Guard Cache: Creating Noisy Side-Channels

Fernando Mosquera , *Student Member, IEEE*, Krishna Kavi , *Senior Member, IEEE*,
Gayatri Mehta , *Senior Member, IEEE*, and Lizy John , *Fellow, IEEE*

Abstract—**Microarchitectural innovations such as deep cache hierarchies, out-of-order execution, branch prediction and speculative execution have made possible the design of processors that meet ever-increasing demands for performance. However, these innovations have inadvertently introduced vulnerabilities, which are exploited by side-channel attacks and attacks relying on speculative executions. Mitigating the attacks while preserving the performance has been a challenge. In this letter we present an approach to obfuscate cache timing, making it more difficult for side-channel attacks to succeed. We create *false cache hits* using a small *Guard Cache* with randomization, and *false cache misses* by randomly evicting cache lines. We show that our *false hits* and *false misses* cause very minimal performance penalties and our obfuscation can make it difficult for common side-channel attacks such as Prime &Probe, Flush &Reload or Evict &Time to succeed.**

Index Terms—**Cache side-Channel attacks, evict &time, flush &reload, guard cache, miss cache, obfuscating cache access timing, prime &probe, victim cache.**

I. INTRODUCTION

MICROARCHITECTURAL innovations such as deep cache hierarchies, out-of-order execution, branch prediction and speculative execution have made possible the design of processors that meet ever-increasing demands for performance. However, these innovations have inadvertently introduced vulnerabilities, which are exploited by side-channel attacks and attacks relying on speculative executions. 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 [1], [2]. While there have been numerous vulnerabilities caused by out-of-order and speculative execution, we will not address them in this work and only focus on cache side-channel attacks such as Evict &Time [3], Prime &Probe [3], [4], Flush &Reload [5].

We describe techniques to obfuscate cache side-channels by causing *false hits* and *false misses*. A *false hit* may appear as if

the requested data is a hit in the primary (L1-D or L2) cache, when the attacker is expecting a miss. This is achieved by using a *Guard Cache*, which has similarities to *Victim Caches* and *Miss Caches* [6]. Given that Guard Cache access times are comparable to primary (L1-D or L2)) caches, the missing data found in the Guard Cache appears as if it was actually in the primary cache. Our *Guard Cache* behaves both as Victim and Miss Caches, and the frequency with which it is used as a Victim or Miss cache can be randomly varied. *False misses* are created by randomly evicting data from primary cache memories.

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 cache addressing, 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.

- Not every data item evicted from the primary cache into the Guard Cache. The probability with which an evicted item is stored in the Guard cache can be varied, making it difficult for the attacker to discover the existence or the size of the the Guard Cache.
- Missing data is not always brought into the primary cache, but stored in the Guard and no data is evicted from the primary cache. The probability of using the Guard cache as a victim cache or miss cache can be varied.
- The probability with which data in the primary caches are evicted, causing *false misses*, can be varied. While higher probability of evictions may offer greater obfuscation, it may also cause higher performance losses. It may be possible to incrementally raise the eviction probabilities when an attack is detected.

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

Fig. 1.　Flowchart of the Guard Cache.

Fig. 2.　Spectre Attack (a) Baseline Mode (b) With Guard Cache.
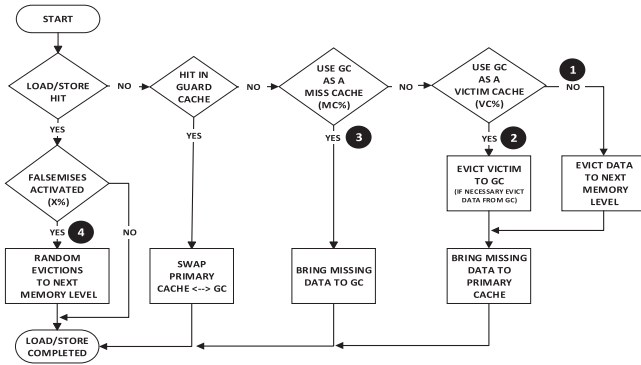
can be used by an attacker to obtain information regarding which memory addresses a victim accessed, and possibly retrieve data from those addresses. We use Guard Caches to create false hits. Unlike other works that use victim caches either always, or only to load speculative accesses as in ReViCe [7], we use Guard Caches in different ways to create noisy side channels and make it very difficult for the attacker to discover the existence of the Guard Cache. Guard Caches can be used throughout the memory hierarchy (L1-I, L1-D, L2 and LLC).

