IE-Cache: Counteracting Eviction-Based Cache Side-Channel Attacks Through Indirect Eviction

Muhammad Asim Mukhtar, Muhammad Khurram Bhatti, Guy Gogniat

IE-Cache: Counteracting Eviction-Based Cache Side-Channel Attacks through Indirect Eviction

M. Asim Mukhtar[1], M. Khurram Bhatti[1], and Guy Gogniat[2]

[1] Information Technology University, Lahore, Pakistan
asim.mukhtar@itu.edu.pk, khurram.bhatti@itu.edu.pk
[2] Université Bretagne Sud, Lorient, France
guy.gogniat@univ-ubs.fr

Abstract. Protecting critical information against eviction-based cache side-channel attacks has always been challenging. In these attacks, attacker reveals secrets by observing cache lines evicted by the co-running applications. A precondition for such attacks is that the attacker needs a set of cache lines mapped to memory addresses belonging to victim, called *eviction set*. Attacker learns eviction set by loading the cache lines at random and then it observes their evictions as a result of victim access. We have found that the relation between the incoming memory location and the resulting evicted cache line eases the learning of an eviction set. In this paper, we propose *Indirect Eviction Cache (IE-Cache)* that is based on the principle of indirect eviction to harden the building of eviction set. In an eviction process of IE-Cache, incoming memory triggers series of replacements based on the cached memory addresses and a secure-indexing function, and the last replaced cache line is evicted. This increases the set size and introduces non-evicting cache lines in the eviction set. Through experimental results, we have shown that a 4-way set associative IE-Cache having 1MB and up to 3 replacements per eviction would require an attacker to generate $\approx 2^{59}$ memory accesses to learn an eviction set with 99% confidence. Moreover, it achieves $1-3\%$ speedup compared to set-associative cache with a random-replacement policy on PARSEC benchmarks.

Keywords: Cache-based side-channel attack · Randomization · Encrypted cache space · Prime+Probe attack.

1 Introduction

Caches are main component of modern computer systems that bridge the performance gap between processor and main memory. Caches are usually shared among applications for efficient utilization of cache space. However, this sharing turns out to be a high-security threat. In particular, the eviction behavior of shared cache lines can reflect the secrets of secure applications to other applications. Eviction behavior can be extracted from the cache using eviction based cache side-channel attacks, which initialize cache lines in such state that victim access of interest (or secure dependent memory access) has to evict attacker's

cache line. In the past few decades, the research on eviction-based cache-side channel attacks mainly focused on extracting secret keys of cryptographic algorithms [1][2][3][4]. Recent advancements in such attacks (Spectre [5] and Meltdown [6]) extend the security threat that these attacks can read all unauthorized memory space.

To mitigate these attacks, state-of-the-art hardware-based countermeasures have been proposed in past few decades [7][8][9][10][11][12], which can broadly fall into two categories partition-based and randomization-based solutions. Partition-based solutions divide the cache among applications to make eviction behavior independent among applications. However, efficient partitioning of cache among applications is NP-Hard problem [13]. Moreover, increasing-trend of a number of cores on-chip is pushing computer architecture toward cache scalability, therefore, partitioning the already low capacity cache for security exaggerates the cache scalability requirement. On the other side, randomization-based solutions can mitigate without limiting the cache space among applications. These solutions randomize the memory-to-cache mapping to increase the attacker's difficulty in finding all memory addresses that would contend with victim memory addresses, also called an eviction set. Unfortunately, existing randomized based solutions are limited in the scope of security. Such as a recently proposed solution called as ScatterCache [14] claimed that building eviction set requires 38 hours but research work by [15] has proved that advanced profiling techniques can be used to reduce 38 hours to less than 5 minutes. Currently, there is no countermeasure that achieves security against eviction-based cache attacks with preserving the sharing feature of the cache. The goal of this paper is to propose a countermeasure while preserving the sharing feature because shared caching naturally adjusts the allocation for each application dynamically and does not need extensive profiling of all applications before allocating cache locations as it is required in case of cache partitioning.

