EA-PLRU: Enclave-Aware Cache Replacement

Atsuko Shimizu
Binghamton University
ashimiz1@binghamton.edu

Daniel Townley
Binghamton University
dtownle1@binghamton.edu

Mohit Joshi
Binghamton University
mjoshi7@binghamton.edu

Dmitry Ponomarev
Binghamton University
dima@cs.binghamton.edu

ABSTRACT

In SGX-based systems, cache lines belonging to enclaves must be encrypted when they are evicted from the last-level cache, and decrypted on a cache miss before they are brought into the cache from memory. Because encryption and decryption introduce overheads in terms of performance, memory pressure and power consumption, it is important to reduce the frequency of LLC misses and replacements of enclave lines. We consider a system where enclave and non-enclave applications co-exist in the system and share the last-level cache. To decrease the frequency of encrypt/decrypt operations, we introduce a new cache replacement policy that slightly favors enclave lines over non-enclave lines. Specifically, we modify the last level of pseudo-LRU replacement logic, so that it favors an enclave line over a non-enclave line regardless of how recently each line has been accessed. We also add a probabilistic component to this new policy to balance performance and make replacement policy non-deterministic and therefore resilient to side-channel attacks that exploit predictable patterns of cache line replacement.

CCS CONCEPTS

• **Computer systems organization** → **Processors and memory architectures**; • **Security and privacy** → *Security in hardware*.

KEYWORDS

Cache Replacement Policies, Secure Enclaves, Intel SGX, Pseudo-LRU

ACM Reference Format:
Atsuko Shimizu, Daniel Townley, Mohit Joshi, and Dmitry Ponomarev. 2019. EA-PLRU: Enclave-Aware Cache Replacement. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy (HASP '19), June 23, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3337167.3337172

HASP '19, June 23, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-7226-8/19/06…$15.00
https://doi.org/10.1145/3337167.3337172

1 INTRODUCTION

Isolated execution systems have recently emerged as a promising solution for protecting secrets from potentially compromised or malicious system software layers. Modern operating systems and hypervisors contain millions of lines of code and multiple exploitable vulnerabilities are discovered in them every year. Even a single exploit in the system software can allow the attackers to easily obtain privilege escalation and render many defenses ineffective. Rather than trusting vulnerable system software to provide protection, isolated execution systems introduce specialized hardware to ensure the privacy and integrity of critical user data [1, 7, 9, 19].

Intel Software Guard Extensions (SGX) [1, 9, 19] is the most prominent industry implementation of isolated execution paradigm. SGX allows sensitive parts of application code to run in secure enclaves that are inaccessible to operating systems, hypervisors, and other software modules. Dedicated SGX hardware performs memory permission checks and only allows accesses to enclave memory pages from inside enclave code. In addition, while the enclave data resides in the clear in the caches, it is encrypted and integrity checked when it leaves the on-chip caches, and is stored in memory in encrypted form. On a cache miss, the enclave data is decrypted and its integrity is verified before the data is brought back into the cache.

Applications that use SGX support can co-exist in the system with regular applications that do not require additional security. In a multi-core or a many-core system, many applications belonging to these two classes will share the last-level cache (LLC). As a result, the LLC houses memory lines that have variable access latencies to bring them into the cache: the lines belonging to enclaves have longer latencies due to the need to decrypt them on the way to the LLC, while the lines belonging to regular applications have lower latencies. According to our experiments on a Intel Skylake processor equipped with SGX, there is about a 100-cycle difference between the cache miss latency of an enclave line and a non-enclave line.

Traditional cache replacement policies are not designed to take this disparity in the cache miss costs into account, and instead base the replacement decisions mostly on the recency information. For example, least-recently-used (LRU) replacement selects (within a selected cache set) a cache line that has not been accessed for the longest duration in the past. As true LRU policy is fairly complex to implement in hardware for higher-associativity caches, commercial implementations often rely on a simpler pseudo-LRU (PLRU) policy [4, 8]. PLRU divides the cache ways into two halves and marks each half as either a cold or a hot half. The hot half is the one that contains the way where the most recent access within the set had

occurred. Recursively, each half is further divided into equal halves and hot/cold statistics are kept at those levels. The victim is selected by following the cold path. This replacement scheme is similar in performance to true LRU, but is much simpler to implement, as it does not require the total ordering of cache ways in terms of their access recency.

To account for different costs and latencies of cache misses in an SGX system, we propose Enclave-Aware PLRU (EA-PLRU) — a simple modification to PLRU that slightly favors enclave lines over non-enclave lines in terms of making replacement decisions within PLRU policy. Specifically, we propose modifying the last level of PLRU selection tree, so that if the choice is between an enclave line and a non-enclave line, the non-enclave line is selected as a victim, leaving the enclave line in the cache and potentially avoiding its costly replacement and future cache miss. We conservatively assume that the costs of encryption upon eviction of an enclave line can be hidden (encryption is performed in the background on the way to memory) and do not impact the number of execution cycles. The only additional penalty that we target arises when an LLC miss to an enclave line occurs - this requires the decryption operation before the data can be brought into the cache.

