the attacker reaches 256 (all of attacker's data), and the attacker is unlikely to succeed even after numerous attempts.

Another data included in Fig. 8 is how the difficulty presented to the attacker increases when random evictions are used. The sixth column shows the minimum number of times the attacker must repeat the Prime & Probe to discover the victim's key characters. As we stated previously, the Proof of Concept attack repeats the Prime & Probe 5 times to discover each character (a total of 200 times to obtain the full 40-character key). But the data in Fig. 8 shows that as the random evictions are used, the attacker must repeat the Prime & Probe many more times. For example at 10% random eviction rate, the
While higher random eviction frequencies aid in the prevention or make it much more difficult for the attack to succeed. Additionally, since in our implementation, we only consider character that is part of the key. Thus, random evictions either prevent the attack or make it much more difficult for the attack to succeed. However, random eviction frequencies aid in the prevention or mitigation of attacks, since the performance loss increases with higher frequencies, one should rely on such higher random evictions for critical code sections as we will describe later in this section and shown in Fig. 11.

In summary, the number of cache lines used by the attacker in relation to the total size of the cache (or cache occupancy) and the number of times an attacker repeats the priming and probing determines the random eviction frequencies needed to successfully obfuscate the attack. Additionally, since in our implementation, we only consider (random) evictions on a cache hit, the probability of evictions also depends on the cache hit rates. For example, when the random eviction frequency is very high, our method evicts most of the cache lines, causing fewer hits, and this in turn, reduces the probability of further evictions. In our experiments, frequencies higher than 30% lead to the eviction of most, if not all, of attacker's 256 array elements. Additional evictions can only evict the attacker's data repeatedly and lead to no further obfuscation.

4.4.2. Using Flush & Reload side-channel

The attacker can also use Flush & Reload side-channel attack to discover this key. In this method, the attacker flushes his/her (256) array elements (in addition to flushing array size as described in Section 2). When the attacker injected code makes the victim access one of these array elements using the key character as an index, a cache line of the attacker array will be filled and becomes valid. The attacker reloads each of 256 array elements to see which of his/her array elements causes a hit to discover the key. Once again, the attacker repeats the experiment several times to be confident of the array element that was accessed by the victim. For our random evictions to be successful, we need to evict the array element accessed by the victim, after the victim accesses it but before the attacker checks (this would cause a miss when the attacker reloads the data). This requires a very high frequency of random evictions. As we have shown, higher eviction frequencies can lead to excessive performance loss. But using Guard cache as a victim cache can prevent Flush & Reload based attack as shown in Fig. 4. The data flushed by attacker will be captured in the Guard Cache and when the attacker reloads the data, most, if not all, array elements appear to be hits, thus obfuscating the discovery of victim's key characters.

4.4.3. Performance impact of false misses

We now present the performance impact of random evictions. Fig. 9 shows the performance loss for SPEC 2017 benchmarks when cache lines are randomly evicted. On every cache hit (either at L1-D or L2), we decide if a random cache line should be evicted based on the eviction frequency. The data in Fig. 9 is for different eviction frequencies. A higher frequency of evictions will cause higher performance losses. For example, if a cache line is evicted 20% of the time a L1-D cache access is a hit, we see a geometric mean performance loss of 90% for SPEC 2017 benchmarks. This is an unacceptable performance loss; however, it can cause significant obfuscation of cache access times. As previously shown in Fig. 4, 10% frequency of random evictions is adequate to cause sufficient obfuscation to prevent Prime & Probe side-channel attacks. A 10% random eviction frequency leads to a 33% geometric mean performance loss for SPEC 2017 benchmarks.

In our experiments, we evicted both modified and unmodified data from caches. The performance loss due to random evictions can be minimized if only unmodified data is selected for random evictions. However, an attacker may circumvent the impact of random evictions by repeatedly writing the same data.

Fig. 9 shows that the performance loss at L2 due to random evictions is significantly smaller since there are significantly fewer accesses to L2. We only select cache data for eviction when L2 cache is accessed and the access is a hit.

