PCache: Permutation-based Cache to Counter Eviction-based Cache-based Side-Channel Attacks

Muhammad Asim Mukhtar
Department of Electrical Engineering
Information Technology University
Lahore, Pakistan
Email: asim.mukhtar@itu.edu.pk

Muhammad Khurram Bhatti
Department of Computer Engineering
Information Technology University
Lahore, Pakistan
Email: khurram.bhatti@itu.edu.pk

Guy Gogniat
Lab-STICC Laboratory, CNRS
Université Bretagne Sud
Lorient, France
Email: guy.gogniat@univ-ubs.fr

***Abstract*—Eviction-based cache-based Side-Channel Attacks (SCAs) are continuously increasing confidentiality issues in computing systems. To mitigate these attacks, randomization-based countermeasures have raised interest because these have the potential to achieve strong security and high performance while retaining the cache features such as high-associativity and operate without the involvement of system software. However, existing countermeasures are proved to be less secure because of the small eviction set size or weak indexing functions used in them. To cope with this issue, we propose a novel randomization-based architecture, called PCache, which introduces hidden members in the eviction sets to enlarge their size, which makes it difficult for an attacker to launch eviction-based cache-based SCAs. PCache replaces cache lines in multiple steps by passing through different permutation functions, which consider bits of tag and index part of the memory address in the replacement process and result in strong indexing function. Experimental evaluations show that PCache provides high security. For a 10MB cache, an attacker needs 2 years to find the eviction set and can launch eviction-based cache-based SCAs with only 28% confidence level. Moreover, PCache performance overhead is only 1.6% at maximum as compared to classical set-associative caches.**

Keywords–Cache-based side-channel attacks; Randomization; Prime+Probe attack.

I. INTRODUCTION

Caches are the main component of a computer that contributes significantly to performance and purposefully each aspect of them is designed to achieve maximum performance. However, performance-based designs raise confidentiality issues. These designs enable cache-based Side-Channel Attacks (SCAs) such that a process can extract the memory access-patterns that indirectly reflect the secrets of co-running processes [1]–[4].

Various cache-based SCAs have been proposed. The prominent one is eviction-based cache-based side-channel attack, where an attacker intentionally fills cache lines with the memory blocks, called eviction set, so that eviction triggers on victim accesses. Recent research works have shown that these attacks can recover keys of cryptographic algorithms [1]–[4], detect user keystrokes [5], and combining with other side-channel can read unauthorized address space of system software or applications [6] [7]. Moreover, these attacks can exploit widely used architectures especially Intel and ARM, and can also extract the secret information of application executing in hardware-assisted trusted execution environments like Intel-SGX and ARM-TrustZone [8].

Numerous countermeasures have been proposed these recent years, which can broadly be categorized into partitioning and randomization approaches. Partitioning-based techniques [9]–[11], which statically divide the cache into multiple non-interference domains, provide strong security but degrade performance, which is the main purpose of caches. Randomization-based techniques [12]–[14], which make eviction set confidential by random memory-to-cache mapping, provide better performance but lead to weak security. Recent research works showed that the eviction set can be revealed using the Prime-Prune-Probe attack in a practically feasible time [15] [16]. However, we observe that the randomization-based techniques can provide security by making eviction set large and introducing new type of member called hidden members, which are relocated as a result of accommodating memory block in cache, making eviction process confusing for the attacker. This greatly increases the effort of the attacker such that the eviction-based cache SCAs become impractical.

