Guard Cache: Creating False Cache Hits and Misses To Mitigate Side-Channel Attacks

Fernando Mosquera
University of North Texas

Krishna Kavi
University of North Texas

Gayatri Mehta
University of North Texas

Lizy K. John
University of Texas

2023 Silicon Valley Cybersecurity Conference (SVCC) | 979-8-3503-2157-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/SVCC56964.2023.10165527

***Abstract*—Cache side-channel attacks have exposed serious security vulnerabilities in modern architectures. These attacks rely on measuring cache access times to determine if an access to an address is a hit or a miss in the cache. Such information can be used to identify which addresses were accessed by the victim, which in turn can be used to reveal or at least guess the information accessed by the victim. Mitigating the attacks while preserving the performance has been a challenge. The hardware mitigation techniques used in the literature include complex cache indexing mechanisms, partitioning cache memories, and hiding or undoing the effects of speculation. In this paper, we present a *Guard Cache* to obfuscate cache timing, making it more difficult for cache timing attacks to succeed. We create *false cache hits* by using the Guard Cache as a Victim Cache, and *false cache misses* by randomly evicting cache lines. Our obfuscations can be *turned-on* and *turned-off* on demand to protect critical sections or randomly to further obfuscate cache access times. We show that our *false hits* cause very minimal performance penalties ranging between -0.2% to 3.0% performance loss, while false misses can cause higher performance losses. We also show that our approach causes different number of cache hits and misses and different addresses causing misses when compared to traditional caches, demonstrating that common side-channel attacks such as Prime &Probe, Flush &Reload or Evict &Time are likely to misinterpret victims' memory accesses. We use very small Guard Caches (1KiB-2KiB at L1 or 2KiB-4KiB at L2) requiring very minimal additional hardware. The hardware needed for random evictions is also minimal.**

***Index Terms*—Cache Side-Channel attacks, Prime & Probe, Flush & Reload, Evict & Time, Victim Cache**

I. Introduction

Recent hardware attacks have exposed serious security vulnerabilities in modern architectures. These attacks can be successful in revealing encryption keys used in some cryptographic applications, as well as revealing information resulting attacks based on speculative executions such as Spectre [1], [2]. 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 observe the memory addresses accessed by a victim and deduce additional information such as keys used by encryption codes such as AES [3], [4]. Yet another modern hardware side-channel attack is made possible by the use of out-of-order and speculative execution of instructions through modern processor pipelines [1], [2]. Even these attacks rely on observing cache accesses to obtain secret information.

In this paper, we focus on developing techniques to obfuscate only cache side-channels by causing *false hits* and *false misses*. A false hit may appear as if the requested data is a hit in cache, when the attacker is expecting a miss. This is achieved by using a *Guard Cache* as a Victim Cache [5]. A Guard Cache is a small fully associative cache that hosts data evicted (i.e., a victim) from primary cache. On an access to this evicted item, if it is found in the Guard Cache, the data is moved back to the primary cache. Given that the Guard Cache access times are comparable to primary cache access times, the missing data found in the Guard Cache appears as if it was in the primary cache. False misses are created by periodically and randomly evicting data (selected randomly) from cache memories. Both these obfuscation techniques can be used at any cache level (L1-I, L1-D, L2, LLC or even with TLBs). In this paper, we will show that these techniques can prevent or at least mitigate well-known side-channel attacks including Evict &Time [6], Prime &Probe [6], [7], Flush &Reload [8], as well as Spectre and its variants [2], [9]–[11] since even these attacks also depend on cache timing analyses.

The main contribution of our work is the different ways in which cache access times are obfuscated which are itemized below. While there are other randomization techniques proposed to prevent side-channel attacks, they focused on randomization of a single aspect of a system, such as delaying some cache accesses, cache partitioning, life-times associated with cached data or use of interfering threads to create random cache accesses. A long-term observation can potentially reveal the patterns of randomization used by these techniques. We randomize several aspects of caches and the combinations themselves can be randomly changed, making it significantly more difficult to observe any meaningful patterns. The degree of randomization can also be varied to change the level of obfuscation with concomitant impact on performance.

- **Obfuscating Cache Timing With False Hits:** We use a small fully associative Guard Cache to create false hits. Data item evicted from the primary cache[1] is saved in the Guard Cache. If the evicted item (or victim) 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

[1]Guard Caches can be used for any cache including L1-Instruction, L1-Data, L2 and LLC.

comparable. We also rely on *random replacement policy* when entries in the Guard Cache need to be replaced. Additionally, not all data evicted from primary caches or ”victims” are placed in the Guard Cache, thus causing false hits more randomly. Yet another obfuscation that is investigated is to use the Guard Cache more like a ”Miss Cache” [5] - on a cache miss, the missing data is brought into the Guard Cache instead of into the primary cache. We saw negligible performance gains or losses with the Guard Caches. While larger Guard Caches can provide more protection since victims can be held for longer periods of time, they require larger silicon areas and consume more power. We found that 1KiB to 2KiB at L1 level and 2KiB to 4KiB at L2 or LLC Guard Caches are sufficient to prevent several types of side-channel attacks.