Despite these conservative assumptions, we demonstrate that EA-PLRU effectively redistributes the cache misses slightly in favor of enclave applications, but without impacting the overall number of cache misses. While the performance of non-enclave applications somewhat degrades, this is more than compensated for by the reduction of LLC misses for enclave applications, which experience a higher miss cost. To better control this trade-off and avoid starvation of non-enclave applications, we also add a probabilistic component to EA-PLRU, where the enclave-favoring policy at the last level of the PLRU tree is used with a given probability; otherwise, a normal recency-based PLRU decision is made. We show that probabilistic EA-PLRU performs even better than deterministic EA-PLRU. The probabilistic policy has an additional benefit of thwarting recently introduced side-channel attacks that are based purely on the knowledge of the cache replacement policy [3]. The non-determinism at the last level of the PLRU tree makes it very difficult, if not impossible for the attacker to reverse-engineer the cache replacement algorithm and mount an attack.

To demonstrate the potential of EA-PLRU, we evaluated this new policy on an in-house trace-driven simulator of a multi-level cache hierarchy. The memory access traces were obtained from Intel's Pin tool [12]. While in this paper we only present the results for the experiments where the entire application is executed inside an enclave, our framework also supports scenarios where only the critical pieces of application code are executed in an enclave. We also use measurements from a real Intel Skylake processor equipped with SGX capabilities to determine the costs of LLC misses for enclave and non-enclave lines, and use these measurements to create an analytical performance model to estimate the impact of EA-PLRU on the number of cycles required by our simulated applications. The key benefits of the proposed EA-PLRU policy are the following:

- EA-PLRU redistributes the LLC misses between enclave and non-enclave applications in a controlled manner to achieve a performance improvement of about 3% on the average according to our analytical model.
- The probabilistic version of EA-PLRU adds non-determinism to replacement decisions, making it difficult to reverse-engineer replacement policies and perform side-channel attacks based on knowledge of cache replacement policies.
- EA-PLRU adds a storage overhead of less than 0.2% to the LLC (assuming an LLC with 64-byte lines), as only one bit is required per cache line to indicate if a line belongs to an enclave or not.
- EA-PLRU requires only a minimal change to the existing PLRU policy and can be easily integrated into current designs.

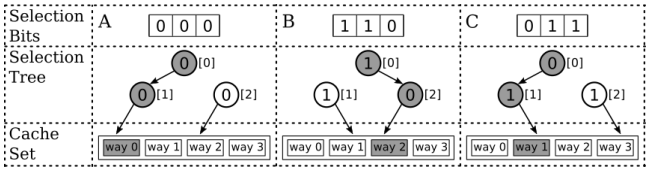

Figure 1: Operation of the Pseudo-LRU replacement policy

2 BACKGROUND

In this section, we provide some background on PLRU replacement policy, and also briefly describe SGX architecture and its performance overheads.

2.1 Pseudo-LRU cache replacement

Our implementation and evaluation of EA-PLRU uses the widely-adopted binary tree-based pseudo-LRU (PLRU) policy as a baseline cache replacement strategy. PLRU replacement policy approximates the *Least Recently Used* (LRU) strategy for set-associative caches. Given a cache set whose lines are fully occupied, LRU replaces the line whose contents were accessed least recently. In general, cached data has a high degree of *temporal locality*, meaning that the most recently accessed lines are the most likely to be accessed in the future. The LRU strategy exploits this property by retaining the cache lines most likely to be accessed, thus reducing unnecessary evictions and improving overall cache performance. While the hardware needed to implement true LRU is prohibitively complex for larger associativity caches, PLRU replacement logic is architecturally simple, and is thus frequently deployed in commercial cache architectures [4, 11].

Figure 1 shows the widely-used binary tree implementation of PLRU. Rather than maintaining total ordering of cache lines in a set according to their recency, PLRU implements a tree whose nodes point, with increasingly fine granularity, *away* from the portion of the cache set where the most recent access occurred. This tree structure is encoded as a series of bits that are stored for each cache set and updated whenever the set is accessed. When a replacement occurs, PLRU uses the path defined by the nodes to select a cache line for replacement, resulting in replacement decisions that reproduce or closely approximate the operation of true LRU.

For example, Figure 1-A shows the PLRU logic for a simple four-way cache. Bit 0, representing the root of the tree, initially points toward the portion of the set containing lines 0 and 1, while bits 1 and 2, which are leaves in the tree, point to cache lines 0 and 2, respectively. Thus, when a new access occurs, the tree selects cache line 0 for replacement. Since line 0 is now the most recently accessed node, PLRU inverts all the bits along the selection path to point away from line 0. The resulting configuration is shown in Figure 1-B. The next time the line is accessed, PLRU will follow the selection bits to line 2, and update the selection bits as shown in Figure 1-C. Note that this final configuration designates line 1 as the next line to be replaced, preserving the more recently-accessed contents of line 0.

2.2 Intel SGX

Intel Software Guard Extensions (SGX) [2, 5, 10, 18] protects sections of critical code by isolating them in secure enclaves. SGX uses dedicated hardware to protect enclave memory from manipulation by software running outside the enclave, including system software that would otherwise have unrestricted access. In this way, SGX protects critical user data from tampering by an untrusted OS, limiting the trusted computing base to the CPU hardware with its associated SGX components.

Addtitionally, SGX protects against physical attacks on the memory and interconnect by encrypting data as it leaves the on-chip cache hierarchy. This operation is performed by the SGX Memory Encryption Engine (MEE), which also decrypts and integrity-checks enclave data as it is read into the cache. While the encryption of outgoing data could be performed in parallel to other computations, the decryption of incoming data imposes a measurable delay in addition to existing cache-miss penalties, making last-level cache evictions for enclaves significantly more expensive than those for regular programs. While these variations in eviction cost can have significant implications for cache performance, current cache replacement policies make no distinction between enclave and non-enclave memory. EA-PLRU introduces a replacement policy that takes into account the heightened cost of moving enclave memory in and out of the caching hierarchy, thus improve performance for mixed workloads that include SGX enclaves.