Fig. 10 shows additional cache misses per 1000 instructions encountered by applications that are caused by random evictions. The figure also includes L1-D cache accesses per 1000 instructions and cache misses for 1000 instructions in the baseline (without random evictions). Since we apply random evictions on every L1-D cache hit, higher cache hits in the baseline can lead to more frequent random evictions. On the other hand, applications that have higher miss rates will likely see less impact due to random evictions as fewer additional cache misses will be encountered by the applications. Streaming applications may not see the effects of false misses since the randomly evicted data may not be accessed by the application. Random evictions may even be beneficial for such applications since when a truly missing data needs to be brought into the cache, there is no need to find a Least Recently Used cache line for replacement as the cache contains many invalid entries. Consider Ibm and wrf_s, both have about the same number of L1-D accesses per 1000 instructions, (see Table 4) but Ibm has higher miss rate (MPKI of 21.36 compared to 2.97 for wrf_s). With higher hit rates and lower miss rates wrf_s encounters additional cache misses with random evictions. The benchmark Ibm with higher miss rates and lower hit rates encounters fewer additional misses caused by random evictions. The application mcf has higher L1-D accesses and higher miss rates which explains the higher number of additional cache misses encountered by the application. The benchmark exchange2_s has fewer L1-D accesses but very low miss rates — indicating that most of the accesses are hits which causes a higher number of random evictions. It should be remembered that higher false misses can aid in further mitigating side-channel attacks.

4.4.4. Using random evictions only when needed

Since high random eviction rates can cause significant performance losses, this method should be used only when needed, for example, to protect critical code sections or when an attack is detected. To simulate turning on protection only when needed we experimented by turning-on false misses (or random evictions) only for a fraction of the application execution time. For example, when the false miss strategy is enabled 10% of the execution time of an application, false misses are introduced

### Table: Additional L1 misses per kilo instructions for different frequencies of random evictions

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Baseline L1-D Accesses per 1000 instructions</th>
<th>Baseline L1-D Misses per 1000 instructions</th>
<th>Additional L1 D Accesses</th>
<th>Additional L1 D Misses</th>
<th>MPKI</th>
</tr>
</thead>
<tbody>
<tr>
<td>lbm_s</td>
<td>10%</td>
<td>20%</td>
<td>30%</td>
<td>40%</td>
<td>50%</td>
</tr>
<tr>
<td>mcf_s</td>
<td>10%</td>
<td>20%</td>
<td>30%</td>
<td>40%</td>
<td>50%</td>
</tr>
<tr>
<td>x264_s</td>
<td>10%</td>
<td>20%</td>
<td>30%</td>
<td>40%</td>
<td>50%</td>
</tr>
<tr>
<td>Geomean</td>
<td>0.71</td>
<td>0.71</td>
<td>0.71</td>
<td>0.71</td>
<td>0.71</td>
</tr>
</tbody>
</table>

Fig. 10. Additional L1 misses per kilo instructions for different frequencies of random evictions.
for 50 million instructions (out of 500 million instructions simulated in our experiments), Fig. 11 shows the geometric mean performance losses for the SPEC 2017 benchmarks. As can be seen, if random evictions are applied only 10% of the applications’ execution, we only see a geometric mean performance loss of 5% at 10% random eviction rate at L1-D level (not 62% if the random eviction are turned on during the entire execution). Even when false misses are introduced for half of the application execution, the geometric mean performance loss is 26% at 10% random eviction rate. We feel that security protection should be used only when needed — to protect critical segments of applications which minimizes performance losses.

4.5. Combined analysis

In the final set of experiments, we used both Guard Cache (i.e., false hits) and random evictions (i.e., false misses). The performance losses are similar to those when only false misses are in place. The performance impact of Guard Cache was negligible. The results are very similar to those shown in Fig. 9.

4.6. Evaluation of SafeLoadOnMiss technique

Table 5 shows the normalized execution times with respect to UnsafeBaseline (that does not rely on any mitigation techniques) for (i) NoSpeculation, whereby speculative execution is disabled, (ii) SafeLoadOnMiss using LRU replacement policy and (iii) SafeLoadOnMiss with random replacement policy. TreePLRU replacement (default in Gem5 [29] Ruby) policy is used in all cache levels in UnsafeBaseline, NoSpeculation and SafeLoadOnMiss modes. Here, the Safe Mode is on for the entire execution duration of applications.

As expected, completely restricting the speculation results in highest performance loss: 89.6% on average. It should be observed that fotonik3d has a very low branch misprediction rate (see Table 4) leading to significant performance loss when speculation is disabled. On the other hand, mcf has high branch misprediction rates and thus preventing speculation leads to lower performance losses. When the cache MPKI is low but branch MPKI is high, the performance loss will be lower. Disabling speculation completely does provide protection against all speculative attacks but it cannot be justified because of the excessive performance losses.