To mitigate the eviction-based cache side-channel attacks in a shared cache, the key challenge is to make attacker unable to build an eviction set. In conventional caches, on the insertion of a new memory block, the incoming block replaces one cache line preferred by replacement policy (or colliding cache line) and then the replaced cache line is evicted from the cache (or evicting cache line). We provide key insight that the building of an eviction set becomes impractical if the colliding and evicting caches lines are different. To propose a solution based on this insight, we present a novel cache architecture, called *IE-Cache*. In an eviction process, incoming memory address replaces one of the cache lines at random, and then the replaced cache line replaces another cache line. This repeats until replacement limit is achieved and the last replaced cache line is evicted. This introduces multiple cache lines in between the incoming address and evicting cache line, yielding benefits in two ways regarding the eviction set. First, it increases the size of the eviction set. Secondly, the eviction set includes cache lines that do not evict as a result of accommodating the incoming address, we call as non-evicting cache lines of eviction set, finding these cache lines without the side information (eviction behavior) enormously increased the com-

plexity of building eviction set by the attacker. We are saying non-evicting with respect to one address but it may evict by another address. We have shown experimentally that it is impractical for the attacker to find all memory addresses that directly and indirectly collide with the victim's access. The contributions of this paper are as follows:

- We propose a countermeasure that mitigates eviction based cache side-channel attacks in a novel way by making the indirect relation between incoming address and evicting cache lines.
- We have experimentally analyzed the security of IE-Cache by demonstrating that it is impractical to build the eviction set by the attacker. This inhibits the attacker to launch eviction-based cache attacks.
- We evaluate the performance impact of IE-Cache in comparison with the set-associative cache architecture having random replacement policy while running PARSEC benchmark using *zsim* simulator [16].

The rest of the paper is organized as follows. Section 2 gives the necessary background. Section 3 presents the IE-Cache. Sections 4 and 5 present the security and performance evaluation of the IE-Cache respectively. Section 6 concludes the paper.

2 Background

In this section, we provide the background on eviction-based cache attacks and Prime+Probe attack.

2.1 Eviction-Based Cache Attacks

Various eviction-based cache side-channel attacks have been proposed in the literature [1][2][3][4]. In these attacks, an adversary finds cache lines that are mapped to victim application and initializes them in an interesting state. Then observes whether the initialized state is changed or not after the victim execution, which results in the extraction of victim secrets in the form of victim accesses. In Prime+Probe attack [3], an adversary loads the cache lines and observes evictions of loaded cache lines by the victim's memory accesses. Evict+Reload attack [2] is similar to Prime+Probe attack except it can only be launched if adversary and victim share memory lines. Flush+Reload attack [4] is similar to Evict+Reload attack except an adversary makes a cache state by flushing instead of loading cache lines. The Flush+Flush [1] attack is a variant of Flush+Reload attack in which an adversary observes the state of cache lines by measuring the time to flush the cache lines instead of reading the memory lines. Evict+Reload, Flush+Reload and Flush+Flush attacks can be successfully launched if the memory deduplication feature is enabled in the system. However, the memory deduplication feature is usually disabled in shared computing environments such as cloud computing [17]. Prime+Probe attack is not dependent

on memory deduplication and requires commonly used instructions such as *mov* and *rdtsc* instructions. Mitigation of Prime+Probe attack is difficult because it can be launched through any instruction that can load the cache such as *mov*, *add*, *jmp*, etc. It is impossible to disable all these instructions, therefore, major modifications are required in the computing stack to mitigate the Prime+Probe attack.

2.2 Prime+Probe attack

Memory accesses can be backtracked to secure information of application. Prime+Probe attack enables an application to extract secret dependent memory accesses of co-running applications by observing the eviction of cache lines. In this attack, the attacker reserves all cache lines where victim memory location of interest can reside in the cache. If victim accesses that memory location, it will evict one of the attacker cache lines, revealing its memory access to the attacker by the action of eviction. However, for a successful attack, the attacker requires memory addresses (or colliding addresses) that share the cache lines with the targeted victim address. In conventional caches like set-associative cache, the mapping of memory-to-cache is static and well-studied in research. Finding colliding addresses in such caches only depends on the indexing bits (specified by the designer) of the memory address that all memory addresses having the same indexing bits will collide in the cache. However, the adversarial effort to learn the colliding addresses has greatly increased in caches that define the mapping of memory-to-cache at run-time and change it over time. The attacker has to find colliding addresses on each change of memory-to-cache mapping through extensive experiments. For finding colliding addresses in such caches, the attacker goes through the following steps.