- **Obfuscating Cache Timing With False Misses:** We randomly evict data from the primary cache, *but not use the Guard Cache for saving the evicted data*. This causes false misses when an attacker is expecting a hit. The frequency of evictions, as well as what types of data, (e.g., only unmodified data) is evicted can be varied to change cache misses.
- **Safe-mode execution:** The obfuscation using either or both *false hits* and *false misses* can be *turned-on* only when needed by the user to protect critical sections of their programs. It is possible to randomly switch between *safe* and *unsafe* modes of execution to make it even more difficult for an attack to succeed.

In the rest of the paper, we will describe our techniques, demonstrate that they prevent some well-known attacks, and evaluate the impact of our techniques on execution performance as well as complexity of the additional hardware needed.

II. 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 cache performance by eliminating cache *thrashing* in direct mapped caches [12]. In a more recent work [13], victim caches were used to house data evicted by speculative load accesses (for example, ReViCe [13]). We feel this requires complex bookkeeping since one need to distinguish between speculative and non-speculative load accesses, as well as removing 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 can also use the Guard Cache as a *Miss Cache* [5]– 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 makes 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 (1KiB or 2KiB at L1 level and 2KiB to 8KiB 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[2] and evict the selected data based on the eviction frequency but do not place it in the Guard Cache. We varied the frequency of evictions from 5% to 20%. The random evictions lead to performance loss but if the percentage of evictions is kept below 5% (which is sufficient to prevent currently known side channel attacks), the loss is small. Additionally, as we will show in Section III, the evictions can be randomly *turned on* and *turned off* to both increase obfuscation and reduce performance penalties. The false misses will make attacks using such techniques as Evict &Time [6], Prime &Probe [6], [7], Flush &Reload [8] more difficult since the attacker will see many more misses than those caused by victim accesses.

Figure 1 shows the working of the Guard Cache in the memory hierarchy. The arrow labeled ”1” shows the case when the Guard Cache is not used - data evicted from the primary cache is not stored in the Guard Cache. Arrow labeled ”2” indicates when a data item is evicted from a Primary Cache (L1, L2 or LLC) and stored in the Guard cache (used as a victim cache). Arrow labeled ”3” indicates the case when the missing data is brought into the Guard cache (used as miss cache) and not into the Primary Cache. Arrow labeled ”4” shows the case when false misses are activated. As described above, data from primary cache is evicted randomly.

We can deploy both false hits and false misses together to increase the randomization of cache timing. Figure 2 shows a simulated Prime & Probe attack. The left column shows the

[2] In our experiments, we only evict unmodified data to avoid the need for write-back along the memory hierarchy. However, one can decide on which types of cache lines to evict e.g., most frequently used vs least frequently used, creating different levels of obfuscation.

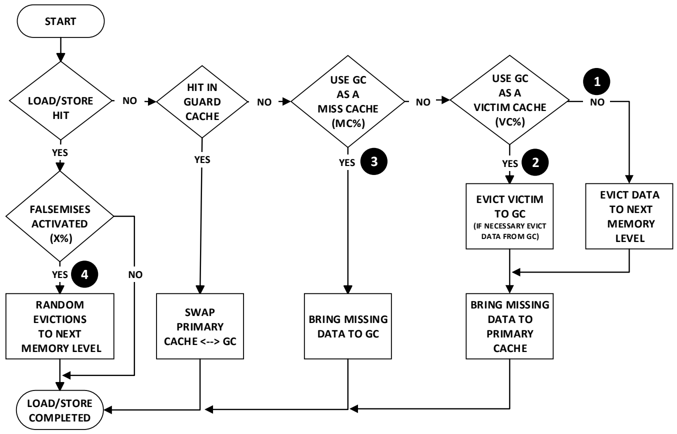


Fig. 1. Flowchart of the Guard Cache

normal mode indicating what attacker sees as cache misses caused by victim's code (evicting attackers primed data). The middle column shows that the attacker sees additional cache misses caused by false misses strategy (shown in red). The right column indicates the case when both false hits (using the Guard Cache) and false misses are turned on. Now some misses caused by victim's access, seen in the left column, are missing (shown in the shaded yellow area) due to false hits.