The goal of this work is to reduce the limitation of randomization-based techniques to improve security. We propose PCache an architecture that evicts a cache line via a series of relocation using already stored content. This introduces hidden members in an eviction set, which cannot be learned using the Prime-Prune-Probe attack. Moreover, relocated members explore all their possible cache locations where they can reside in cache. These all cache locations become members of eviction set, which exponentially increases the number of members in eviction set. We show that PCache provides strong security and high performance without reducing the associativity and involvement of system software. Our main contributions are:

- We propose PCache, which achieves strong security against eviction-based cache-based SCAs by a novel approach of making eviction set size large and introducing hidden members in the replacement process.
- We evaluate the security of PCache by estimating the effort required by an attacker to learn the eviction set.
- We find new approaches that can be used to launch eviction-based cache-based SCAs on PCache. These are *Exclude-Prime-Probe*, which is an approach to find the hidden members of the eviction set, and *Eviction Distribution Estimation*, which is an approach to launch eviction-based cache-based SCAs without the need of learning hidden members of eviction set.
- We build PCache in Champsim simulator, which is a

 ISBN: 978-1-61208-818-1

trace-driven simulator, and compare the performance of PCache with the set-associative cache using SPEC CPU2017 benchmark.

Section II presents the background and related work. Section III presents the PCache architecture and operation. Section IV discusses the security perspective of PCache and new approaches to attack PCache. Sections V and VI present the experimental evaluation of security and performance, respectively. Section VII concludes the work.

II. Background and Related Work

A. Conventional Cache

Caches are a type of memory that is faster and smaller than the main memory. Caches buffer the memory blocks for the near future so that if the processor demands those blocks, they will be bring from cache rather than the main memory, resulting in reduced memory access latency. A basic storage cell of cache is called the cache line, and a group of cache lines is called a cache set. The cache line typically stores contiguous 64 bytes of main memory, which we call a memory block. Each memory block maps to one cache set but can be placed in any cache line in the cache set. The memory blocks that map to the same cache set are the conflicting blocks and cause replacement in case of cache set in full. The memory address is divided into three parts: offset, index and tag. The offset indicates the byte in the cache line. The index indicates the cache set where memory block can be stored. The tag differentiates the identity of one memory block from others in a cache set.

B. Eviction-Based Cache-Based SCAs

Numerous eviction-based cache-based SCAs have been proposed [1]–[4]. Using these attacks, an attacker finds the cache locations (lines or sets) that are shared with a victim's program, which usually has control flow dependency with secret information. Then, the attacker initializes these cache locations of its interested state (by evicting or flushing them). These cache locations will be changed if the victim accesses them. For example, in Prime+Probe attack [1], the attacker fills cache-sets by its memory lines and observes evictions of these cache-sets after the victim's execution. Evict+Reload [2] is same as Prime+Probe attack except it could be launched in case of attacker and victim share memory lines. Flush+Reload [3] is similar to Evict+Reload except the attacker initializes cache state by flushing cache-sets instead of evicting cache-sets. The Flush+Flush [4] attack is a variant of Flush+Reload attack that attacker perceives the state of cache-sets by measuring time required to flush these cache lines instead time required to access these cache lines.

C. Secure Cache Architectures

A range of countermeasures against cache-based SCAs have been proposed by modifying the cache architecture [9] [12]–[14] [17]. All of these countermeasures either partition the cache capacity or randomize the memory-to-cache mapping. The disadvantages of partitioning-based countermeasures are that these require invasive changes in software and degrade performance because of under-utilization of cache. Randomization-based countermeasure appears to be more promising. The state-of-the-art on randomization-based countermeasures are RPCache [12], NewCache [17], CEASER [14] and ScatterCache [13]. RPCache and NewCache randomize the mapping of memory lines to cache sets using permutation tables. The drawback of these countermeasures is that they require storage-intensive permutation tables, which limits the cache scalability. CESAR has proposed the concept of encrypting the memory-to-cache mapping using DES. The main drawback of CEASER is that it uses a set-associative cache that limits the encrypted space, which can be learned by an adversary in a few seconds [16]. Moreover, CEASER proposed key remapping to overcome the learning issue but this approach incurs performance degradation. ScatterCache uses hashing over skew-associative caches to randomize the memory-to-cache mapping. ScatterCache has extended the time to learn the mapping by an adversary. However, ScatterCache is proved to be less resilient to eviction-based cache-based SCAs [15].