- The attacker randomly chooses N memory addresses and loads them into the cache by accessing them. As attacker has randomly chosen the memory addresses, there is a possibility that these memory addresses collide with each other and cause eviction of group members. To eliminate self-collisions in group G, attacker reads the accessed memory addresses again and observes there access latency. If attacker observes a longer time it means that memory address is evicted because of self-collision and needs removal from the group. Attacker iterates the action of accessing and removing memory addresses until all memory access results shorter time. Let n_{rmv} indicates the number of iterations required to eliminate self-collisions. The attacker now has a set of G' $\leq$ G addresses, which are guaranteed to reside at a different location in the cache.
- The attacker calls the victim to access memory, expecting an eviction by victim access if correctly sampled G' memory addresses.
- After that attacker accesses G' addresses again and measures their memory access latency to observe the eviction. Attacker will find the colliding address in case of longer access time is observed.

Then attacker repeats the above steps until enough addresses are obtained for the attack, which also depends on the parameter of cache architecture. Note that after the first iteration, the victim access of interest is in the cache and need to be evicted before next iteration. Therefore, an attacker has to access many different memory addresses to ensure the eviction of victim cache line.

3 IE-Cache - Proposed Cache

The objective of IE-Cache is to eliminate the direct eviction of cache line as a result of inserting new memory location to counter eviction-based cache side-channel attacks. We consider that an adversary has access to all user-level instructions except those which are related to cache management such as $clflush$ and $prefetchtx$. Flush+Flush and Flush+Reload attacks cannot be launched because of non access to $clflush$ instruction. Moreover, physical attacks are not considered in the threat model of IE-Cache. In addition to achieving security, we also focus to retain the fundamental design features of cache such as transparent to the user and less reliance on OS. In the following section, we discuss the design and working of IE-Cache.

3.1 IE-Cache: High level Design

IE-Cache is inspired form Zcache architecture [18], which is proposed to improve the performance by increasing the associativity without increasing the physical ways. However, Zcache is also vulnerable to cache-based side-channel attacks. We have replaced the static hash function with the cipher function to make it resilient to cache-based side-channel attacks.

Figure 1a illustrates the high-level design of IE-Cache with 8 cache lines and 3 ways. IE-Cache employs key-based indexing function for each way. These functions use multiple keys - one for secure and other for a non-secure domain. One bit is added in each cache line to distinguish the secure and non-secure cached data. IE-cache performs eviction of cache line in multiple steps. First, it searches cache lines for eviction in multiple levels. Then it selects the candidate using random replacement policy and evicts it. lastly, it relocates the cache lines to accommodate the incoming block. Figure 1c illustrates the eviction process for the accommodation of non-secure memory block (Y) in IE-Cache having 2 levels of search. In first level, Incoming memory block belonging to a non-secure domain (Y) selects the replacement cache lines (or candidates) using the non-secure key, let say, it selects E (non-secure), A (secure) and C (non-secure). Then in second level, the selected candidates further selects the cache lines using their domain specific key shown in the second level of Figure 1c. Random replacement policy chooses a candidate for eviction from last level of search and evicts it. Lastly, series of relocation happens to maintain the cache organization. Let us consider that random replacement policy selects T for eviction then A will be relocated to T and Y will be added to A location. There are two important points here. First, A is moved to other location of its own interest and can become a member

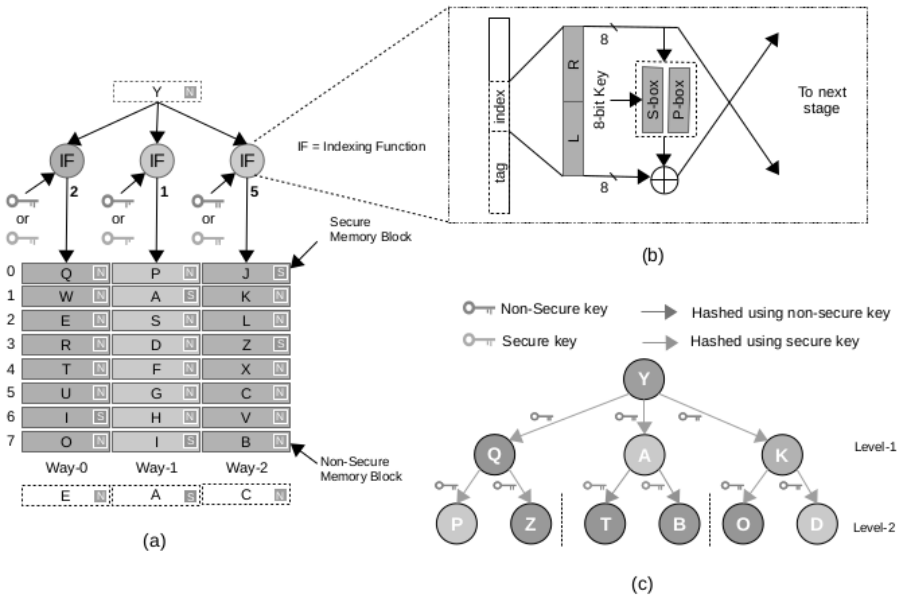

Fig. 1. High level design of IE-Cache (a) Cache architecture (b) Indexing function (c) Replacement candidates tree

