For Side-Channel Attack vectors Against the LLC, Dynamically Calculating Minimum Eviction Sets May Be Quicker Than You Thought

Ms. Priyadarshni Samal[1], Dr. Chinmay R. Pattanaik[2]*
*[1]*Assistant Professor,Dept. Of Computer Science and Engineering, NIT , BBSR*
[2]Associate Professor,Dept. Of Computer Science and Engineering, NIT , BBSR
priyadarsini@thenalanda.com, chinmayaranjan@thenalanda.com*

Abstract

Conflict-based cache side-channel attacks aimed at the last-level cache require minimal eviction sets (LLC). Finding rather than computing minimal eviction sets becomes the only option in the most constrained scenario, when attackers have no influence over the mapping from virtual addresses to cache sets. Prior to the recent revelation that it is possible to find minimal eviction sets in linear time, this method was thought to be time-consuming.

The goal of this work is to locate the smallest eviction sets with the smallest latency while also enhancing the current algorithms. Using a perfect cache, a thorough examination of the current techniques has been conducted. Our investigation demonstrates that the maximum latency for locating minimal eviction sets can be further decreased.

the average latency is significantly lowered from O(w2n) to O(wn); Three contemporary processors are used to produce practical experiments. We show that minimal eviction sets can be obtained on all processors, including a most recent Cof- fee Lake one, in a fraction of a second using a number of new strategies described in this research, such as using concurrent multithread execution to avoid the thrashing resistant cache replacement policies. Additionally, it is the first time that it has been demonstrated that it is possible to find minimal eviction sets with entirely random addresses without fixing the page offset bits, which paves the way for a successful attack against fully randomised LLCs in the event that they are ever implemented.

1 Introduction

Cache-based side-channel attacks [31] has become serious security problems in recent years. They have been utilized to recover cryptographic keys [11], bypass the address space lay- out randomization (ASLR) [6], inject faults directly into the DRAM [7], and construct covert channels for attacks against speculative execution [15].

Many of the aforementioned attacks require the adversary to bring specific cache blocks into controlled states. This can be achieved by two types of attacks: *flush-based* attacks and *conflict-based* attacks. Flush-based attacks use explicit cache control instructions, such as clflush on x86 [31], to invalidate target cache blocks. These attacks are accurate but the explicit cache control instructions may require privilege to execute [32] and the target cache blocks must be shared between the adversary and the victim. If either of the condi- tions is not satisfied, an attacker could launch conflict-based cache attacks to achieve similar effect. By accessing a se- quence of carefully chosen addresses, called an *eviction set*, enough cache replacements are triggered so that the target cache block is evicted out of the cache by one of them [22]. Conflict-based attacks can be launched in a sandbox without privilege [21] and, when attacking the last-level cache (LLC), the target cache block can belong to another process or vir- tual machine [33] located on another core [18]. Compared with flush-based attacks, conflict-based attacks are more ver- satile. Conflict-based attacks targeting the LLC are the type of attacks researched by this paper.

An eviction set is a collection of (virtual) addresses that con- tains enough number of addresses mapping to the same cache set containing the target cache block. Accessing this collec- tion in a certain order triggers a cache replacement that evicts the target cache block from the cache set [26]. It is widely known that accessing a large number of addresses is suffi- cient to evict any cache block from any levels of caches [29]. However, accessing extra addresses beyond the necessary can introduce undesirable noise [7] and reduce the speed of attack below the required minimum [5]. According to a study on an ARM Cortex-A53, evicting a cache block using a set of 800 congruent addresses can be 33 times slower and less accurate than using a small set of 21 addresses [16]. Pruning a large eviction set to its minimal is crucial for the success of all targeted and stealth side-channel attacks.

Computing minimal eviction sets typically involves par- tially reversing the mapping from virtual addresses to physi- cal addresses [16]. Reversing this mapping is relatively easy if the mapping is already exposed by the operating system (/proc/self/pagemap in old Andriod systems [16]) or the adversary has already obtained the root privilege (a malicious kernel [8]). If neither is the case, the adversary may infer a part of the physical address by acquiring large chunks of vir- tually and physically contiguous memory, which is normally done by using huge pages [11, 18]. In the most restricted case where the adversary has no control over the mapping from virtual to physical addresses, such as in a sandbox [7, 21], computing minimal eviction sets becomes a challenging task.