Table 5 also shows that delaying speculative load misses (SafeLoadOnMiss) with and without the random replacement policy results in very small performance penalties of 5.4% and 2.1% respectively. Even though the use of random replacement policy increased the performance loss, it will improve the attack coverage by mitigating threats such as Speculative Interference attack [19] (as detailed in Section 3.2.4). Although fotonik3d and exchange2 have lower misprediction rates, they also have lower L1-D MPKI. This means fewer delays for load instructions in speculative execution, leading to almost negligible performance loss. The benchmark bwaves has no significant loss due to extremely low branch misses (Branch MPKI = 0.00). On the other hand, higher branch MPKI and cache miss MPKI values lead to higher performance losses for mcf and gcc. Even though ibm has the highest L1-D MPKI value, its low branch MPKI was able to balance out the performance penalty by utilizing the instruction level parallelism. In summary, SafeLoadOnMiss performs poorly when both branch mispredictions and L1-D misses are high and having a lower value for either one of the miss parameters seems to cancel out the performance effect. Adding the random replacement policy on top of SafeLoadOnMiss showed an extra slowdown of 3.3% (5.4%-2.1%) which aligns with the performance loss added to the UnsafeBaseline (3.4%, see Table 2).

4.7. Switching between safe and unsafe modes

Launching attacks while running a real application such as SPECspeed benchmark is very difficult and may require modifications to benchmark codes (which is not recommended). Finding a real world working Spectre attack example capable of doing actual damage is out-of-scope for this research. To emulate the attack environment (either to protect critical code sections or when an attack is detected), we enter the SafeLoadOnMiss mode for 10%, 20%, and 30% of the time and switch back to UnsafeBaseline the rest of the time. To avoid catching a specific section of the workload where the load miss activity or the branch activity is low, we distribute the partial SafeLoadOnMiss modes (10%, 20% and 30%) uniformly across the entire execution of an application. We used 1000 equi-width chunks for each partial SafeLoadOnMiss mode (enter and exit SafeLoadOnMiss mode 1000 times) and uniformly distributes them across the 500 million instructions of the simulation window. We switch to the random replacement policy whenever we are in the SafeLoadOnMiss mode. Fig. 12 shows (on Y-axis) the percentage of performance loss due to the SafeLoadOnMiss technique when compared to an unprotected baseline system if the protection is on only for short periods of time (10%, 20%, 30% of the execution time of an application) as well as when the protection is on during the entire execution of an application (100%). As can be seen from the figure, the average performance degradations when the protection is on for only 10%, 20%, and 30% of execution of an application are 1.2%, 1.9%, and 2.5% respectively. As shown in Table 5, we see a 5.39% loss if the protection is on all the time (see Fig. 13).

4.8. Hardware overhead

As mentioned in Section 3.2.1, the only additional buffer structure required in our design is the Safe Queue. Each Safe Queue entry contains the address of the load and the corresponding sequence number. If we assume the address is 64 bits and the sequence number and additional flags (non-speculative flag, Section 3) require 16 bits, each entry is
F. Mosquera et al.

Fig. 12. Percentage of performance loss using SafeLoadOnMiss mitigation technique for specspeed 2017 benchmarks, normalized to unprotected (or unsafe) baseline. The data is for the cases when the SafeLoadOnMiss scheme is active with random replacement policy for 10%, 20%, 30%, and 100% of the total execution time.

Fig. 13. On-Demand security protection. (a) The normal unsafe mode where the specpre attack is successful. (b) The Safe Mode is on throughout the execution and the attack is unsuccessful. (c) The Safe Mode is entered and exited at random intervals: Attack is prevented in Safe Mode but the attack is successful when the Safe Mode is exited.

10 bytes long. In our experiments, we observed that the maximum occupancy of the Safe Queue is 13 (gcc). The size of the Safe Queue should be adjusted according to the issue width of the core pipeline and the number of requests that can be served in parallel by the cache port. Thus, for our experimented design a minimum of 16 entries are enough for the Safe Queue. This results in an additional buffer requirement of 160 bytes, compared to Ghostminion which uses 2 KiB storage [20], CleanupSpec which uses 1 KiB [17]. Thus, the additional cost of 160B per core is significantly lower and the control logic involved in our design is easy to implement. Likewise, as shown in Section 4.3, 1 KiB to 2 KiB Guard Cache is sufficient to prevent many cache timing based attacks.

5. Related work

The research community has been very active in the area of hardware security as can be seen by numerous publications in recent premier architecture and systems conferences. Thus it is not possible to review all such works. We have already described some key publications that are very closely related to our research. We will describe some additional works that are useful in understanding the types of attacks, detecting attacks, hardware and software mitigation techniques.