of other cache set, which also means even if Y is evicted and requested again for accommodation in the cache, it is not necessary that A is again a member of Y. Second, as the key is managed by hardware and kept secret, an adversary does not know the relation among cache lines and results in unknown evictions. If adversary tries to find a set of memory addresses that can cause eviction of desired cache line using random evictions, then he finds the evicting cache lines belongs to last level only. To find the intermediate level cache lines, attacker requires to generate large number of memory accesses to find one non-evicting cache line, which we have analyzed in the Section 4.

3.2 Suitable Indexing Function

The main objective of the indexing functions in IE-Cache is to achieve sufficient pseudo-random permutation with low latency. We observe that different indexing functions (i.e. QARMA and DES) used in previous randomization-based countermeasures [14][19] can fulfill our objectives, and the scope of security of these countermeasures are limited because of the direct eviction but not because of indexing functions used in them. Therefore, these indexing functions can also be used in IE-Cache. We find block cipher used in CEASER [19] most suitable for IE-Cache because it gives flexibility in the input size and incurs low latency of about one or two cycles. For demonstration purpose, 8MB cache with 4 ways, we have implemented cipher having 3 stages with 16-bit input and output. Fig-

ure 1b shows one stage of DES. For the detailed implementation of cipher, we recommend the reader to see [19].

3.3 Security Domain and Key Management

The concept of hardware managed key and creation of security domain are not new as these are already in Intel SGX [20] and ARM TrustZone [21] architectures. These processors are also vulnerable to cache-based side-channel attack. Therefore, IE-Cache suits for such architectures that already have a secure bit in cache lines and hardware managed key mechanism because these architectures require localized modification if IE-Cache is integrated into such platforms.

The key in our IE-Cache plays a crucial role in the security, therefore, its confidentiality is important. We ensure this confidentiality in our design by mandating that the key is fully managed by hardware. There must not be any way to configure or retrieve this key in software. Each time the system is powered up, a new random key is generated.

3.4 Increased Complexity of Prime+Probe

IE-Cache makes both profiling and exploitation steps of Prime+Probe harder by increasing the eviction set size and introducing the non-evicting members in the eviction set.

In the eviction process of IE-Cache, cache lines are selected in multiple levels such that the number of cache lines increases exponentially from one level to the next. For example, for 4 ways and 2 levels, cache lines selected in first and second levels are 4 and 12 respectively. For a successful Prime+Probe attack, the attacker needs to fill all 16 cache lines to observe the eviction. However, because of the random replacement policy, the attacker cannot guarantee the memory access will be placed in the targeted cache line. In case of memory accessed by attacker is placed at the cache line other than the targeted one, attacker has two options to fill the cache line 1) attacker flushes the cache and accesses the same memory address again or 2) attacker accesses another memory address that can be placed in the cache line. Because of no access to flush instruction, the attacker has to indirectly flush using random memory accesses, which is costly. Therefore, the attacker has to adopt the second option and needs to access multiple memory addresses to ensure the filling of the targeted cache line. We analyze the expected number of memory addresses required by the attacker using bins and balls analysis that how many throws are required to fill the interested bin with at least one ball with 99% confidence. Considering IE-Cache having 4-ways and 2 levels, this analysis results that the attacker requires 16 memory addresses to ensure filling of one cache line and 64 memory addresses for four cache lines. Accessing these memory addresses by the attacker will ensure filling of cache lines belonging to the first level of the tree but which four memory addresses have successfully been placed in the targeted cache lines are unknown to the attacker. Therefore, to observe the victim access, the attacker has to load all those cache lines that can be selected by 64 memory addresses on the second level of the