The use of the Guard Cache causing *false hits* may also prevent attacks such as Spectre. The left hand side of Figure 3 shows a successful attack using a proof-of-concept code from [14]: 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 are not 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, reducing the time for bounds check, and limiting the accesses to out of bounds data.

For attack models based on cache timing analyses to differentiate between cache hits and misses, 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 maybe 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 Figure 3, even a 1KiB Guard Cache at L1-D level obscures data during Spectre attack (and prevents the attack), and while 3% random evictions may not completely prevent this attack, 5% or higher rates of evictions prevent the attack. While these numbers are based on the available proof of attack codes, the sizes of Guard Caches and the frequencies of evictions can be varied to achieve desired levels of protection.

III. Results and Analysis

We evaluated our design using Gem5 [15] System-call Emulation (SE) mode to accurately model a single high performance X86 CPU core. The configuration uses 64KiB L1-D (8-Way), 32KiB L1-I (4-way) and 2MiB L2 (16-way) caches. We executed several SPEC CPU2017 benchmarks in system call emulation mode, fast forwarding for 1 billion instructions, then collecting performance data for 500 million instructions. We evaluated the benchmarks in *baseline* (no false hits or misses), *only false hits* with different Guard Cache sizes, different frequencies for using Guard Cache as a victim cache or as a miss cache, *only false misses* with different rates of random evictions, and with *both false hits and misses*. We studied the use of *false hits* (using Guard cache) and *false misses* at both L1-D and L2 levels.

A. Analysis of False Hits:

In this section, we present the performance impacts caused by our Guard Cache for several different SPEC 2017 benchmarks. We varied the Guard Cache sizes (1KiB-2KiB at L1-D level and 2KiB-4KiB at L2 level). We have already shown (Figure 3) that these sizes are more than sufficient to prevent currently known side-channel attacks, and these sizes for Guard Caches require very small additional hardware. We varied the fraction of the time a data item that is evicted from the primary cache (L1-D or L2) is moved to the Guard Cache: the first number for each result in Figure 4 shows this percentage. We varied how often the Guard Cache is used as a Miss Cache, that is, on a demand miss, the missing data is brought in to the Guard Cache and no data is evicted from the primary cache: the second number for each result in Figure 4 shows this percentage. Thus, 90-10 shows the results when 90% of all evictions from the primary cache are moved to the Guard Cache (used as victim cache), and 10% of demand misses are brought into Guard Cache (used as miss cache). As can be seen, the results in Figure 4 show very minimal impact on performance ranging between -0.2% to 3.0% performance loss. Negative bars indicate performance gains – LRU 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.

Figure 5 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 cache misses that are satisfied by the Guard Cache per 1000 instructions executed. Guard Cache is likely to result in performance benefits when application exhibits higher cache conflicts; this can be seen from higher percentage of *false hits*. Consider *cactus* with 2.92 L1-D misses per 1000 instructions without

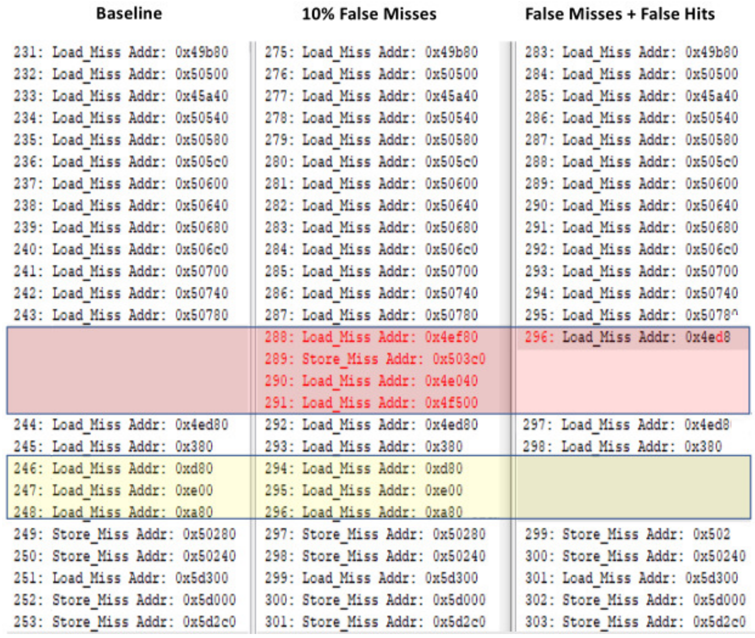
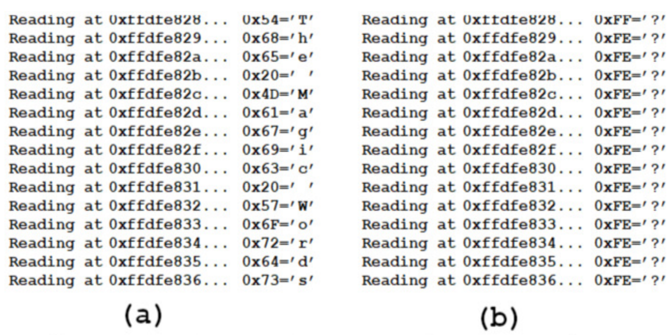