5.1. Types of attacks based on timing cache accesses

Based on the access latency, an attacker can deduce whether or not a cache line is in the cache through two methods to set up the cache, which in turn may reveal some information about the accesses made by a victim [3,4,22,23]. Some commonly referenced attacks include:

- Evict & Time attack [3], in which the attacker evicts a cache line triggering a miss when the victim accesses the data item.
- Prime & Probe [3,4] where the attacker “primes” by occupying all cache lines. Later the attacker probes to see which cache lines caused misses, inferring the lines accessed by the victim.
- Two variations of Flush & Reload [5] which are Evict & Reload [22] and Flush & Flush [23]

5.2. Mitigating cache side-channel attacks

In general, the solutions to protect against timing attacks are implemented to avoid sharing of caches so that attacker cannot evict the victim’s cache lines and time accesses. Another approach obfuscates how the addresses are mapped to cache sets, consequently for the attacker is more difficult to determine any portion of the addresses accessed by the victim. In Partition Locked Cache (PILcache) [34], each cache line is augmented with an ID field and a lock bit L. Dynamically Allocated Way Guard (DAWG) [11] is a mechanism to secure way partitioning of set associative caches. DAWG isolates hits, misses, and metadata updates across protection domains. There are several proposals for randomizing mapping addresses to cache sets, including Random Permutation Cache (RPcache) [14,34] and NewCache [35].

---

6 Sequence number only needs to be wrapped around after twice the size of ROB (similar to timestamp in [20]). Thus, in this proposal 9 bits are enough to represent the sequence number (192 * 2 < 2 ** 9).
5.4. Mitigating speculative attacks

Software Based. Return trampoline or “Retpolines” [40,42] isolates indirect branches from speculative execution. It uses properties of the Return Stack Buffer (RSB) as a prediction structure to control speculation. Indirect Branch Control (IBC) is an architectural mechanism which has been added to the x86 ISA to help software control of branch prediction of indirect branches. It consists of 3 features: Indirect Branch Prediction Barrier (IBBP), Indirect Branch Restricted Speculation (IBRS) and Single Thread Indirect Branch Predictors (STIBP).

Jump Switches [43] is a software protection against Spectre Variant 2 with 2 less overhead than retpolines solution. It transforms indirect calls into conditional direct calls. Conditional Speculation [44] and SpectreGuard [45] use software hints to disable speculation on sensitive instructions. New Operating Systems have added Kernel Page Table Isolation (KPTI) [46] to fix meltdown attacks, where the kernel memory space is separated from user space and uses a separate page table to prevent speculative read of kernel memory from user-space; however at the cost of an additional page table switch during every transition from user space to kernel-space and back, on every syscall instruction.

Hardware Based. There are several approaches that rely on delaying visibility of speculative accesses, including Selective Delay [18] and InvisiSpec [16]. InvisiSpec [16] uses speculative buffers for executing instructions in speculative mode, including speculative memory accesses. Upon a correct prediction, the memory accesses are reissued (hence known as “redo” technique).

A different technique, CleanupSpec [17] can be viewed as an “Undo” approach: it evicts speculatively fetched data from caches (and uses random replacement to obfuscate attacks that rely on metadata such as LRU information). Revice [30] is similar to CleanupSpec [17]: it allows speculative caching of data but hides the operation until it is safe, in part by introducing access jitters to conceal access timing. These techniques are fundamentally based on exposing or making a permanent cache state only after the Visibility Point (VP), when the instructions cannot be squashed by prior operations (that is, when the speculation is correct).

In [18], the authors propose three techniques to mitigate Spectre-like attacks. They explore two versions of delaying speculative loads that miss in the cache and a third technique that uses value prediction for load misses. Delaying load misses is similar to our technique proposed here. However, our implementation is different and our technique can be turned on when needed. We also propose random replacement policies for cache memories to make Speculative Interference attacks [19] more difficult.

Ghostmimion [20] uses a separate cache like structure (Ghostmimion) to hide speculative accesses and associates time stamps with the accesses to assure both non-interference of speculative instructions with (earlier) non-speculative instructions and also visibility of (metadata) later accesses to earlier access, which can occur with out-of-order speculative executions. While this is a potentially complete solution against Speculative Interference attacks [19], it may not be easy to use for on demand protection.