tree, so that relocation from the first level to second level causes eviction. As each cache line related to the first level selects 3 cache lines on the next level in IE-Cache having 4 ways, total possible cache lines selected by 64 cached memory addresses are 192 at the second level. The attacker needs 16 memory addresses (using the same bin and ball analysis) to ensure filling of each 192 cache lines, which means 3072 memory addresses are required to fill the second level. This concludes that the eviction set size is 3136 cache lines for IE-Cache having 4 ways and 2 levels in which the number of evicting (first level) and non-evicting (second level) cache lines are 64 and 3072. The attacker in profiling steps needs to find 3136 memory addresses against one victim's memory access. Also in exploitation step attacker needs to load 3136 memory addresses, yielding difficulties for the attacker in two ways. First, attack resolution decreases enormously because of too many memory loads are needed. Second, given that the cache line is of 64 bytes, 3136 cache lines cover 196 KB scattered memory space, hence it is difficult to identify the victim's memory access in case of concurrent processing of multiple applications. Figure 2 shows the number evicting and non-evicting addresses in an eviction set for a varied number of ways and levels of IE-Cache.

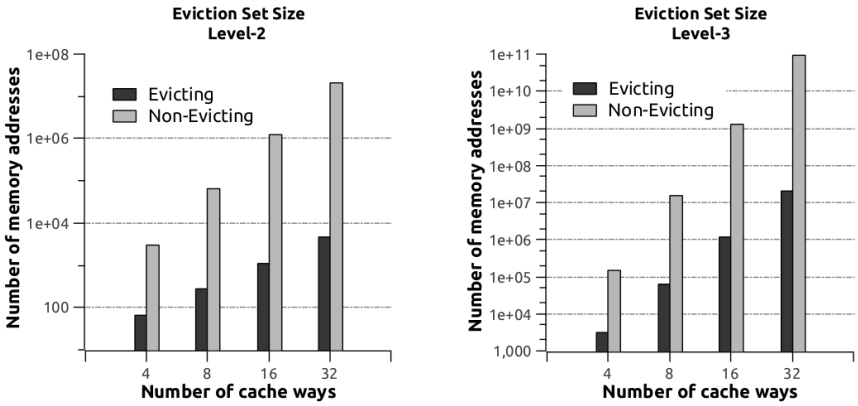

Fig. 2. Eviction set size for varied number of ways and levels of IE-Cache

Finding non-evicting members of the eviction set is another difficulty for the attacker. Victim memory access causes eviction from the last level and relocates the cache lines belonging to upper levels. As the non-evicting cache line does not evict as a result of victim access, there is no direct information to an attacker via timing channel. However, we find that the attacker can use indirect timing analysis to know the relation between non-evicting and evicting cache lines. For example in Figure 1c, eviction of P (evicting cache line) is dependent on the presence of Q (non-evicting cache line) in the cache, which means that P will not evict if Q is not in the cache. In the profiling step, let say that the attacker has successfully found the P using profiling technique discussed in Section 2.2, the attacker knows that Q is also in group G' (set of randomly selected non-colliding memory address). To find Q, the attacker access all memory addresses

belonging to G' group again except the one randomly selected address (expecting a Q). Then the attacker calls victim to access memory, aiming that the cache line P will not evict if randomly taken out address is Q. After this, the attacker accesses P again while measuring its time to know the eviction. The attacker will observe the eviction of P on taken out candidate multiple times that the random replacement policy at-least selects P once for eviction if the taken out candidate is not Q. If the attacker finds that taken out member has reduced the probability of eviction of P in multiple runs, the attacker expects taken out member is Q. In case of more levels, for example for three levels, evicting member depends on two non-evicting members, hence, the attacker has to observe eviction of interested last level cache lines by taking out all possible pairs formed in G' group. This greatly increases the time to find non-evicting cache lines as attacker needs $\approx 2^{27}$ years to finds one eviction set for IE-Cache having 4 ways, 3 levels and 2^{12} cache lines per way. Note that the attacker has to flush and place the group members again in cache for observing the eviction behavior on the next taken out member. Placing of non-colliding group members again in cache becomes difficult because there is a probability that two addresses that do not collide in one arrangement of placement may collide in other. Accessing the same memory again does not guarantee the placement in the same cache way as placed in the previous turn because of random replacement policy. Therefore, ensuring the placement of all group members in the cache, attacker has to access all group members multiple times while measuring their access latency. If attacker finds any access with longer latency, it has to repeat the action of flush and placing members again, aiming to place in such arrangement that group members do not cause self evictions. let n_{pl} indicate the number of iteration required to place the non-colliding member of group in the cache.