3 EA-PLRU DESIGN

In this section, we present the implementation details of the new replacement policy.

3.1 Enclave-Aware Probabilistic Cache Replacement

EA-PLRU limits enclave encryption overheads by increasing the retention of enclave memory lines in the cache hierarchy. This is accomplished by preferentially evicting non-enclave memory during cache replacement, effectively substituting (comparatively) inexpensive non-enclave memory evictions for enclave memory evictions that could lead to additional cryptographic overhead.

EA-PLRU implements this strategy using a simple heuristic that can be easily integrated into existing PLRU cache designs. Specifically, this scheme retains the structure of the PLRU selection tree, and applies the baseline selection logic for all non-leaf nodes, but modifies the selection logic at the leaf nodes to favor enclave-owned cache lines. To distinguish between enclave and non-enclave memory, EA-PLRU logically extends each cache line with a *enclave bit* that is set to one if line belongs to an enclave, and to zero if it is a regular line. If both lines referred to by a leaf selection bit are either enclave or non-enclave, then the least recently used line is evicted, as in the baseline policy. If, however, one of potential victim line belongs to an enclave and the other belongs to a regular application, EA-PLRU selects the non-enclave line for eviction regardless of its recency.

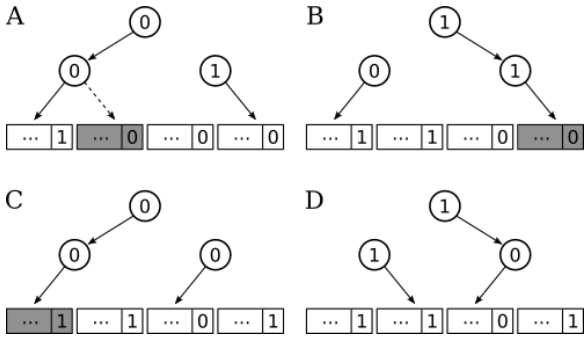

Figure 2: Example of EA-PLRU

Figure 2 shows the operation of EA-PLRU in a simple four-way cache, over three consecutive memory reads by an enclaved process. In Figure 2-A, the higher level selection policy selects the first pair of cache lines for replacement, according to the baseline PLRU policy. Line 0 currently contains data from an enclave and is the least recently used in this pair, while the data in line 1 originated from a non-enclaved application. Because the lines contain different enclave bits, the alternate allocation policy is used, and the non-enclave line in the pair is selected for eviction even though line 0 is less recently used. In this case, the modified update logic forgoes the usual inversion of the leaf node, as line 0 is still the least-recently used. In Figure 2-B, the nodes have been updated to indicate the most recent access to line 1, and the next replacement is thus directed to the pair including lines 2 and 3. Currently, data from a non-enclaved program occupies both of these lines, and the selection bit points to line 3. Since the enclave bits are false for both lines, the default PLRU policy is used, and the incoming line displaces line 3. In Figure 2-C, the replacement bits again point to line 0 as the least recently used. Since both of lines 0 and 1 now contain enclave data, the default policy is again applied, and the enclave evicts its own data according to PLRU. Figure 2-D shows the final state of the selection logic after this operation has completed.

The baseline EA-PLRU replacement strategy achieves the objective of retaining more enclave cache lines in memory, but tends to impose excessive overheads on non-enclaved programs that can lead to overall performance degradation. To balance the caching requirements of enclave and non-enclave programs, EA-PLRU applies the modified strategy only to a designated fraction of replacements, randomly reverting to the default PLRU policy in to provide adequate service to non-enclaved programs. The relative frequencies with which the modified and baseline strategies are used can be adjusted to meet the requirements of various workloads. Although

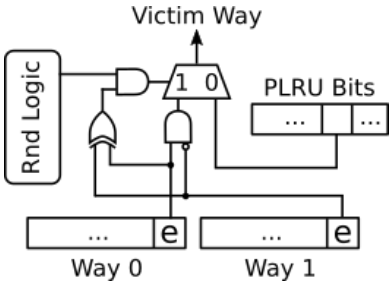

Figure 3: EA-PLRU integration with baseline pLRU hardware

deterministic selection of the replacement strategy would also be possible, randomization greatly simplifies integration with baseline PLRU hardware, and has the added benefit of hindering a certain class of cache side-channel attacks.

3.2 Hardware Implementation

The heuristic strategy used by EA-PLRU greatly simplifies the system's integration into the baseline PLRU logic. As shown in Figure 3, EA-PLRU interposes a selector that chooses the victim specified by the enclave-aware replacement policy, or by the baseline LRU logic. If the enclave bits in the referenced cache lines are different, and a signal from the *randomization logic* is high, the line whose enclave bit is zero is selected as the victim. This selection is accomplished by taking the logical AND of the enclave bit of the first set and the inverted enclave bit of the second way. The output of this operation will indicate the position of the non-enclave set. If the signal from the randomization is low or the enclave bits for the two sets are the same, the line specified by the baseline policy is chosen for eviction.