D. Prime-Prune-Probe Attack

In this section, we present the Prime-Prune-Probe attack [15], which is used to learn the eviction set of recent randomization-based countermeasures, i.e. CEASER and ScatterCache. We explain this attack as we use it to show the security of our countermeasure. In Prime-Prune-Probe attack, the attacker chooses group, say it g, of addresses at random and fills the cache with g addresses. Then, attacker prunes the self-collision by again accessing the group addresses and removing the address from group g on observing longer access latency, which means it is evicted as a result of a collision with other members of groups. After pruning, g group contains fewer addresses, say it g'. The attacker then calls the victim program to execute, which may evict cache lines filled with g' addresses. Then, the attacker observes eviction, and if it finds eviction of g' address from cache line as a result of victim accesses, it considers evicted address as a member of the eviction set. The attacker repeats the Prime-Prune-Probe attack until it learns all members of the eviction set.

III. PCache: Permutation Based Cache

PCache achieves security against eviction-based cache-based SCAs by making large eviction set to deprive attackers of initializing cache lines with the required confidence, and it introduces cache lines in replacement process that relocate in the cache to achieve indirect eviction of another cache line, increasing the effort of the attacker to find relocating cache lines.

The objective of PCache is to mitigate eviction-based cache-based SCAs that share cache lines such as Prime+Probe and Evict+Reload attacks. We consider that an attacker has access to user-level privilege instructions except cache management related instructions such as $clflush$ and $prefetchtx$. Because of no access to $clflush$ instruction, Flush+Flush and Flush+Reload attacks cannot be launched. Moreover, physical attacks are not considered in the threat model of PCache. 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 system software.

Structurally, each way in PCache is indexed by different permutation functions. Permutation functions compute the values using incoming address to find cache locations in each way. PCache operates differently on hit and miss operation. On hit, incoming address goes through all permutation functions and

 ISBN: 978-1-61208-818-1

completes in single lookup to all ways. However, on miss, ways of PCache are seen as multiple groups and incoming address goes through permutation functions of first group only, as shown in Figure 1.

On event of cache miss, the replacement process completes in multiple steps and requires multiple lookups to ways. To understand the miss operation, we use an example, given in Figure 1, which shows PCache having 6 lines and 6 ways. Alphabets A-Z indicate the address stored in cache line and PF indicates the permutation function of each way. $G1$ and $G2$ indicate the groups of cache-ways, each group contains 3 cache-ways. V is the incoming address that triggers the process of replacement, which completes in multiple steps. First, incoming address V goes through permutation functions of $G1$ and replaces one cache-line at random from $G1$. let us assume the replaced cache line is C. Instead of evicting C, the replaced cache line C is moved to next group $G2$. let us assume the second replacement is P. Lastly, the replaced cache line in $G2$, which is P, will be evicted. In this example, there are only two groups, therefore, only one relocation happened, that is, from $G1$ to $G2$. In the case of more than two groups, the process of relocation continues until last group and evicts cache line from it.

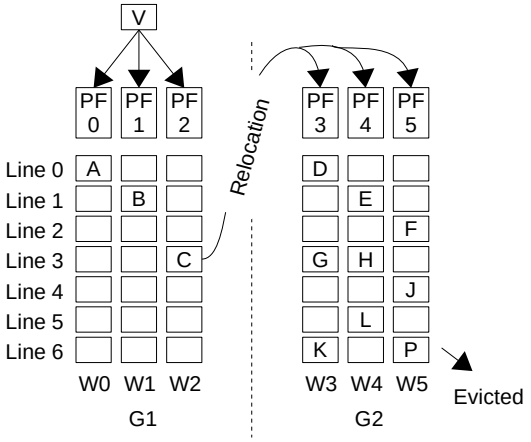

Figure 1. PCache having 6 ways, 6 lines and 2 groups. PF: Permutation function, Wx: Ways