Fig. 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. Arrow labeled "2" indicates when a data item evicted from a Primary Cache (L1, L2 or LLC) is 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. We rely on *random replacement policy* when entries in the Guard Cache need to be replaced.

We also create *false misses* by randomly evicting cache lines. On every cache access we select a cache line randomly and evict that cache line from the Primary Cache (but do not place it in the Guard Cache): this is indicated by the arrow labeled "4" in Fig. 1. The false misses will make attacks such as Evict &Time [3], Prime &Probe [3], [4], Flush &Reload [5] more difficult since the attacker will see many more misses than those caused by victim accesses.

The use of the Guard Cache causing *false hits* and random eviction causing *false misses* can be used to obfuscate side channel attacks such as Prime & Probe, Flush & Reload or Evict & Time. Since even the attacks that exploit speculative execution rely on cache timing (primarily using Flush & Reload), our technique can also prevent attacks such as Spectre. The left hand side of Fig. 2 shows a successful attack using a proof-of-concept code from [8]: 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. Guard Cache saves the flushed array bounds variable making the bounds check very fast, thus minimizing any data that is speculatively loaded into caches. The table at the bottom of Fig. 2 shows that even a 1KiB Guard Cache at L1-D level prevents Spectre attack. While 5% random evictions may not completely prevent this attack, 10% or higher rates of evictions prevent the attack. Random evictions are better suited for mitigating attacks that look for misses such as Prime and Probe, since we create additional random misses, than those that look for cache hits such as Spectre.

Since most cache side channel attacks are based on observing cache hits or misses to *specific cache lines*, we feel that our false hits and false misses obfuscate cache timing and prevent or at least make it very difficult for most timing-based attacks to succeed.

To make it more difficult for the attacker to discover the presence of, or the size of Guard Cache, only a fraction of all data evicted from primary cache is stored in Guard Cache. The use of Guard Cache as a miss cache and random replacement for evicting data from Guard Cache also makes it difficult to discover the size of the Guard Cache.

III. RESULTS AND ANALYSIS

We evaluated our design using Gem5 [9] 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 frequency for using the Guard Cache as a Victim Cache or as a Miss Cache, *false misses* with different rates of random evictions, and with *both false hits and misses*. We studied the use of *false hits* (using Guard caches) and *false misses* at both L1-D and L2 levels.

Analysis of False Hits: In this section we evaluate the performance losses due to the use of our Guard Caches for several different SPEC 2017 benchmarks. We varied the Guard Cache
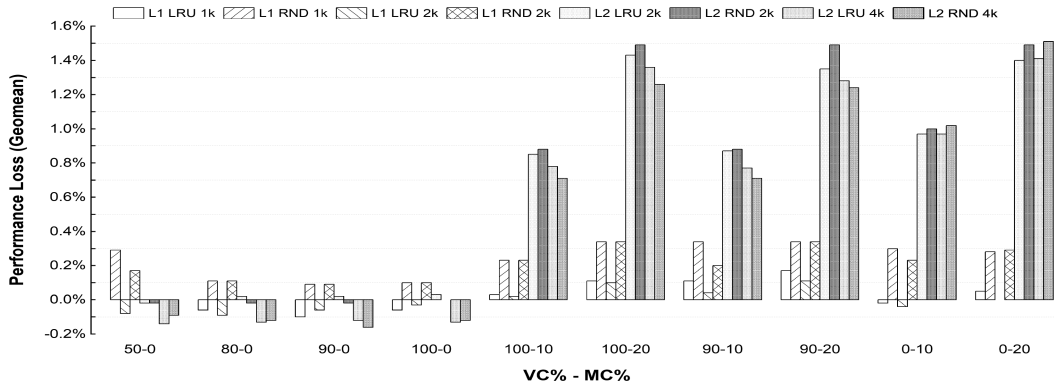
Fig. 3.   Guard Cache used as a victim cache and miss cache. X-axis designates the fraction of the evicted lines moved to Guard Cache (VC%) and the fraction of demand misses brought to Guard Cache (MC%).