Fig. 2. Simulated Prime & Probe Attack: False Hits and False Misses Obfuscate Cache Misses and Timing



Protection Method	Freq (%)	Size Guard Cache	Attack prevented
False Misses L1	10%	-	✓
	5%	-	✗
False Hits GC L1	-	1 KB	✓
	-	2 KB	✓

Fig. 3. Spectre Attack (a) Baseline Mode (b) With Guard Cache

a Guard Cache and a Guard Cache of even 1KiB effectively eliminates these cache misses (shown as false hits).

This also indicates that most side-channel attacks such as Prime&Probe, Evict&Time or Flush&Reload that rely on observing which accesses caused misses, will fail because most of such cache misses become invisible with the use of a Guard Cache. Figure 2 in Section II demonstrated that Guard Cache makes many of the L1-D evictions caused by Prime&Probe attack invisible. On the other hand, *lbm* has very high L1-D miss rates (21.36 misses per 1000 instructions at L1-D), but these misses are not satisfied by Guard Cache. Such a behavior may potentially indicate that the application is a streaming application. Figure 5 shows false hits data for different Guard Cache sizes. It should be noted that most applications see very insignificant performance impact due to Guard Caches that are larger than 4KiB or 8KiB. Higher number of data memory accesses places higher demand on Guard Cache and large Guard Caches will be more beneficial for such applications. The application *roms* appears to benefit from larger Guard Cache (more *false hits* with larger Guard Cache). This behavior may indicate capacity misses since the application shows high MPKI (11.6 misses per 1000 instructions), but 1KiB Guard Cache shows very minimal benefit, but our goal is not improved performance but hiding some cache misses.

Figure 5 also includes additional cache hits due to Guard Caches at L2 level. The L2 cache misses that are found in L2 level Guard Cache is very small. This is expected since there are fewer memory accesses and misses at L2 level. Moreover it should be noted that we use Random Replacement policy with our Guard Caches. This means that a data item evicted from the primary cache and moved to the Guard Cache may be evicted later when another data item evicted from the primary cache needs space in the Guard Cache and Random Replacement policy may cause more recently evicted item to be replaced in the Guard Cache.

It should be noted that the data in Figure 4 and Figure 5 are collected with no side channel attack. However, when an attack such as Prime & Probe, Flush & Reload or Evict & Time is underway, the GC and random evictions will have higher impact on performance. These attacks result in higher levels of cache misses, many of which are caught by the GC, and the random evictions present unexpected misses to the attacker. For example, for simulating a proof of concept attack representing Prime & Probe as well as Specture attack (the same attacks that we used to produce Figure 2 and Figure 3), 1 KiB Guard Cache at L1-D resulted in more than 100% additional cache hits and 5% random evictions caused 200%

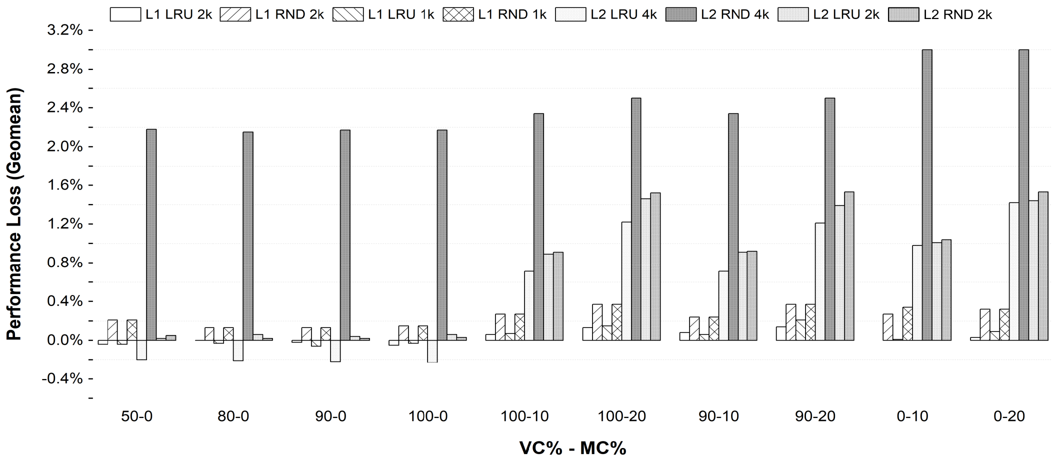


Fig. 4. Guard Cache used as a Victim Cache and Miss Cache. X-axis designates the fraction of the evicted lines that are moved to Guard Cache (VC%) and the fraction of demand misses that are brought to Guard Cache (MC%)