While explaining the PCache operation, we have discussed only one path ($V \rightarrow C$ and $C \rightarrow P$) of relocations but other paths are also possible, as in each group one cache line is selected at random. Figure 2 shows all possible paths represented in tree diagram. The numbers 0-5 indicate the permutation functions used to index the ways. Alphabets A-P indicate the address stored in cache-lines selected by each permutation function. In the replacement process, there are two types of members. First, members that get evicted by incoming address, which are shown at the last level in Figure 2, we call them evicting members. Second, members that relocated to other cache lines as a result of accommodating incoming address, we call them hidden members. Note that, all cache lines belonging to path need to be filled to cause eviction of a specific line in PCache. In the perspective of security, hidden members are unknown to the attacker, and finding these require great effort. We discuss in detail the security perspective in Section IV. Note that, we refer PCache ways and groups configuration as ways/groups, for example, PCache given in Figure 1 refers as $6/2$ Pcache.

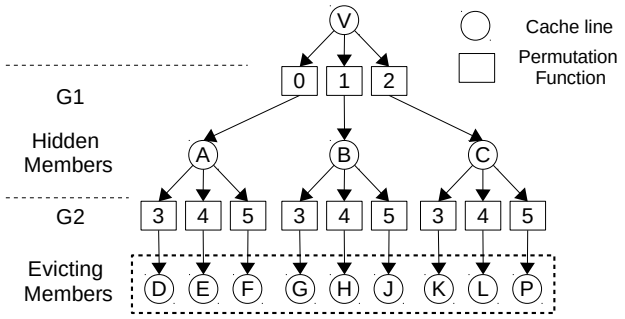

Figure 2. Cache-lines involved in replacement process represented in tree diagram.

A. Permutation Function

Selection of indexing function is critical in terms of both security and performance. In terms of security, it should not be predictable and in terms of performance, it comes in the critical path of memory access, therefore, it should be a low latency circuit. As PCache evicts the cache lines before relocating them multiple times, the attacker is limited to learn index function using cache collisions because of no direct relation of eviction between incoming and evicted cache lines. We have used a simple permutation function as an indexing function. This is better in terms of performance as it requires few gates to implement and incurs low latency. We have designed permutation functions to achieve the following objectives. First, memory addresses select different cache lines in each group, so that the member in one group does not conflict in other groups. Second, we consider tag and index bits of memory address to take part in the whole replacement process. As the system software remaps the physical mapping of application frequently, this changes the member of eviction set frequently and makes it impractical for an attacker to execute eviction-based cache-based SCAs.

IV. Security Perspective

Eviction-based cache-based SCAs have three phases. First, attacker *finds* members of eviction sets. Second, attacker *fills* cache with members of eviction sets, Third, attacker *observes* state of eviction sets in cache. PCache makes each step of attacker difficult. Because of hidden members in evicting sets, attacker's effort of learning eviction sets is greatly increased. Because of large eviction sets, filling and observing eviction sets becomes difficult while attack.

A. Finding Members of Eviction sets

Prime-Prune-Probe attack, which is an approach to learn members of eviction sets in random memory-to-cache mapping, can find the evicting members of eviction sets only. In case of PCache, this approach fails to learn the hidden members in eviction sets. This is because most of the hidden members does not evict as a result of victim access but relocate to another way. Moreover, there are members that only become a member of eviction set against interested victim address if they are placed in a specific way. In other cache-ways, these memory addresses may become a member of other eviction set.

To launch eviction-based cache-based SCAs, we find that attacker may adopt two approaches. First, the attacker tries

to learn hidden members indirectly by breaking the path in replacement candidate tree, we call *exclude-prime-probe*. Second, the attacker does not find hidden members but tries to estimate the eviction distribution against all possible victim accesses, we call *eviction distribution estimation*. We discuss both approaches in the following sections.