4 Security Evaluation

For security evaluation, we have built the python model of IE-Cache and have considered the following assumptions. First, the cache has a random replacement policy and fills the invalid location with high priority. Second, the mapping from memory address to output indices is pseudo-random. Lastly, attacker and victim processes are executing only. We have taken this assumption in favor of the attacker.

We have verified the security of IE-Cache by analyzing the number of attacker and victim memory accesses needed to build an eviction set for Prime+Probe attack as discussed in section 2.2). As victim memory access results in eviction of cache lines belonging to last level in IE-Cache, profiling discussed in section 2.2 can only find the evicting cache lines belonging to an eviction set. For non-evicting cache lines in an eviction set, we used the method discussed in section 3.4. We experimentally obtained the attacker and victim accesses to find evicting and its non-evicting member averaged over 100,000 simulator runs. Then we multiplied the accesses obtained for one evicting and non-evicting member with the total number of respective members (shown in Figure 2) in an eviction set.

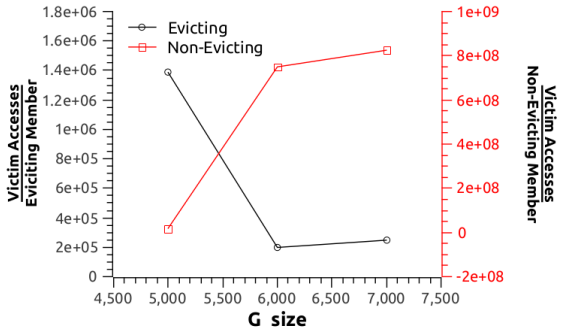

Fig. 3. Victim accesses required to find evicting and non-evicting cache lines of an eviction set of IE-Cache having 4 ways, 2^{11} lines and 2 levels.

Figure 3 shows the victim accesses and attacker accesses per victim access for IE-Cache having 4 ways, 2^{11} cache lines and 2 levels. As the group size increases, results indicate that the victim accesses become less to find evicting cache line, making it easy for the attacker. However, victim accesses become greater to find non-evicting cache line, making it harder for the attacker. This inverse effect indicates that advanced profiling fails in the case of IE-Cache.

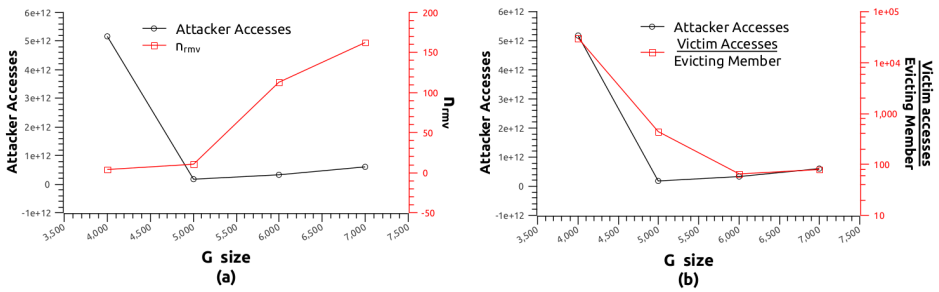

Fig. 4. Attacker accesses required to find evicting cache lines against (a)n_{rmv} and (b) Victim calls required to find evicting cache lines.

Figure 4 shows the attacker accesses required to find evicting cache lines versus different parameters for IE-Cache having 4 ways, 2^{11} cache lines and 2 levels. Figure 4a shows that the group size of 5000 requires less number of attacker accesses as compared to other sizes. We have observed that the iteration required to eliminate the self collisions increases if group size becomes larger than 5000 (shown in Figure 4a), which increases the attacker accesses. Inversely, for group size smaller than 5000, attacker finds evicting cache lines in a greater number of turns (shown in Figure 4b), which increases the attacker accesses.

Table 1 presents the adversarial effort required to find evicting and non-evicting cache lines of an eviction set for different parameters of IE-Cache. For time calculation, we had taken the same assumption as in research work [14], i.e, 9.5ns cache hit time, 50ns cache miss time, 0.5ms victim execution time and 3.6ms cache flush time. Results show that the time to build an eviction set is increased with the increase in level and cache lines. This is because the non-evicting cache lines are increased in an eviction set. Moreover, results show that a bigger group is required to find the evicting line in caches having a greater number of cache lines, which increases the iterations to check evictions of evicting members against non-evicting members and yields increase in time.