Benchmark	Performance 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. 4.   Average performance loss due to random evictions with different eviction frequencies.

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

Fig. 5.   Performance Loss when random evictions are turned on only for a fraction of execution.

sizes, 1 KB - 2 KB at L1-D level and 2KB-4 KB at L2 level. We varied the fraction of data items evicted from the primary cache (L1-D or L2) and moved to the Guard Cache: the first number for each result in Fig. 3 shows this percentage. We also 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 Fig. 3 shows this percentage. Thus, 90-10 shows the results when 90% of all evictions from the primary cache are moved to the Guard cache, and 10% of demand misses are brought into Guard cache. As can be seen, the results in Fig. 3 show very minimal impact on performance (ranging between −0.2% to 1.5% performance loss - negative numbers indicate performance gains). LRU replacement policy for Guard Cache used as a victim cache results in performance gains while Random Replacement results in losses. The use of the Guard Cache as a Miss Cache results in slightly higher performance losses than when used as a Victim Cache. Guard Caches at L2, with Miss Cache mode results in higher performance losses.

Analysis of False Misses: Next, we analyze the performance impact of random evictions causing *false misses*. Fig. 4 shows the performance losses for different SPEC 2017 benchmarks for random eviction frequencies of 5, 10, 15%. On every cache access that is a hit (either at L1-D or L2), we decide if a random cache line should be evicted based on a selected frequency, and

evict a randomly selected cache line. Fig. 4 includes data for both L1-D and L2 caches. As expected, higher rates of random evictions cause higher performance losses, but may provide greater obfuscation against side-channel attacks. Fig. 4 shows that for some benchmarks (*bwaves, imagick, roms, wrf*), the performance loss is more than 40% when the *random eviction* rate is 5%, while the performance loss for other benchmarks is substantially smaller. Application memory access behavior causes different amounts of false misses, and different amounts of performance loss. An application that exhibits higher cache miss rates (for example, *lbm, mcf*) may not see significant impact due to additional misses caused by random evictions, while applications that exhibit very low cache miss rates (for example *x264, xz*) may see higher impact on performance impact due to random evictions. Also, since random evictions occur on cache accesses that are hits, higher number of accesses to cache and higher hits may also cause more evictions due to *false misses*. A detailed application characteristics of SPEC 2017 can be found in [10]. Additionally, an application only sees performance loss if the randomly evicted data is accessed. Streaming applications may not see the effects of false misses since the randomly evicted data may not be accessed.

We also 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). Fig. 5 shows the results. This data is to show that if side-channel mitigation is turned on only when needed (either when an attack is detected or when executing critical

code segments, such as critical kernel codes), the performance loss may be acceptable.

Combined Analysis: In the final set of experiments, we used both the 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 the Guard Cache is negligible. The results are very similar to those shown in Fig. 4.

IV. RELATED RESEARCH

Based on the access latency encountered by memory accesses (i.e., Load and Store instructions), an attacker can deduce whether or not an access was a hit or a miss in cache. The attacker either looking for misses to his/her data, indicating that victims access evicted attacker data, or the attacker evicts selected cache lines to see if the victim accessed the evicted data [3], [4], [11], [12]. Mitigation techniques either disallow sharing of cache sets [13], [14], changing how addresses are mapped to cache sets [15], [16], [17]. Other techniques include Random Fill Cache Architecture [18] that replaces demand fetch with random cache fill within a configurable neighborhood window and Ghost Thread [19] that uses additional threads that injects random cache accesses in the same address region than the protected process. ClepsydraCache [20] assigns each cache entry a random time-to-live to reduce conflicts on cache addresses. Our approach requires minimal changes to cache designs or changes to the microarchitecture of processors.