The randomization logic, shown in Figure 4, is also relatively simple. As a basis for a random signal, a *randomization register* (a) is populated from an entropy source, such that each bit has an equal chance of being 0 or 1. These bits are then combined (b) to generate random signals that are true with varying frequencies. For example, a signal that is true in 1/8 cases can be generated by taking the logical AND of the bits in a three-bit randomization register: the random numbers 111 will produce a randomization signal of 1, while 000-110 will produce a randomization signal of 0, meaning that enclave-aware replacement will be applied to 1/8 (12.5%) of evictions. Other gate configurations can be used to produce additional probabilities. To tune EA-PLRU to meet the

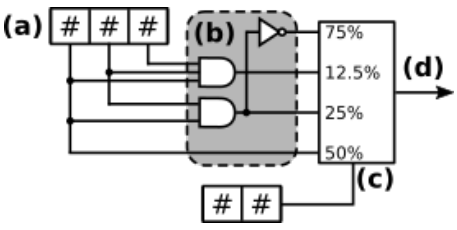

Figure 4: EA-PLRU randomization logic

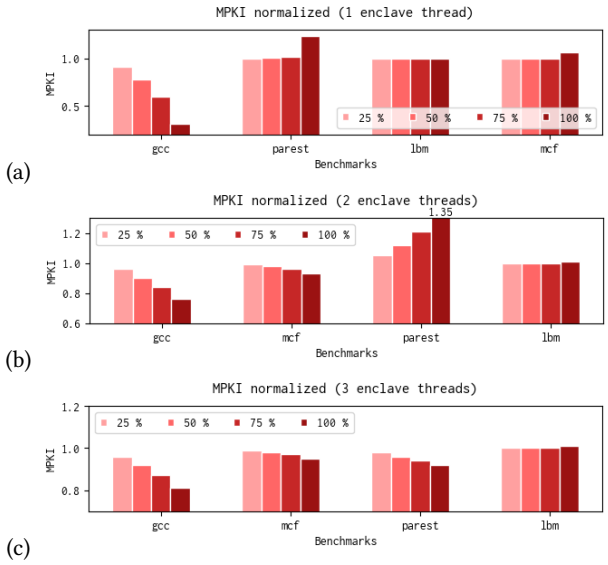

Figure 5: MPKI of workloads with 4 threads.

requirements of specific workloads, a multiplexer (c) controlled by a configuration register can select one of several random signals generated from the randomization register. The resulting signal (d) serves as the randomization input for the EA-PLRU selection logic.

The impact of EA-PLRU in terms of added hardware complexity is minimal. For a typical LLC with 64 byte lines, such as the LLC in current Intel designs, the storage overhead for the enclave bits is less than 0.2%. The modified selection logic entails a small increase in logical depth relative to the baseline replacement policy, and delays incurred due to this extension, if any, will be almost entirely masked by LLC fetch latencies, which typically exceed 100 cycles. Additionally, the randomization logic itself can be pipelined to pre-compute randomization bits, completely removing it from the critical path of the replacement logic.

4 EVALUATION METHODOLOGY AND RESULTS

In this section, we first describe how we estimated the additional cost of SGX decryption operation on an LLC miss. Then we show the results of our performance evaluation that assess the cache performance in isolation. We estimate the overall impact on the execution time by incorporating the additional SGX-related costs into a simple performance model. While these results are preliminary and the model is simple, the results demonstrate the performance potential of our new replacement policy.

4.1 Estimating Cache Miss Costs in SGX

Cache misses for an enclave have an additional overhead when compared to non-enclave misses. In general, the performance overheads of SGX can result from two main sources. First is the overhead added due to the encryption and decryption from the Memory Encryption Engine. The second is the overhead associated with entering and exiting an enclave. In addition, extra security measures such

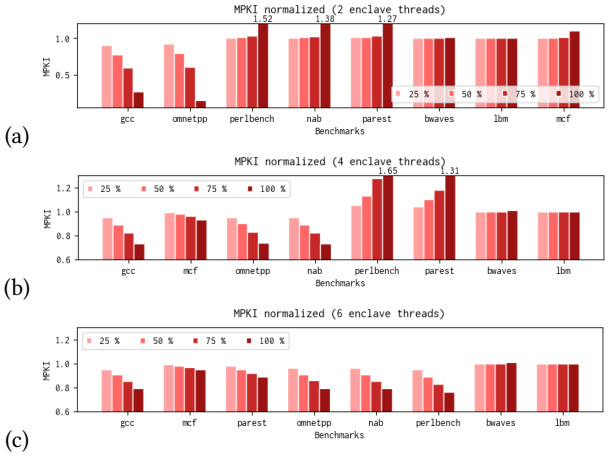

Figure 6: MPKI of workloads with 8 threads.

as integrity tests and memory usage limitations may also affect performance.

We evaluated the memory encryption overhead of cache misses by creating a micro-benchmark consisting of 2 million load instructions. Experiments were performed on a Dell Desktop machine with 8-core Intel i7-7700 (SkyLake) CPU at 3.60GHz, using 7.5GB RAM and running on an Ubuntu 16.04 operating system. We used Intel SGX Linux 2.3 SDK libraries to create enclave programs.