Table 1. Adversarial effort to find eviction set in 4 way IE-Cache having different levels and cache lines. CL = number of cache lines, L = number of levels, G=size of randomly sampled memory addresses, G'=non-colliding member of group, n_{rmv}=turns required to find non-colliding addresses, n_{pl}=number of turn required to place non-colliding members in cache, avg_v=average victim access and avg_a=average attacker accesses per victim access.

CL	**L**	**G (k)**	**G'**	n_{rmv}	n_{pl}	**Evicting**			**Non-Evicting**		
						avg_v	avg_a	**Time (hr)**	avg_v	avg_a	**Time (hr)**
2^{11}	2	7	5720	163	64	2.46E5	5.98E11	22	2.53E12	1.07E16	2.69E06
		6	5582	113	28	1.97E5	3.33E11	13	2.30E12	3.33E15	2.43E06
		5	4993	11	3	1.36E6	1.77E11	7	5.42E10	1.17E12	5.95E04
		4	4000	4	1	9.26E7	5.07E12	213	1.08E10	2.82E11	1.29E04
2^{11}	3	8	6945	90	67	1.06E7	9.58E12	191	4.99E16	1.28E16	2.32E12
		7	6866	73	21	2.48E7	7.37E12	342	1.12E18	1.66E18	1.21E12
2^{12}	2	16	11497	430	71	3.96E5	2.82E12	58	5.13E13	4.06E17	5.84E07
		15	11224	367	61	2.15E5	1.64E12	34	3.23E13	2.45E17	3.67E07
		14	11035	318	49	1.69E5	1.20E12	25	1.56E13	1.39E17	1.91E07
		13	10893	218	38	2.95E5	2.24E12	46	1.26E12	1.22E16	1.43E06
		12	10542	205	10	4.64E5	2.99E12	63	7.82E11	5.86E15	8.88E05
2^{12}	3	16	13446	154	56	1.12E7	5.71E13	293	5.71E17	1.13E17	6.79E12
		15	13554	142	84	1.46E7	6.69E13	1121	9.82E18	2.96E21	1.34E13
		14	13292	144	40	3.10E7	1.93E13	1336	2.05E18	2.67E18	2.40E12
		13	12974	17	25	5.85E7	8.28E13	1657	1.94E18	3.95E20	2.27E12

5 Performance Evaluation

We have used micro-architecture simulator, *zsim* [16], for performance evaluation of IE-Cache. Table 2 shows the configuration used in our experimental setup. A 4-way IE-Cache is introduced at the L3 in the cache hierarchy and it is designed for a different level-of-search, i.e. 2 and 3 levels. We have evaluated the IE-Cache for each level-of-search against baseline architecture by executing 11 workloads of PARSEC using *medium* input set. In each run, we have taken one program at random for each domain (secure and non-secure). We have calculated the weighted speedup metric, which is a sum of ratios of the program's IPC executing in a group to its IPC when it is executing in isolation, and normalized it to baseline architecture.

Table 2. Baseline Configuration

Core	4 cores, 2.2 GHz, OoO model
L1 cache	Private, 32kB, 8-way set associative, split D/I
L2 cache	Private, 256kB, 8-way set associative
L3 cache	Shared, 1MB, 16-way set associative or 4-way IE-Cache
Memory	200-cycle latency, 8GB/s peak memory BW