1) Exclude-Prime-Probe Method: Eviction of cache lines as a result of victim access indicates the presence of all hidden members in the Pcache, which can be seen as a branch of the tree. To launch eviction-based cache-based SCAs, the attacker needs to know each member of the branch to cause eviction on victim access. let us assume that the attacker has found the evicting members using a g', as discussed in Section II-D. Then, to find hidden members, the attacker again places memory addresses belonging to g' in the PCache excluding the randomly selected addresses from them, which attacker expects that these may be the parents (or hidden members) of evicting members. The number for excluded addresses depends on the number of relocations, which defines how much parents are of evicting members. After placing members again, the attacker calls victim and observes the eviction of the evicting members. Lastly, if any evicting member remains un-evicted or its probability of eviction is small relative to all turns, the attacker considers the excluded address of g' as a parent of it. We used this approach to find the hidden members of the eviction set and estimate the attacker's effort required to show the security of PCache.

2) Eviction Distribution Estimation: Another approach that the attacker can adopt is to estimate the evictions of each location in the PCache against each victim's access. For this, the attacker randomly fills whole PCache and then allows the interested victim program to access PCache, which causes eviction of attacker's filled cache lines. The attacker observes these evictions and relates the cache lines having high eviction probability with the interested victim access. The attacker has to access as many times to ensure that all possible evicting cache lines should be selected multiple times for eviction. The number of access required by the attacker indicates the effort required to learn eviction distribution. The number of memory accesses can be modeled as *coupon collector's problem*, which gives the expected number of accesses (n_{access}) needed such that β portion of the evicting members of a replacement tree is evicted. This can be obtained using $E[n_{access}] = -n_{em} \cdot ln(1-\beta)$, where as n_{em} is the number of evicting members in tree. For $32/4$ PCache, which has 2^{12} evicting members per tree, would require $\approx$ 18.86k victim calls to evict $\beta = 99\%$ of the eviction members of tree. This gives the eviction estimation cache lines against one victim address. However, for a successful attack, the attacker needs to know against all vulnerable victim address space. For example, in the case of AES, the attacker has to learn against all cache lines belonging to AES tables, which are 128. Therefore, the attacker needs 2.4 million victim calls to estimate the eviction distribution. Moreover, in case of multiple accesses to PCache, in Section V-B, we show that estimation of eviction distribution become indistinguishable because of all cache lines are selected for eviction and different victim memory accesses evict the same cache lines.

B. Filling and Observing Complexity

Assuming that the attacker has learned the eviction sets, to launch the attack, it needs to place learned memory addresses in the cache at the right cache-way to get evicted on victim access. As there is a random replacement policy, so the number of accesses required to place a memory address in the right way cannot be done in one access but multiple accesses. The number of accesses required by the attacker to place in an interesting cache way can be viewed as the bin-and-ball problem and can be given using the following equation.

$$n_{access} = \frac{log(1-confidence)}{log(1-p)} \tag{1}$$

Where n_{access} is the number of accesses required by the attacker to fill interested cache way, $confidence$ indicates the probability by which event of filling can be fulfilled, and p indicates the probability with which a memory address can successfully be placed at the right location in PCache. p defines the worst case and best case for the attacker, it can vary form $1/w$ (worst case) to 1 (best case), where w is the number of ways. Worst case is case when attacker selects a memory address that is a part of eviction set only if it is placed at one specific cache way. Inversely, the best case is the case when attacker selects a memory address that is a part of eviction set irrelevant to cache ways. As attacker has no information about that the memory address belongs to best or worst case, practically attacker has to assume worst case for every address to launch attack successfully with 99% confidence.

Depending on n_{access}, the attacker has to fill all lines of PCache that are involved in the replacement process, which is shown in replacement candidate tree in Figure 2. We will use the word tree to refer to the PCache in following discussion for simplicity. To fill first level of tree, attacker has to access $n_{access} \cdot w$. This number of accesses will guarantee the filling of first level of tree but which memory accesses are placed at the right cache ways is unknown to the attacker. Therefore, on next level attacker has to fill child of each memory address accessed for filling of first level of tree. This exponentially raises the number of memory accesses required on each level. Total number of accesses for filling of cache belongs to one replacement candidate tree can be given by

$$T[n_{access}] = \sum_{i=0}^{l} n_{access}^{i} \cdot w^{i} \tag{2}$$

$T[n_{access}]$ is the total number of accesses required by the attacker to attack, whose value varies depending on n_{access}. The attacker has to consider the worst-case to derive n_{access} for a successful launch of eviction-based cache-based SCAs but $T[n_{access}]$ becomes greater than cache capacity and limits the attacker from filling the PCache with 99% confidence. Figure 3 shows the maximum confidence level with which the filling of PCache can be fulfilled in $32/4$ PCache. This shows that the attacker can fill PCache with maximum of 17%, 26% and 28% confidence level for 1MB, 8MB and 10MB, respectively. Attacker can use n_{access} at maximum of 1 for 1MB and 2 for 8MB and 10MB PCache.