Our experiment calculates the difference between the number of cycles incurred while serving a cache hit and a cache miss for a similar environment. In order to calculate the number of CPU cycles required by this microbenchmark, we used the serializing `CPUID` instruction along with `RDTSC` instruction to start the timestamp counter, and execute `RDTSCP` instruction to stop timestamp counter, only when all instruction before it complete, followed by `CPUID` instruction. The difference between the counter readings provides the number of cycles executed by this code; this is a standard practice that is used to measure fine-grain timing [20]. In this section, we will describe our process while referring to Listing 1.

```
1  // Enclave with Cache Miss/Hit Simulation
2  ecall (eid, M1)      // Entering enclave
3  asm(clflush(M1))  // Added when testing miss
       latencies
4  ocall()              // Exiting enclave
5  asm(CPUID, RDTSC)// Record Start time
6  ocall_return         // Entering enclave
7  Load M1[0]           //
8  ecall_return         // Exiting enclave
9  asm(RDTSCP, CPUID)    // Record end time
10
11 // Non Enclave with Cache Miss/Hit Simulation
12 asm(clflush(M1)) // Added when testing miss
       latencies
13 asm(CPUID, RDTSC)
14 Load M1[0]
15 asm(RDTSCP, CPUID)
```

Listing 1: Pseudocode for measuring cache hit and miss costs (in CPU cycles) of enclaves and non-enclaves.

For enclaves, we begin our experiment making an `ECALL`, copying the array to enclave with the enclave id `eid` and switching to the enclave (line 2). When running a cache miss simulation, a `clflush` instruction is executed to flush the cache line (line 3). For a hit simulation, line 3 is omitted, and execution proceeds directly to the `OCALL` in line 4 from the enclave to non enclave mode. An `OCALL` is needed to start recording the timestamp, as the `RDTSC` and `CPUID` in line 5 are privileged instructions which cannot be executed inside an enclave. After starting the timer we return back to the enclave from non enclave mode (line 6), we perform a read operation for the first array element (line 7) and then return from the `ECALL`, exiting the enclave (line 8). Once we are out of enclave, the timer is stopped immediately using the `RDTSCP` followed by `CPUID` instructions (line 9). We conducted this same operation of reading the data while experiencing a cache hit and a cache miss for an enclave. A similar experiment was conducted to benchmark non-enclave execution (lines 12-15), while keeping all other conditions the same. We iterated these instructions 2 million times in each simulation to establish the average number of cycles for cache hit/miss scenario.

We determined the miss penalties for enclave and non-enclave execution by taking the difference in the cycles incurred in their respective hit and miss simulations. For the enclave simulation, this operation nullifies the `ECALL` and `OCALL` overheads. We established the cache miss costs specific to SGX memory encryption by taking the difference the enclave and non-enclave miss penalties. Through this evaluation, we found that the mean LLC miss penalties for non-enclave and enclave programs were 246 and 357 cycles, respectively. SGX encryption thus added an average of 111 cycles to the miss penalty, an increase of 1.45X relative to non-enclave latencies.

4.2 Experimental Setup

To evaluate the performance of EA-PLRU, we used Intel Pin [12] to collect traces of memory references of the SPEC CPU 2017 benchmarks. These memory traces are then fed into our trace-based cache simulator, which simulated EA-PLRU for 1 billion memory references. To represent various workloads on an SGX system, our simulator designated benchmarks as enclave or non-enclave programs, and applied the EA-PLRU replacement accordingly.

We mix randomly selected SPEC benchmarks and designate different numbers of enclave programs in each workload, while varying the total number of threads in a workload and changing the probability of enclave-aware replacement.

We simulated a three-level cache hierarchy which remained consistent across all experiments. The L1 cache was split into an instruction and data cache, each 32KB in size. The L2 cache was 256 KB and the L3 cache was 8MB. The L1 and L2 were private to each core, while the L3 was shared across all cores. The L3 was inclusive of the private caches. To calculate the overall cycle penalty due to cache misses, we used the mean miss penalties for enclave and non-enclave memory measured in Section 4.1.

4.3 Performance Results and Discussions

We first examine the MPKIs (Misses per Kilo-Instructions) of enclave and non-enclave benchmarks in the workloads, then we discuss the overall performance of each workload. We refer to the enclave-aware replacement probability as EP, where $EP = x$ means that the cache was configured such that there is a $x\%$ probability

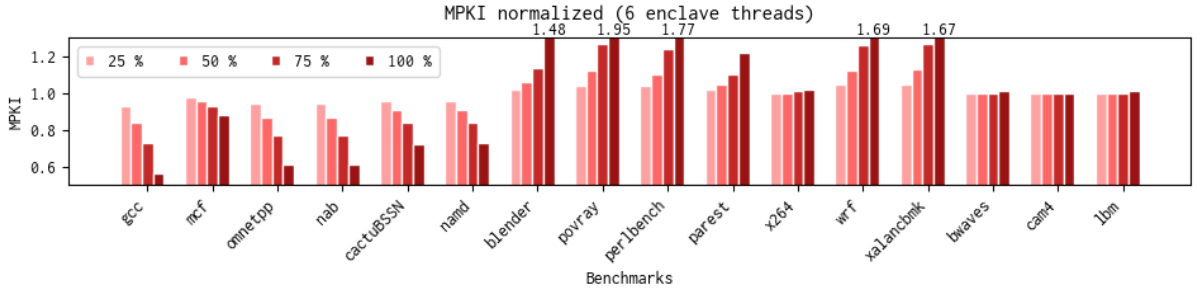

Figure 7: MPKI of workloads with 16 threads, where 6 of the threads are enclaves.