	Baseline				L1 Misses that are found in GC1		L2 Misses that are found in GC 2	
Benchmark	L1-D accesses per 1000 instructions	L1-D Misses per 1000 instructions	L2 accesses per Million instructions	L2 Misses per Million instructions	1KiB GC1	2KiB GC1	2KiB GC2	4KiB GC2
bwaves_s	286.80	3.95	3955.43	3952.57	0	0	0	0
cactuBSSN_s	474.63	2.92	96860.32	288.84	1362096	1378503	0	0
deepsjeng_s	402.52	0.28	6853.09	115.76	4877	7752	4	4
exchange2_s	158.94	0.00	7.28	2.57	0	0	0	0
fotonik3d_s	395.64	0.05	3146.18	44.52	0	0	0	0
imagick_s	425.87	7.38	7392.09	294.19	151	322	0	0
lbm_s	333.96	21.36	21445.10	13027.49	34	56	1	2
leela_s	361.94	0.15	979.39	8.79	6104	11145	0	0
mcf_s	459.63	15.51	15393.62	5692.01	52734	96158	767	1472
roms_s	335.88	11.60	11610.84	6499.45	646139	2221141	6	7
wrf_s	331.86	2.97	3417.83	28.36	5140	9893	0	0
x264_s	362.58	0.06	1174.48	50.09	28	83	0	0
xz_s	383.38	0.25	349.82	176.36	1123	2594	6	10
Geomean	351.78	0.71	3090.07	228.87				

Fig. 5. Additional cache hits due to Guard Cache

additional misses.

B. Analysis of False Misses

In the next set of experiments, we evaluate the use of *random evictions* to create false misses. Figure 6 shows the performance loss for SPEC 2017 benchmarks when cache lines are randomly evicted. On every cache access (either at L1-D or L2) that is a hit, we decide if a random cache line should be evicted based on a selected frequency. The data is for different frequencies of random evictions (or *false misses*). Higher frequency of evictions will cause higher performance losses. For example, if a cache line is evicted at 20% of the time a L-1D cache access is a hit, we see a geometric mean performance loss of 170% for SPEC 2017 benchmarks. This is unacceptable performance loss; however, it can cause significant obfuscation of cache access times. We anticipate that 5% frequency of random evictions is adequate to cause sufficient obfuscation which causes a geometric mean performance loss of 23%. 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.

Application behavior again causes different amounts of false misses. Figure 7 shows additional cache misses per 1000 instructions encountered by applications because of using different rates of random evictions. The figure also includes L1-D cache accesses per 1000 instructions and cache misses for 1000 instructions in the baseline. Since we apply random evictions on every (L1-D) cache access that is a hit, higher cache accesses can lead to more frequent random evictions. However, applications that have higher miss rates will likely see less impact due to random evictions (or fewer additional cache misses encountered by the applications). Streaming applications, on the other hand, may not see the effects of false misses since the randomly evicted data may not be accessed. Consider for example, lbm and wrf_s, both have about the same number of L1-D accesses per 1000 instructions, but lbm has higher miss rate (MPKI of 21.36 compared to 2.97 for wrf_s); this leads to more random evictions and additional cache misses for wrf_s than those for lbm. On the other hand, mcf has higher L1-D accesses (and higher miss rates) explaining 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 higher number of random evictions. It should be noted that higher false misses can aid in further mitigating side-channel attacks (Figure 2 shows false misses caused significantly more misses than those caused for Prime&Probe attack).

To simulate *turning on* protection only when needed (for example, to protect critical sections) we experimented by *turning-on* false misses 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 for 50 million instructions (out of 500 million instructions simulated in our experiments). Figure 8 shows the geometric mean performance losses for

Benchmark	Perfomance Loss(%)					
	L1D Freq 5%	L1D Freq 10%	L1D Freq 20%	L2 Freq 5%	L2 Freq 10%	L2 Freq 20%
bwaves_s	44%	98%	233%	0%	0%	0%
cactuBSSN_s	2%	14%	133%	29%	57%	118%
deepsjeng_s	14%	42%	118%	2%	5%	12%
exchange2_s	6%	16%	79%	0%	0%	0%
fotonik3d_s	8%	24%	102%	2%	5%	12%
imagick_s	55%	184%	530%	1%	6%	15%
lbm_s	-2%	31%	152%	1%	2%	3%
leela_s	27%	69%	149%	1%	1%	3%
mcf_s	7%	38%	114%	1%	1%	3%
roms_s	62%	117%	202%	0%	0%	0%
wrf_s	47%	122%	318%	5%	8%	14%
x264_s	35%	83%	175%	0%	1%	2%
xz_s	17%	54%	158%	0%	0%	0%
Geomean	**23%**	**62%**	**173%**	**3%**	**6%**	**11%**