The recent development in the cache hardware adds an- other layer of difficulties to the challenge. One of them is the undocumented complex addressing [10, 19] used in Intel processors. The LLC in these processors is sliced and the mapping from physical addresses to slices is determined by a set of undisclosed hash functions. For an adversary targeting the LLC, she needs to decipher the hash functions even when the mapping from virtual to physical addresses is fully re- versed. Consequently, the complex addressing in several Intel processors has been deciphered [13, 19]. A set of undisclosed but static hash functions is hardly a strong defense. Neverthe- less, a recently proposed LLC remapping technique, namely CEASER [23], may turn all the mentioned reversing effort in vain. CEASER dynamically and randomly remaps physical addresses to cache sets in the LLC using a low latency block cipher. It is a very compromising defense against all conflict- based side-channel attacks. To defeat CEASER in the same way as the complex addressing, an adversary needs to com- pute an eviction set and exploit it all in the short remapping period. This would be an extremely intimidating task.

Instead of trying to reverse the mapping from virtual ad- dresses to cache sets, several approaches [18, 21, 26] in the literature discussed the possibility of finding minimal eviction sets with limited or even no control over the mapping. These algorithms normally comprise two steps: (1) the adversary first blindly collects a large collection of addresses enough to evict the target cache block, and then (2) prunes the large collection into a minimal eviction set. The size of the initial large collection is normally proportional to the size of the cache [26]. For a collection of n addresses, the original prun- ing algorithm proposed by Liu *et al*. [18] and Oren *et al*. [21] requires $O(n^2)$ memory accesses, which is terribly slow for large LLCs. Vila *et al*. [26] have recently proposed an opti- mized pruning algorithm which reduces the bound to $O(w^2n)$, where w is the number of ways in each cache set. In other words, *the necessary time for finding a minimal eviction set is linear with the size of the cache*. In addition, this linear bound invalidates the latency assumption used by CEASER which still assumed the naive bound of $O(n^2)$ memory accesses [23]. Although, in theory, the optimized algorithm [26] finds minimal eviction sets in linear time, in practice, the success rate on modern processors (after Haswell) is

ISSN: 2278-4632
Vol-10 Issue-3 No.01 March 2020

as low as just around 15% [26] and the latency in finding eviction sets is still too long for practical attacks.

Several reasons are known to contribute to this. *Translation lookaside buffer (TLB) noise*: Pruning from a large eviction set can cause false positive errors [5] as the TLB entry for the target cache block can be undesirably evicted from the TLB leading to a long latency similar to a miss in the LLC. *Adaptive cache replacement policies*: Introduced from Ivy- Bridge [12, 29], adaptive cache replacement policies are used to resist thrashing and scanning patterns in caches. This leads to both false positive errors where the target cache block is prematurely evicted and false negative errors where accessing an eviction set fails to evict the target cache block due to its scan-like pattern. *Hardware prefetching*, *inaccurate system timer*, *process scheduling* and potentially other undocumented hardware optimizations in the cache system may all contribute to the low success rate.

Vila's work [26] is great in discovering the linear time algo- rithm but it fails to push the algorithm to the limit, which we would try in this paper. We therefore focus on two ques- tions: **(a) In theory, how fast can an adversary find a minimal eviction set?** Vila *et al*. [26] provided only the up- per bound but not the average latency. Our analyses show: *The upper bound can be further reduced from* $O(w^2n)$ *to* $O(wn)$; *the average number of memory accesses is seriously less than the upper bound*; *the latency assumption used by CEASER [23] is significantly overestimated*. We then ask: **(b) In practice, how fast can a minimal eviction set be found on modern processors?** We have analyzed the source code provided by Vila *et al*. [26] and studied various eviction tech- niques [5, 7, 29]. A handful of new techniques are proposed to improve the speed and the accuracy of the optimized algo- rithm [26]. *Most importantly, we propose to use concurrent multithread execution to circumvent the thrashing resistant cache replacement policies [12]*.

Utilizing minimal eviction sets in actual attacks is out of the scope of this paper. We assume they can be efficiently used in attacks without reversing the virtual to physical address mapping [18, 20] and attacks launched inside a sandbox [5, 7, 21] or an SGX enclave [25] without huge pages.

Summary of major contributions: Our major contribu- tions are both theoretical and practical. On the theoretical side, we reduce the upper bound of the pruning algorithm [26] from $O(w^2n)$ to $O(wn)$ and provide an estimation of the av- erage number of memory accesses using an ideal cache. On the practical side, we propose to use concurrent multithread execution to circumvent the thrashing resistant cache replace- ment policies and provide a handful of new techniques to significantly improve the speed and the success rate of find- ing minimal eviction sets. To our best knowledge, we are the first to successfully find minimal eviction sets on a modern Intel processor (Skylake) with literally no information of the address mapping, even without exploiting the fact that page offset bits control a part of the cache set index.