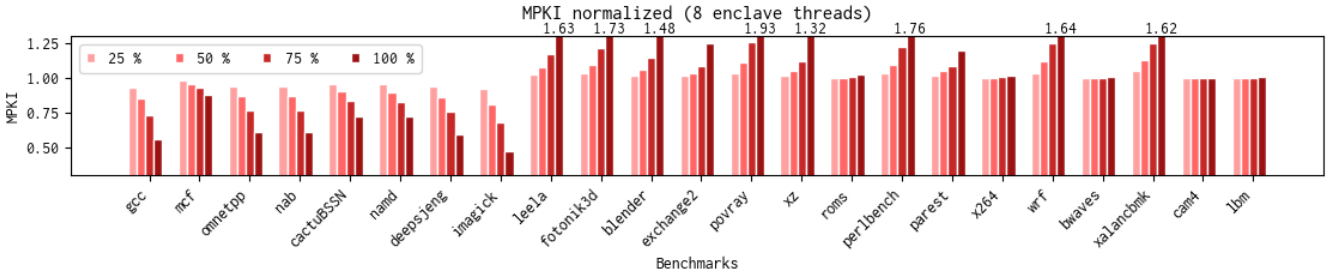

Figure 8: MPKI of workloads with 23 threads, where 8 of the threads are enclaves.

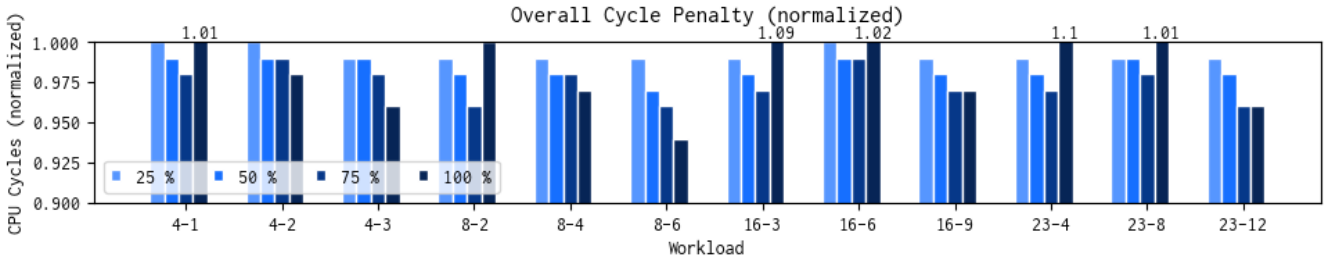

Figure 9: Overall cycle penalty across different workloads.

of using the EA-PLRU policy for cache line replacement instead of PLRU.

Figure 5 shows the MPKIs of the smallest workload of 4 threads. Figure 5(a) shows the results for the experiment when *gcc* runs as an enclave, while the remaining threads run as non-enclaves. In the extreme case of $EP = 100$, *gcc*'s MPKI decreases by 69%, while *parest*, *lbm*, and *mcf* suffer an MPKI increase of 24%, 0.4%, and 6%, respectively. The performance overhead is negligible up to $EP = 75$, where the non-enclave programs experience up to 3% overhead. Figure 5(b) shows *gcc* and *mcf* as the two enclave programs running. At $EP = 100$, *gcc* and *mcf* experience a decrease in MPKI of 24% and 7%, respectively, while *parest* and *lbm* suffer an increase of 35% and 0.6%, respectively. Figure 5(c) shows three enclave programs with *lbm* as the single non-enclave program. *gcc*, *mcf*, and *parest* have a performance improvement of 19%, 5%, and 9%, respectively, while *lbm* only experiences a 0.6% overhead. As expected, as there are more enclave programs, each enclave program experiences a redistributed amount of performance gain. For example, *gcc* initially had a 69% performance gain at $EP = 100$ when it was the only enclave program, but with three enclave programs running alongside, *gcc*'s performance gain reduced to 19%.

Figure 6 shows the MPKIs of workloads with 8 threads. Figure 6(a) has *gcc* and *omnetpp* as enclave programs. In the extreme case of $EP = 100$, *gcc* and *omnetpp* improves performance by 73% and 85%, which however, degrades the performance of the remaining non-enclave programs up to 52%. At $EP = 75$, the performance overhead of the non-enclave programs are negligible, with an MPKI increase of up to 3% while enclaves experience up to a 40% MPKI decrease. Figure 6(b) adds *mcf* and *nab* as enclave programs. $EP = 75$ was negligible in previous runs but for the workload in Figure 6(b), the non-enclave programs, *perlbench* and *parest*, suffer up to 28% and 18%, respectively. At $EP = 50$, *perlbench* and *parest* experience up to 13% overhead, while the non-enclave programs experience

up to a 11% performance improvement. Figure 6(c) has 6 enclave programs running, where at $EP = 50$, the performance overhead of non-enclave programs go up to 1% and the enclave programs obtain up to a 24% performance improvement. Again we see a redistribution of performance gain as more enclave programs are run. With $EP = 50$, average performance improvement start at 22% with 2 enclaves and go down to 8% with 6 enclaves. $EP = 50$ experiences a good performance trade off between the enclave and non-enclaves in Figure 6(c).

Figure 7 shows a workload with 16 threads, where 6 of those threads are enclaves. We also ran 16-threaded workloads with 3 and 9 enclave threads, whose overall performance is shown in Figure 9. The non-enclave programs experience overheads between 5% at $EP = 25$ and 95% at $EP = 100$. Even at $EP = 25$, enclaves still gain a performance boost of up to 7%.