V. Security Evaluation

Security of PCache is based on the fact that members of eviction sets greatly increase the effort of attacker. In this section, we evaluated effort required by the attacker to find evicting members using Prime-Prune-Probe method and hidden members using Exclude-Prime-Probe method, discussed in Sections II-D and IV-A1, and using Eviction Distribution Estimation, discussed in Section IV-A2.

 ISBN: 978-1-61208-818-1

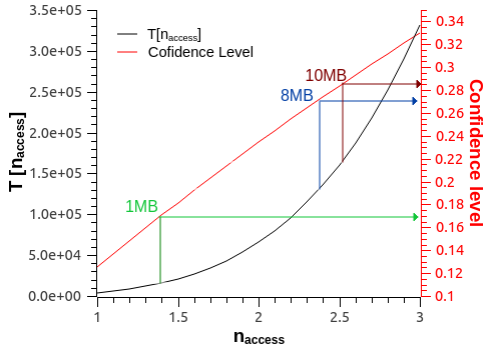

Figure 3. Confidence level required for filling of 32/4 PCache

A. Evaluation Using Prime-Prune-Probe and Exclude-Prime-Probe Method

We have made a model of PCache using Python. In experiment, we have taken the following assumptions. First, we have assumed noise-free model that the attacker and victim are running only. Second, we have filled the cache with attacker addresses using random function. Third, we have randomly selected the permutation function of each level. Lastly, we have evaluated the attacker effort on 1MB, 8MB and 10MB cache with 4 and 8 groups of ways.

We have measured the turns required to reveal 1000 hidden members. Then, we have multiplied the number of total addresses required by attacker with the average turns measured using experiment. For time calculation, same setting is used as in research work [15], which compromised security of ScatterCache, that is, flush time 0.5ms, victim execution time 3ms, cache hit time 9.5ns and cache miss time 50ns.

TABLE I. TIME REQUIRED TO LEARN EVICTION SET IN 32/4 PCACHE.

Capacity (MB)	n_{ve} (k)	T_E (hours)	n_{vh} (k)	T_H (hours)
1	301	0.39	113.03	613.6
8	411	1.04	919.52	12605.8
10	491	1.17	1150.68	18497.1

Results of experiment is shown in Table I. In this table n_{ve} and n_{vm} indicate the number of victim call against one evicting and hidden member respectively, and T_E and T_H indicate the time required to find evicting and hidden members,respectively, in eviction set for $n_{access} = 2$. Results show that time required by the attacker to find hidden members of one replacement candidate tree becomes difficult as cache capacity is increased or the number of ways are increased in a group. The attacker would need $\approx$ 25 months (or 2 years) to learn eviction set against one memory address in 10MB cache with 32 ways and 4 groups. Even after learning the whole eviction set attacker can launch attack with only 28% confidence. As permutation is tag dependent, the attacker has to again find the hidden members once the physical mapping is changed by operating system. Moreover, we have assumed fixed permutation mapping, security can be improved by making them configurable permutation functions so that these functions should be changed once before time given in Table I for specific cache configurations.

B. Evaluation Using Eviction Distribution Estimation

To estimate eviction distribution, we have executed experiment as discussed in Section IV-A2 and extracted number of evictions per each cache line in 8MB cache with 4 groups per way by accessing memory accesses 18.86k times. Result in Figure 4 shows the number of evictions per each cache line against single memory access. This result shows that the number of evictions per cache lines against one memory addresses may be distinguishable.