Fig. 6. Average performance loss using random evictions (FalseMiss Scheme) at different eviction frequencies in L1D and L2 caches

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 2% at 5% random eviction rate at L1-D level (not 23% if the random eviction takes place during entire execution times as shown in Figure 6). Even when *false misses* are introduced for half of the application execution, the geometric mean performance loss is 9% at 5% random eviction rate. We feel that security protection should be used only when needed - to protect critical segments of applications which minimizes performance losses.

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.

Benchmark	Baseline		Additional L1 MPKI		
	L1-D Accesses per 1000 instructions	L1-D Misses per 1000 instructions	L1D Freq 5%	L1D Freq 10%	L1D Freq 20%
bwaves_s	286.80	3.95	15.05	43.85	127.00
cactuBSSN_s	474.63	2.92	22.40	50.36	184.13
deepsjeng_s	402.52	0.28	20.04	42.02	92.87
exchange2_s	158.94	0.00	8.10	16.53	37.98
fotonik3d_s	395.64	0.05	19.65	39.73	80.30
imagick_s	425.87	7.38	21.18	61.04	192.48
lbm_s	333.96	21.36	0.99	25.27	90.56
leela_s	361.94	0.15	29.81	38.02	95.27
mcf_s	459.63	15.51	14.89	41.81	200.91
roms_s	335.88	11.60	24.17	42.05	104.91
wrf_s	331.86	2.97	16.31	43.48	123.58
x264_s	362.58	0.06	19.41	42.14	89.32
xz_s	383.38	0.25	19.09	40.77	93.45
Geomean	**351.78**	**0.71**	**14.68**	**38.92**	**106.94**

Fig. 7. Additional L1 Misses per Kilo Instructions for different frequencies of random evictions

Protection activation time (%)	Perfomance Loss(%)		
	L1D Freq 5%	L1D Freq 10%	L1D Freq 20%
10%	2%	5%	16%
50%	9%	26%	81%
100%	23%	62%	173%

Fig. 8. Performance Loss when random evictions at L1-D are activated only for a portion of application execution times

C. 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 Figure 6.

D. Discussion

Our research should be compared with techniques that focus on mitigating side-channel attacks. As will be described in Section IV, known techniques reported average losses ranging between 1% and 15% for various SPEC benchmarks (SPEC2000, SPEC2006 and SPEC2017). Many of these techniques require changes to cache addressing, ability to lock cache sets for different processes, or encrypting addresses. Our techniques proposed in this paper (i.e., false hits and false misses) require minimal changes to cache memories and insignificant increase in hardware. As we have shown, even a small Guard Cache can cause sufficient difficulty to side-channel attacks. Additional techniques such as randomly not storing data evicted from primary caches (L1-D or L2) in Guard Cache or using Guard Caches as Miss Caches can make it difficult for the attacker to determine the presence of a Guard Cache or the size of such a resource. Likewise, we have already demonstrated the use of *false misses* with random evictions can be used only during a portion of execution. Additional techniques such as randomly *turning on* random evictions and *turning off* or randomly increasing the duration of *false misses* ian be explored.

We use very small Guard Caches (1KiB-2KiB at L1 or 2KiB-4KiB at L2) requiring very minimal additional hardware. The hardware needed for random evictions is also minimal. Our techniques can be used along with other approaches for mitigating cache side-channel attacks such as those described in Section IV. However, the combination of techniques may cause higher performance losses while possibly providing higher levels of protection against security attacks.

IV. Related Research

We will summarize some key works on cache side-channel attacks that are most closely related to our research.

1) Cache Side-Channel Attacks: Most common side-channel attacks were reported in [6], [7], [16], [17].