Figure 8 shows a workload of 23 threads, where 8 of the threads are enclaves. Workloads of 23 threads with 4 and 12 enclave threads were also run, but for the sake of space, their overall performance results are shown in Figure 9. Between $EP = 25$ to $EP = 100$, the non-enclave performance degrades between 5% to 93%, while enclave performance improves between 8% to 53%. Like the 16-threaded workload, $EP = 25$ is a good trade off between enclave and non-enclave performance.

Finally, Figure 9 shows the overall cycle penalty across increasing values of *EP*. As seen consistently in the previous results, enclave performance improves with higher values of *EP* while non-enclave performance degrades. The overall cycle penalty consists of the total number of non-enclave and enclave cache misses of each workload, multiplied by their miss costs. Each type of miss (non-enclave and enclave) has a different cycle penalty. From our studies described in Section 4.1, on average, each non-enclave cache miss costs 246 cycles and each enclave cache miss costs 357 cycles. We used these average cycle penalties in our simulations. The overall cycle penalties are normalized to only using PLRU (which is equivalent to $EP = 0$). The overall cycle penalty shows the effect of the inverse relation of enclave and non-enclave performance changes caused by EA-PLRU. The workload names in Figure 9 follow the naming convention `x-y`, which means that the workload consisted of a total of `x` threads, where `y` of those threads are enclave threads. For example, the `23-4` workload in Figure 9 consisted of 23 threads total, where 4 of the 23 threads were enclaves.

As shown in Figure 9, in the extreme case of $EP = 100$, the `23-4` workload experienced the highest overall performance penalty of 10%. Across all workloads, the overall performance at $EP = 75$ improves between 1-4%. At $EP = 50$, we see a performance improvement of up to 3%, and at $EP = 25$, performance improves up to 1%. As shown in the previous results, non-enclave programs suffer negligible performance overhead with *EP* values at 25 and 50. For *EP* of 75 or less, there is no overall performance degradation. In general, the extreme value of $EP = 100$ hurts overall performance when there are significantly less enclave threads over non-enclave threads, such as in workloads `4-1`, `16-3`, and `23-4`. Interestingly, in more balanced workloads between non-enclave and enclave programs, $EP = 100$ is beneficial overall, even though non-enclave programs suffer significant overheads. From all of our results, an *EP* value between 25-50 improves performance overall while causing negligible non-enclave performance overheads across all workloads. At $EP = 50$, non-enclave programs have an average of 7.1% overhead while the overall performance gain is 3%. These results encourage the idea of incorporating the workload characteristics as a metric to dynamically change the enclave-aware replacement probability in future work, which can potentially improve EA-PLRU.

5 SECURITY BENEFITS OF EA-PLRU

In addition to improvements in cache performance, the replacement strategy used by EA-PLRU has beneficial side-effects with regards to security. Specifically, the non-deterministic selection of a replacement strategy in EA-PLRU mitigates cache side-channel attacks that exploit predictability in traditional replacement schemes.

In a typical side-channel attack, an attacker occupies multiple cache lines with its own data, and measuring subsequent access latencies to determine which lines the victim has replaced. Cache side channels are not addressed by the current SGX security model, and thus pose a significant threat to SGX systems. Traditional versions of this attack require the attacker to flood multiple ways in each target set, an aggressive and anomalous access pattern that can be easily detected [14, 15, 21]. However, a recent work exploits the deterministic behavior of existing replacement policies to target the cache ways the victim is expected to use, greatly limiting the detectable side-effects of the attack [3]. By introducing randomness into the replacement policy, EA-PLRU provides an effective mitigation to this type of attack. Although security is not the primary focus of the EA-PLRU design, this side-effect is valuable in light of recent research demonstrating the SGX framework's vulnerability to cache side-channels.

6 FUTURE WORK

While the results presented in this paper demonstrate the potential of EA-PLRU to accelerate SGX caching performance, further research is required to explore further optimizations. One idea is to have the enclave-aware replacement probability to be dynamic depending on the current performance and workloads of enclave and non-enclave programs. As shown in the results, non-enclave programs can suffer high overheads if the enclave-aware probability is too high, and it would be beneficial if the probability can dynamically adjust to these circumstances. Further performance benefits can be realized by limiting the number of ways in each set that can be occupied by non-enclave data, since this data has priority over enclave data.

7 RELATED WORK

Previous work has proposed alternate cache replacement policies that address the general problem of variable-latency memory accesses [13, 24]. These works address differential latencies introduced by the general organization of modern multi-processor architectures [13], or by specific properties like memory-level parallelism [24]. In contrast, EA-PLRU specifically addresses variations in replacement costs arising under SGX. Additionally, EA-PLRU advances this area of research by proposing a novel, probabilistic replacement policy that can be easily integrated into widely-used PLRU logic.

Randomized or probabilistic alternation between eviction policies has been used in previous work to attenuate the effects of aggressive cache replacement schemes [23]. EA-PLRU applies this strategy to a new replacement algorithm optimized for SGX systems.

Prior works have described various randomization-based mitigations to cache side-channel attacks. Random Permutation Cache [25], for instance, introduced an element of randomness into the line replacement policy. More recent proposals [22, 26] unpredictably modify the index used for cache lookups, thus limiting the attackers ability to target sensitive cache sets. Alternative cache side-channel defenses generally involve dividing caches into partitions assigned to specific programs, thus eliminating cache contention entirely [6, 16, 17]. However, these approaches generally rely on partitioning at cache-way granularity, making them difficult to scale, and also make use of system software support, making them vulnerable under the threat model assumed by SGX. Future research is needed to establish how the proposed replacement policy impacts information leakage in the caches in general and weather the additional victimizations of non-enclave lines can somehow be exploited by attackers.