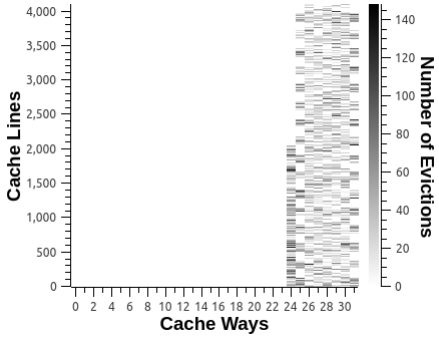

Figure 4. Number of Evictions per cache lines as a result of 18.86k accesses using one victim memory address.

As evictions occur from the last group of PCache, so probability of evicting same cache lines by different memory addresses is high. This increases the difficulty for an attacker if multiple memories are accessed by victim, which is practical assumption that there are always multiple memory accesses by program. We have extracted the eviction per cache lines using 100 memory addresses, which is shown in Figure 6. This result shows that each cache line is selected at-least-once for eviction and makes it impractical for an attacker to distinguish memory addresses from Eviction Distribution Estimation.

VI. PERFORMANCE EVALUATION

We have performed microarchitectural simulation using trace-based simulator ChampSim. Table II shows the configuration used in our study. The L3 cache is 8MB shared between 2 cores. For PCache, we have taken Permutation function latency of 1 cycle.

We have used 20 SPEC CPU2017 benchmarks as workloads for performance evaluation. For each benchmark, we have used a representative slice of 1 billion instructions and built 19 groups of workload where each group contains 2 benchmarks. We have performed simulation until all workloads in group finish executing 1 billion instructions. For measuring aggregate performance, we have measured the weighted speedup metric of proposed cache and normalized it to the baseline.

TABLE II. BASELINE CONFIGURATION

Component	Specification
Core	2 cores
L1 cache	Private, 32kB, 8-way set-associative, split D/I
L2 cache	Private, 256kB, 8-way set-associative
L3 cache	Shared, 8MB, 32-way set-associative or 32/4 PCache

Figure 5 shows the performance of 32/4 PCache with random replacement policy. As performance is normalized to the baseline, so bar higher than 100% is better. PCache with LRU is competitive to set-associative cache with LRU policy (or baseline) on most of the workloads. Moreover, results in Figure 5 show that PCache also outperforms the baseline

 ISBN: 978-1-61208-818-1

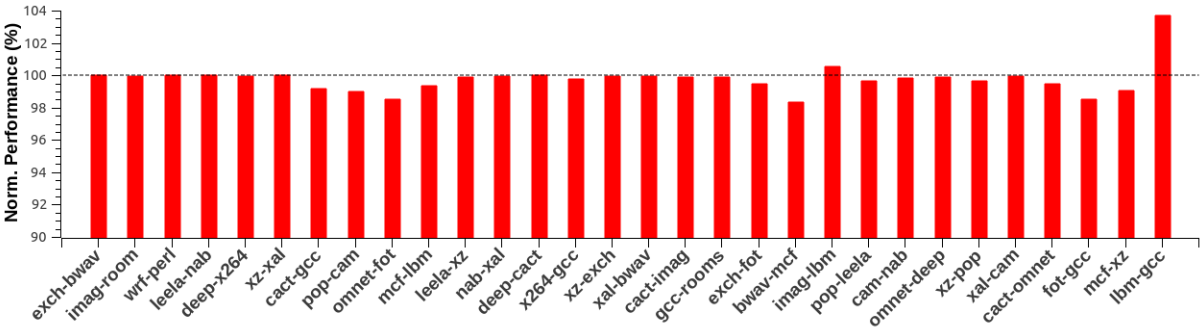

Figure 5. Normalized Performance of 8MB 32/4 PCache over SPEC CPU2017 workloads.