One approach to prevent such attacks is to disallow sharing of cache memories. Dynamically Allocated Way Guard (DAWG) [18] is a mechanism to secure way partitioning of set associative caches. Depending on how much of the cache is set aside for partitioning and if the partitioning is applied at L2 or L3 cache, the average performance loss for SPEC2006 benchmarks ranges between 7% to 15%. In Partition Locked Cache (PLcache) [19], each cache line is augmented with a process ID field and a lock bit L (to prevent other processes from evicting data). PLCache reported an average of 2% performance loss for SPEC2000 benchmarks. NewCache [20] replaces the fixed address decoder of a direct mapped cache with a dynamic (inverse) line-number mapper, which can be implemented by content addressable memory (CAM). The researchers report an average loss of 1% for SPEC2000 benchmarks. Ceaser [21] is another architecture based on randomized mappings, which employs a Low-Latency Block-Cipher (LLBC) to translate the physical line-address into an encrypted line-address, and access the cache with this encrypted line-address. In addition, Ceaser periodically changes the encryption key and performs dynamic-remapping to improve robustness. Since the technique is applied only at LLC, the researchers report an average performance penalty of 1% for SPEC2006 benchmarks. ScatterCache [22] is also based on randomized mappings by associating each address with a set of up to n-ways. In [23], the authors modify LRU replacement for shared caches (L2 or LLC) to prevent the eviction of victim data by an attacker. However, this requires additional bits with cache lines to track which core contains a copy of the data. It should be noted that victim's data may still be evicted if no other possible cache line can be found for eviction. But it may be possible to augment this technique with a Guard Cache to capture victim's data if it is evicted. The authors also restrict the use of CLFULSH from user space. In a related work [24], the authors delay bringing shared data into higher level caches (e.g., L1) on the first access, making the access appear as a miss. Our approach is more general and combines several different randomization techniques to increase the level of obfuscation.

2) Speculative and Cache Side-Channel Attacks: Some side channel attacks are based on speculative execution [4], [7], [18], [25], [26]. Since our focus is on cache timing attacks, we will not include discussion of techniques specifically designed for preventing or mitigating such attacks (see for example, [14], [27], [28]).

3) Other Randomization Techniques: Random Fill Cache Architecture [29] replaces demand fetch with random cache fill within a configurable neighborhood window: the missing data is sometimes provided directly to the processor without bringing to cache. This may help in obfuscating cache timing, there were no reported studies on the effectiveness of this approach or potential performance impacts. Covert-Enigma [30] is a random perturbation-based defense technique that introduces random timing delays to memory accesses. Ghost Thread [31] is a defense mechanism against side channel attacks through a flexible library that injects random cache accesses in the same address region than the protected process. It uses additional threads to cause these random accesses, which can be invoked through library calls. ClepsydraCache [32] assigns each cache entry a random time-to-live to reduce conflicts on cache addresses. The idea is obfuscating conflict-based evictions with time-based evictions. This solution is applicable to LLC with a minimal performance overhead.

V. Conclusions and Future Work

Cache side-channel attacks use access latencies to determine if an access is a cache hit or a miss. Attackers may deliberately evict specific cache lines and observe to see if the victim accesses that data (causing a miss on the access). A cache miss causes longer latency and the attacker can observe the delays using available performance counters.

We proposed and evaluated techniques to obfuscate the timing 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". These different techniques of obfuscating cache timing can be combined randomly to further increase noise in the cache timing. We collected performance data using different Guard Cache sizes; 1KiB to 2KiB at L1-D and 2KiB-4KiB 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. Additionally, we randomly evict data from 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. We believe that the mitigation techniques should be amenable to being deployed only when needed. The protection should be *turned on* only when executing critical code segments, and *turned off* otherwise. It may also be possible to *turn-on* protection automatically when an attack is detected or suspected. Numerous approaches for detecting different types of hardware attacks have been described in the literature, see for example [33]–[35]. Any of these or other techniques can be used to detect and enable *safe mode* operation. We have shown that performance loss due to *false misses* can be minimal if random evictions are *turned on* only a for short duration. It may then be possible to gradually increase the frequency of evictions to increase the level of obfuscation, trading off performance with higher levels of attack mitigation. Both Guard Caches and random evictions have significantly smaller impact on performance at L2 level since there are significantly fewer accesses to L2 cache.

REFERENCES

[1] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *arXiv preprint arXiv:1801.01207*, 2018.

[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 *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1–19.

[3] D. J. Bernstein, "Cache-timing attacks on aes," 2005.

[4] N. Lawson, "Side-channel attacks on cryptographic software," *IEEE Security & Privacy*, vol. 7, no. 6, pp. 65–68, 2009.

[5] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ser. ISCA '90. New York, NY, USA: Association for Computing Machinery, 1990, p. 364–373. [Online]. Available: https://doi.org/10.1145/325164.325162

[6] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of aes," in *Cryptographers' track at the RSA conference*. Springer, 2006, pp. 1–20.

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

[8] Y. Yarom and K. Falkner, "Flush+ reload: A high resolution, low noise, l3 cache side-channel attack," in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 2014, pp. 719–732.