While past approaches have similarity to components of the proposed mechanism, we combine hardware structures and randomization policies in a manner that brings additional robustness by making it significantly more difficult to observe any meaningful timing patterns. 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 ability to change the degree of randomization is also a useful feature of our scheme.

V. CONCLUSION AND FUTURE WORK

Our focus in this contribution is the mitigation of timing based side-channel attacks such as Prime&Probe, Flush&Reload and Evict&Time. However, since even speculative execution attacks (such as Spectre and its variants) rely on cache timings our techniques should be useful against such attacks.

We proposed and evaluated techniques to obfuscate the timing by introducing *false hits* and *false misses*. We use a small Guard Cache as both a "Victim Cache" and a "Miss Cache". We collected performance data using different Guard Cache sizes; 1 K to 2 K at L1-D and 2K-4 K 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 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. *We believe that the obfuscations should be triggered only when needed, either to protect critical sections or when an attack is suspected or detected. And the run-time system should be provided with a range of options to prevent or at least make it very difficult for an attack to succeed.*

REFERENCES

[1] D. J. Bernstein, "Cache-timing attacks on AES," 2005. [Online]. Available: https://mimoza.marmara.edu.tr/~msakalli/cse466_09/cache%20timing-20050414.pdf

[2] N. Lawson, "Side-channel attacks on cryptographic software," *IEEE Secur. Privacy*, vol. 7, no. 6, pp. 65–68, Nov./Dec. 2009.

[3] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of AES," in *Proc. Cryptographers' Track RSA Conf.*, Springer, 2006, pp. 1–20.

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

[5] Y. Yarom and K. Falkner, "Flush reload: A high resolution, low noise, L3 cache side-channel attack," in *Proc. 23rd USENIX Secur. Symp.*, 2014, pp. 719–732.

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

[7] S. Kim et al., "ReViCe: Reusing victim cache to prevent speculative cache leakage," in *Proc. IEEE Secure Develop.*, 2020, pp. 96–107.

[8] G. Saileshwar and M. K. Qureshi, "Cleanupspec: An" undo" approach to safe speculation," in *Proc. IEEE/ACM 52nd Annu. Int. Symp. Microarchitecture*, 2019, pp. 73–86.

[9] N. Binkert et al., "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011. [Online]. Available: https://doi.org/10.1145/2024716.2024718

[10] A. Limaye and T. Adegbija, "A workload characterization of the SPEC CPU2017 benchmark suite," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2018, pp. 149–158.

[11] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," in *Proc. 24th USENIX Secur. Symp.*, 2015, pp. 897–912.

[12] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush flush: A fast and stealthy cache attack," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*, Springer, 2016, pp. 279–299.

[13] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "Dawg: A defense against cache timing attacks in speculative execution processors," in *Proc. IEEE/ACM 51st Annu. Int. Symp. Microarchitecture*, 2018, pp. 974–987.

[14] J. Kong, O. Aciicmez, J.-P. Seifert, and H. Zhou, "Deconstructing new cache designs for thwarting software cache-based side channel attacks," in *Proc. 2nd ACM Workshop Comput. Secur. Architectures*, 2008, pp. 25–34.

[15] 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, Sep./Oct. 2016.

[16] M. K. Qureshi, "CEASER: Mitigating conflict-based cache attacks via encrypted-address and remapping," in *Proc. IEEE/ACM 51st Annu. Int. Symp. Microarchitecture*, 2018, pp. 775–787.

[17] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard, "ScatterCache: Thwarting cache attacks via cache set randomization," in *Proc. 28th USENIX Secur. Symp.*, 2019, pp. 675–692.

[18] F. Liu and R. B. Lee, "Random fill cache architecture," in *Proc. IEEE/ACM 47th Annu. Int. Symp. Microarchitecture*, 2014, pp. 203–215.

[19] R. Brotzman, D. Zhang, M. Kandemir, and G. Tan, "Ghost thread: Effective user-space cache side channel protection," in *Proc. 11th ACM Conf. Data Appl. Secur. Privacy*, 2021, pp. 233–244.

[20] J. P. Thoma et al., "Clepsydracache–preventing cache attacks with time-based evictions," 2021, *arXiv:2104.11469*.