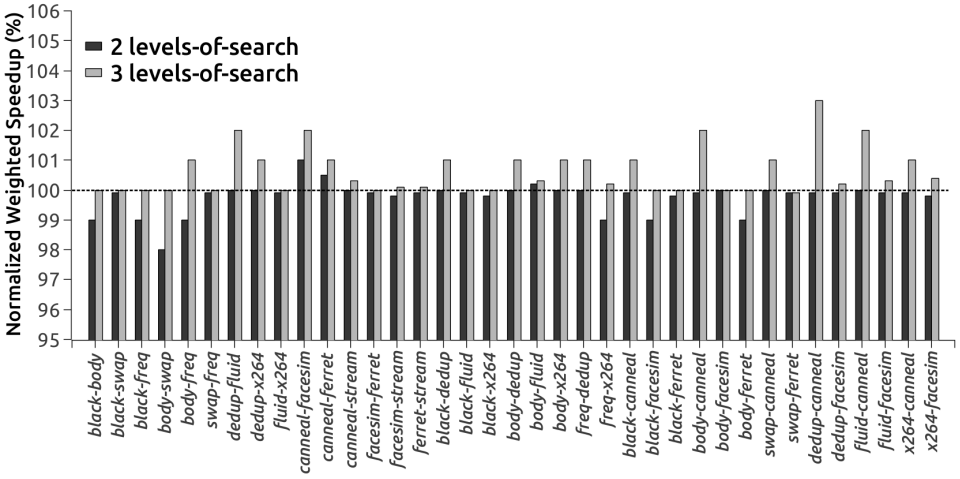

Fig. 5. Normalized Performance of IE-Cache with 2 and 3 levels

Figure 5 shows the normalized weighted speed up metric of IE-Cache. In this figure, a bar higher than 100% indicates performance improvement of IE-Cache as compared to baseline architecture. Results in Figure 5 show that IE-Cache with 3 levels-of-search outperforms 1% - 3%. This performance is improved because of the scattered mapping of memory to cache lines and increased cache associativity. Workload *canneal* and *dedup*, which frequently use L3 cache, shows an improvement of about 3% because of the increased associativity of IE-Cache. IE-Cache with 2 levels-of-search shows low performance on some of the load because of low associativity as compared to baseline.

6 Conclusion

This paper proposes a novel way to mitigate eviction-based cache side-channel attacks while retaining the shared feature of the cache. We have modified the relation between the incoming memory address and evicting cache line that introduces non-evicting members in the eviction set, which are harder to learn by collisions. The attacker has to generate at least $\approx 2^{59}$ memory accesses to find an eviction set. This takes about $\approx 2^{27}$ years to find one eviction set. Moreover, IE-Cache provides strong security with $1 - 3\%$ improvement in performance.

References

1. Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+flush: A fast and stealthy cache attack. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721*, pages 279–299. Springer-Verlag, 2016.
2. Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. Armageddon: Cache attacks on mobile devices. In *25th USENIX Security Symposium*, pages 549–564, Austin, TX, 2016. USENIX Association.
3. 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*, pages 605–622, May 2015.
4. Yuval Yarom and Katrina Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, San Diego, CA, 2014. USENIX Association.
5. Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.
6. Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.
7. Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 189–204, Bellevue, WA, 2012. USENIX.

8. 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)*, pages 974–987, Oct 2018.
9. Jingfei Kong, Onur Aciicmez, Jean-Pierre Seifert, and Huiyang Zhou. Deconstructing new cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 2Nd ACM Workshop on Computer Security Architectures*, CSAW '08, pages 25–34. ACM, 2008.
10. F. Liu, H. Wu, K. Mai, and R. B. Lee. Newcache: Secure cache architecture thwarting cache side-channel attacks. *IEEE Micro*, 36(5):8–16, Sep. 2016.
11. Fangfei Liu, Qian ge, Yuval Yarom, Frank McKeen, Carlos Rozas, Gernot Heiser, and Ruby Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. 03 2016.
12. F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *2016 HPCA*, pages 406–418, March 2016.
13. Ugo Fiore, Adrian Florea, Arpad Gellert, Lucian Vintan, and Paolo Zanetti. Optimal partitioning of llc in cat-enabled cpus to prevent side-channel attacks. In *Cyberspace Safety and Security*, pages 115–123, Cham, 2018. Springer.
14. Scattercache: Thwarting cache attacks via cache set randomization. In *28th USENIX Security Symposium*, Santa Clara, CA, 2019. USENIX Association.
15. Antoon Purnal and Ingrid Verbauwhede. Advanced profiling for probabilistic prime+probe attacks and covert channels in scattercache. *ArXiv*, abs/1908.03383, 2019.
16. Daniel Sanchez and Christos Kozyrakis. Zsim: fast and accurate microarchitectural simulation of thousand-core systems. *ACM SIGARCH Computer Architecture News*, 41:475, 07 2013.
17. Fernando Vañó-García and Hector Marco-Gisbert. Slicedup: a tenant-aware memory deduplication for cloud computing. In *UBICOMM 2018*, UBICOMM International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies, pages 15–20, United States, 11 2018. IARIA.
18. D. Sanchez and C. Kozyrakis. The zcache: Decoupling ways and associativity. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 187–198, Dec 2010.
19. Moinuddin K. Qureshi. Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping. pages 775–787, 10 2018.
20. Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. Intel® software guard extensions support for dynamic memory management inside an enclave. HASP, pages 10:1–10:9, New York, NY, USA, 2016. ACM.
21. Wenhao Li, Yubin Xia, and Haibo Chen. Research on arm trustzone. *GetMobile: Mobile Comp. and Comm.*, 22(3):17–22, January 2019.