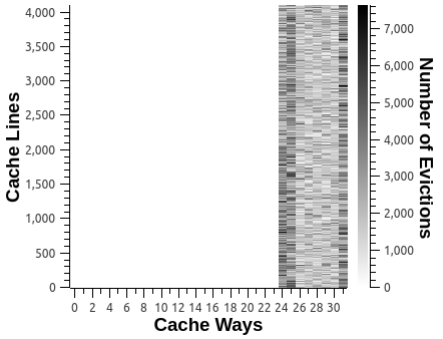

Figure 6. Number of Evictions per cache lines as a result of 18.86k accesses using 100 sequenced victim memory addresses.

of about 3% on $lbm - gcc$ because of conflict misses are reduced. However, PCache with random replacement policy shows degradation as compared to baseline on some workloads but a maximum of 1.6%, and performance loss is between 1.4% to 1.6%. Overall, the performance loss is only 0.002% on average as compared to the set-associative cache over SPEC CPU2017.

VII. Conclusion and Future Work

We have presented PCache, a cache design that provides security against eviction-based cache-based SCAs by making large eviction sets and introducing hidden members in the replacement process. PCache divides the cache into multiple groups and relocates a cache-line to multiple groups before eviction. In each group, relocated line passes through different permutation functions, resulting in difficulty for an attacker to find these relocated lines (or hidden members) in practically feasible time. Our evaluation shows that, for 10MB cache, the attacker needs 2 years to learn eviction set against one memory address. Moreover, a large eviction set and random replacement policy limit the attacker to launch eviction-based cache-based SCAs with only 28% confidence. Along with strong security, PCache has low-performance overhead of about 1.6% at maximum as compared to set-associative cache with LRU policy. While we have analyzed PCache as last-level-cache, this idea can also be extended on other shared structures like Translation Lookaside Buffers, which are also vulnerable to SCAs.

References

[1] 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, May 2015, pp. 605–622.

[2] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "Armageddon: Cache attacks on mobile devices," in 25th USENIX Security Symposium, 2016, pp. 549–564.

[3] 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.

[4] D. Gruss, C. Maurice, K. Wagner, and S. 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, 2016, pp. 279–299.

[5] D. Wang et al., "Unveiling your keystrokes: A cache-based side-channel attack on graphics libraries." in NDSS, 2019.

[6] P. Kocher et al., "Spectre Attacks: Exploiting speculative execution," in 40th IEEE Symposium on Security and Privacy (S&P'19), 2019, pp. 1–19.

[7] M. Lipp et al., "Meltdown: Reading kernel memory from user space," in 27th USENIX Security Symposium (USENIX Security 18), 2018, pp. 973–990.

[8] M. A. Mukhtar, M. K. Bhatti, and G. Gogniat, "Architectures for Security: A comparative analysis of hardware security features in Intel SGX and ARM TrustZone," in 2019 2nd International Conference on Communication, Computing and Digital systems (C-CODE), 2019, pp. 299–304.

[9] T. Kim, M. Peinado, and G. 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), 2012, pp. 189–204.

[10] 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), Oct 2018, pp. 974–987.

[11] F. Liu et al., "CATalyst: Defeating last-level cache side channel attacks in cloud computing," in 2016 HPCA, March 2016, pp. 406–418.

[12] 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, ser. CSAW '08. ACM, 2008, pp. 25–34.

[13] M. Werner et al., "ScatterCache: Thwarting cache attacks via cache set randomization," in 28th USENIX Security Symposium (USENIX Security 19), 2019, pp. 675–692.

[14] M. K. Qureshi, "CEASER: Mitigating conflict-based cache attacks via encrypted-address and remapping," in 51st IEEE MICRO, 2018, pp. 775–787.

[15] A. Purnal and I. Verbauwhede, "Advanced profiling for probabilistic Prime+Probe attacks and covert channels in ScatterCache," ArXiv, vol. abs/1908.03383, 2019.

[16] R. Bodduna et al., "Brutus: Refuting the security claims of the cache timing randomization countermeasure proposed in CEASER," IEEE Computer Architecture Letters, vol. 19, no. 1, 2020, pp. 9–12.

[17] 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, Sep. 2016, pp. 8–16.