8 CONCLUDING REMARKS

In SGX-supported systems, enclave and non-enclave applications often co-exist with each other and share the last-level cache. As the cache miss costs of enclave and non-enclave programs are different due to the encryption/decryption overhead encountered by enclave misses and replacements, it is important to reconsider classical recency-based cache replacement policies under these new conditions. In this paper, we proposed EA-PLRU, a new cache replacement policy that favors enclave lines over non-enclave lines at the last level of PLRU tree in a controllable manner. We showed that this policy is extremely simple to implement over existing PLRU schemes while incurring very low overhead. EA-PLRU provides performance benefits for enclaves and has a potential to make the LLC more secure against a class of side-channel attacks that exploits cache replacement policy. While our current results are preliminary, they demonstrate that considering non-uniform cache miss costs, in addition to recency, is a viable approach to developing more effective replacement policies for LLCs in SGX-equipped systems.

9 ACKNOWLEDGMENTS

This paper was made possible by NPRP grant 8-1474-2-626 from the Qatar National Research Fund (a member of Qatar Foundation). The statements made herein are solely the responsibility of the authors.

REFERENCES

[1] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. 2013. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, Vol. 13. ACM New York, NY, USA.

[2] Ittai Anati, Shay Gueron, Simon P Johnson, and Vincent R Scarlata. 2013. Innovative Technology for CPU Based Attestation and Sealing. *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy, HASP* (2013).

[3] Samira Briongos, Pedro Malagón, José M Moya, and Thomas Eisenbarth. 2019. RELOAD+ REFRESH: Abusing Cache Replacement Policies to Perform Stealthy Cache Attacks. *arXiv preprint arXiv:1904.06278* (2019).

[4] Wen-tzer Thomas Chen, Peichun Peter Liu, and Kevin C Stelzer. 2006. Implementation of a pseudo-LRU algorithm in a partitioned cache. US Patent 7,069,390.

[5] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. (2016).

[6] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2012. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Transactions on Architecture and Code Optimization (TACO)* 8, 4 (2012), 35.

[7] Dmitry Evtyushkin, Jesse Elwell, Meltem Ozsoy, Dmitry Ponomarev, Nael Abu Ghazaleh, and Ryan Riley. 2014. Iso-x: A flexible architecture for hardware-managed isolated execution. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 190–202.

[8] Jim Handy. 1993. *The cache memory book*. Academic Press, Boston.

[9] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. 2013. Using innovative instructions to create trustworthy software solutions. *HASP@ ISCA* 11 (2013).

[10] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Carlos Rozas, Vinay Phegade, and Juan del Cuvillo. 2013. Using Innovative Instructions to Create Trustworthy Software Solutions. *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy, HASP* (2013).

[11] Kevin A Hurd. 2000. A 600 MHz 64 b PA-RISC microprocessor. In *2000 IEEE International Solid-State Circuits Conference. Digest of Technical Papers (Cat. No. 00CH37056)*. IEEE, 94–95.

[12] Intel. [n. d.]. Pin - A Dynamic Binary Instrumentation Tool. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

[13] Jaeheon Jeong and Michel Dubois. 2003. Cost-sensitive cache replacement algorithms. In *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings*. IEEE, 327–337.

[14] Mehmet Kayaalp, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. 2016. A high-resolution side-channel attack on last-level cache. In *Proceedings of the 53rd Annual Design Automation Conference*. ACM, 72.

[15] Mehmet Kayaalp, Khaled N Khasawneh, Hodjat Asghari Esfeden, Jesse Elwell, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. 2017. RIC: relaxed inclusion caches for mitigating LLC side-channel attacks. In *Design Automation Conference (DAC), 2017 54th ACM/EDAC/IEEE*. IEEE, 1–6.

[16] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. 2018. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. (2018).

[17] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. 2016. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*. IEEE, 406–418.

[18] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday Savagaonkar. 2013. Innovative Instructions and Software Model for Isolated Execution. *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy, HASP* (2013).

[19] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. 2013. Innovative instructions and software model for isolated execution. *HASP@ ISCA* 10 (2013).

[20] Gabriele Paoloni. 2010. How to benchmark code execution times on Intel IA-32 and IA-64 instruction set architectures. *Intel Corporation* (2010), 123.

[21] Mathias Payer. 2016. HexPADS: a platform to detect âĂIJstealthâĂİ attacks. In *International Symposium on Engineering Secure Software and Systems*. Springer, 138–154.

[22] Moinuddin K. Qureshi. 2018. CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping. *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2018), 775–787.

[23] Moinuddin K Qureshi, Aamer Jaleel, Yale N Patt, Simon C Steely, and Joel Emer. 2007. Adaptive insertion policies for high performance caching. *ACM SIGARCH Computer Architecture News* 35, 2 (2007), 381–391.

[24] Moinuddin K Qureshi, Daniel N Lynch, Onur Mutlu, and Yale N Patt. 2006. A case for MLP-aware cache replacement. *ACM SIGARCH Computer Architecture News* 34, 2 (2006), 167–178.

[25] Zhenghong Wang and Ruby B Lee. 2007. New cache designs for thwarting software cache-based side channel attacks. In *ACM SIGARCH Computer Architecture News*, Vol. 35. ACM, 494–505.

[26] Zhenghong Wang and Ruby B Lee. 2008. A novel cache architecture with enhanced performance and security. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 83–93.