[9] N. Abu-Ghazaleh, D. Ponomarev, and D. Evtyushkin, "How the spectre and meltdown hacks really worked," *IEEE Spectrum*, vol. 56, no. 3, pp. 42–49, 2019.

[10] M. Löw, "Overview of meltdown and spectre patches and their impacts," *Advanced Microkernel Operating Systems*, p. 53, 2018.

[11] Z. He, G. Hu, and R. Lee, "New models for understanding and reasoning about speculative execution attacks," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 40–53.

[12] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," *ACM SIGARCH Computer Architecture News*, vol. 18, no. 2SI, pp. 364–373, 1990.

[13] S. Kim, F. Mahmud, J. Huang, P. Majumder, N. Christou, A. Muzahid, C.-C. Tsai, and E. J. Kim, "Revice: Reusing victim cache to prevent speculative cache leakage," in *2020 IEEE Secure Development (SecDev)*. IEEE, 2020, pp. 96–107.

[14] G. Saileshwar and M. K. Qureshi, "Cleanupspec: An" undo" approach to safe speculation," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 73–86.

[15] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 1–7, aug 2011. [Online]. Available: https://doi.org/10.1145/2024716.2024718

[16] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, 2015, pp. 897–912.

[17] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+ flush: a fast and stealthy cache attack," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2016, pp. 279–299.

[18] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "Dawg: A defense against cache timing attacks in speculative execution processors," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 974–987.

[19] J. Kong, O. Aciicmez, J.-P. Seifert, and H. Zhou, "Deconstructing new cache designs for thwarting software cache-based side channel attacks," in *Proceedings of the 2nd ACM workshop on Computer security architectures*, 2008, pp. 25–34.

[20] F. Liu, H. Wu, K. Mai, and R. B. Lee, "Newcache: Secure cache architecture thwarting cache side-channel attacks," *IEEE Micro*, vol. 36, no. 5, pp. 8–16, 2016.

[21] M. K. Qureshi, "Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 775–787.

[22] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard, "Scattercache: Thwarting cache attacks via cache set randomization," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 675–692.

[23] M. Yan, B. Gopireddy, T. Shull, and J. Torrellas, "Secure hierarchy-aware cache replacement policy (sharp) defending against cache-based side channel atacks," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 347–360, 2017.

[24] K. Ramkrishnan, S. McCamant, P. C. Yew, and A. Zhai, "First time miss: Low overhead mitigation for shared memory cache side channels," in *Proceedings of the 49th International Conference on Parallel Processing*, 2020, pp. 1–11.

[25] Z. He and R. B. Lee, "How secure is your cache against side-channel attacks?" in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017, pp. 341–353.

[26] R. Spreitzer, V. Moonsamy, T. Korak, and S. Mangard, "Systematic classification of side-channel attacks: A case study for mobile devices," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 1, pp. 465–488, 2017.

[27] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. W. Fletcher, and J. Torrellas, "Invisispec: Making speculative execution invisible in the cache hierarchy (corrigendum)," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 1076–1076.

[28] C. Sakalis, S. Kaxiras, A. Ros, A. Jimborean, and M. Själander, "Efficient invisible speculative execution through selective delay and value prediction," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2019, pp. 723–735.

[29] F. Liu and R. B. Lee, "Random fill cache architecture," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2014, pp. 203–215.

[30] A. Dhavlle, S. Rafatirad, K. Khasawneh, H. Homayoun, and S. M. P. Dinakarrao, "Imitating functional operations for mitigating side-channel leakage," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.

[31] R. Brotzman, D. Zhang, M. Kandemir, and G. Tan, "Ghost thread: Effective user-space cache side channel protection," in *Proceedings of the Eleventh ACM Conference on Data and Application Security and Privacy*, 2021, pp. 233–244.

[32] J. P. Thoma, C. Niesler, D. Funke, G. Leander, P. Mayr, N. Pohl, L. Davi, and T. Güneysu, "Clepsydracache–preventing cache attacks with time-based evictions," *arXiv preprint arXiv:2104.11469*, 2021.

[33] C. Pierce, M. Spisak, and K. Fitch, "Capturing 0day exploits with perfectly placed hardware traps," in *proc. BlackHat Conf*, vol. 7, 2016.

[34] H. Wang, H. Sayadi, S. Rafatirad, A. Sasan, and H. Homayoun, "Scarf: Detecting side-channel attacks at real-time using low-level hardware features," in *2020 IEEE 26th International Symposium on On-Line Testing and Robust System Design (IOLTS)*. IEEE, 2020, pp. 1–6.

[35] C. Li and J.-L. Gaudiot, "Online detection of spectre attacks using microarchitectural traces from performance counters," in *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 2018, pp. 25–28.