6. Conclusion

In this paper, we presented the SecurityCloak framework — an On Demand protection against side-channel and speculative execution attacks. Systems can be protected by entering Safe Mode and turning on protection or exiting Safe Mode to resume normal execution. Safe Mode can be entered when an attack is detected or when a user wants to protect critical code sections. Any known technique to prevent (or at least mitigate attacks) can be used within our framework. However, since the main goal is the ability to dynamically turning on and turning off protection, a mitigation technique should incur minimal overheads when entering or exiting Safe Mode. In this paper, we rely on very simple techniques to protect against side-channel attacks such as Prime & Probe, Flush & Reload, Evict & Time, as well as Spectre and its variants that capitalize vulnerabilities resulting from speculative execution in modern computer architectures.

To mitigate cache timing attacks, we proposed and evaluated techniques to crate false access times by introducing false hits and false misses. We use a small Guard Cache (a fully associative cache with very similar access latencies as the primary data caches) to cause false hits. We use Guard Cache as both a “Victim Cache” and a “Miss Cache”. We collected performance data using different Guard Cache sizes; 1 KiB to 2 KiB at L1-D and 2 KiB–4 KiB at L2 cache levels. We varied the percentage of the time the Guard Cache is activated as a Miss Cache and as a Victim Cache. We have seen negligible impact on performance, but we have shown that the use of a Guard Cache can prevent several side-channel attacks, particularly those that rely on Flush & Reload or Evict & Reload attacks.

Additionally, we randomly evict data from the primary cache, potentially causing a cache miss when a hit is expected. We have collected performance data by varying the frequency of random evictions. As can be expected, higher eviction frequencies lead to higher performance losses, but potentially greater obfuscation of cache timing. Our techniques incur very minimal hardware (small amounts of Guard Caches) and minimal complexity to vary the frequency of random evictions. Random evictions can prevent side-channels that rely on Prime & Probe attacks.

False hits and false misses can prevent or at least mitigate speculative attacks such as Spectre since Spectre attacks still rely on either Prime & Probe or Flush & Reload techniques. But we also propose an additional technique to mitigate attacks that rely on speculative loads to acquire data illegally. Our technique (labeled SafeLoadOnMiss) delays all speculative load accesses that miss in L1-D cache until the speculation is resolved. Our technique results in 2.1% (geometric mean) performance loss for SPEC 2017 benchmarks when the protection is on for the entire execution of benchmarks. But, the loss is only 0.2% if the protection is on for 10% of execution (assuming that an attack is active only for 10% of the execution time). To mitigate Speculation Interference attacks [19], we use random replacement policies at all cache levels when in Safe Mode. This causes a small amount of additional performance loss (5.3% geometric mean when protection is on only 100% of the time and about 1.2% when the protection is on for only 10% of time).

Techniques proposed in this paper require minimal changes to cache memories and an insignificant increase in hardware. We use very small Guard Caches (1 KiB–2 KiB at L1 or 2 KiB–4 KiB at L2) requiring very minimal additional hardware. As we have shown, even these small Guard Caches can cause sufficient difficulty to side-channel attacks. The
OnMiss requires very small (160 Bytes) additional storage for the hardware needed for random evictions is also minimal. The SafeLoad-F. Mosquera et al.
an attack detection method. The detection method should be efficient for real-time detection so that the system can enter Safe Mode without delays and prevent the attack as quickly as possible. A detection method should also result in highly accurate detection of attacks — false hits can cause higher performance losses. In this paper, we use a simple detection technique based on our observation that most side-channel attacks based on speculative executions use an excessive number of cache flushes. We conservatively enter Safe Mode when a cache flush instruction is executed. After an interval, based on our observations, if no additional flushes are detected, we exit Safe Mode.

We plan to extend the SecurityCloak framework by exploring additional security prevention techniques as well as additional detection techniques.

CRediT authorship contribution statement

Fernando Mosquera: Conceptualization, Data curation, Formal analysis, Investigation, Software, Writing – original draft, Writing – review & editing. Ashen Ekanayake: Conceptualization, Data curation, Formal analysis, Investigation, Software, Writing – original draft, Writing – review & editing. William Hua: Supervision, Validation, Visualization. Krishna Kavi: Conceptualization, Resources, Supervision, Validation, Visualization, Writing – original draft. Gayatri Mehta: Conceptualization, Supervision, Validation, Visualization, Writing – original draft. Liyz John: Conceptualization, Validation, Visualization, Writing – original draft, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

This research was supported in part by National Science Foundation grants #1763848 and #1828105. The authors would also like to thank Tony Chen of UNT Texas Academy of Mathematics and Science (TAMS) program for collecting some of the data presented here and acknowledge the feedback received from Brandon Potter and Mike Ignatowski of AMD.

References