2 Preliminaries

Caches and Virtual Memory

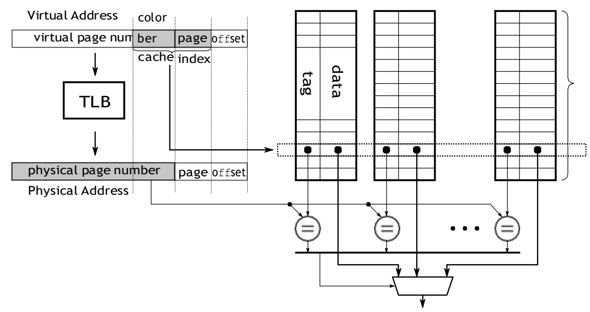
In modern processors, caches are utilized to store recently or frequently used data to reduce access time. Data are stored in units of fixed-sized cache blocks. Commonly, caches are set- associative which allow a group of cache blocks to reside in one of the many cache sets. Cache sets are addressed by cache index, which is typically a subset of the address bits shared by all cache blocks in the same set. When the processor

accesses
page b11

b5 b0

way-0 way-1 way-(w-1)

...



data

cache sets

set

an address, the cache checks whether the corresponding cache set has a block matching with the address (a hit). If no match is found (a miss), the cache block is fetched from memory and stored in the cache set for future use. If the cache set is fully occupied at this moment, a block is chosen by a replacement policy and evicted from the set. As a commonly used policy, least-recently used (LRU) would keep recently accessed cache blocks in the cache.

Caches are hierarchically organized in modern processors. Each core contains a pair of small level-one (L1) caches for data and instruction. The core may contain a medium-sized level-two (L2) cache. All cores share a large last-level cache (LLC). An inclusive cache hierarchy is normally adopted. When a cache block is evicted from the LLC, it is also purged from all cache levels. A cache coherence protocol is utilized to ensure that data are correctly updated in all caches.

User land applications run in the virtual memory space while physical memory is dynamically allocated to virtual memory in unit of pages, normally 4KB sized. The mapping is stored in page tables which are also cached in the cache hierarchy. L1 caches are typically addressed by virtual ad- dresses, while the LLC is addressed by physical addresses. Figure 1 depicts a virtually indexed and physically tagged cache (normally used as L1). The virtual to physical address translation proceeds in parallel with cache set accesses. A translation lookaside buffer (TLB) is used to directly translate the virtual page number into the corresponding physical page number, if such translation has been recently used and cached in the TLB. If TLB fails to translate a virtual page number, the page table is accessed to refill the TLB, which leads to a long latency penalty. Also shown in Figure 1, virtual and physical addresses share the same page offset bits, which is also partially used in the cache index. The LLC has a similar structure but without the TLB as it is indexed by physical addresses.

Eviction Set

Definition 1. For a specific cache, two virtual addresses x and y are *congruent*, denoted as x y, if and only if x and y are mapped to the same set, $set(x) = set(y)$ [26], but they do

Figure 1: A virtually indexed physically tagged cache.

$$x \simeq y \iff set(x) = set(y) \wedge cb(x) \neq cb(y) \quad (1)$$

Congruence is an equivalence relation. The equivalence class $[x]$ of x with regarding to is the set of cache blocks mapping to the same cache set with x. We now give the defi-nition of an eviction set.

Definition 2. A set of virtual addresses S is an eviction set for a target address x if x / S and at least a addresses in S are congruent with x [26]:

$$x \notin S \wedge |[x] \cap S| \geq a \quad (2)$$

The intuition behind Definition 2 is that, if x is initially stored in a cache set, accessing congruent elements in a certain order can systematically evict x from the cache set, where a is the minimal number of congruent elements needed.

For caches adopting permutation-based replacement poli- cies [2], such as FIFO, least-recently used (LRU) and pseudo- LRU (PLRU) [3], sequentially and repeatedly accessing w congruent cache blocks, where w is the number of ways in a cache set, guarantees the eviction of any block originally stored in the set. In this case, a is equal with w. However, for processors adopting thrashing resistant replacement poli- cies [12], sequentially accessing is recognized as a scan and thus the accessed blocks are the first to be replaced rather than those originally stored. As a result, sequentially accessing an eviction set no longer guarantees the eviction of the target, even when *a w*. The key here is to change the access pattern. As analyzed in Section 4.1, choosing a proper traverse function and utilizing concurrent multithread execution re- duce the minimally required number of congruent elements to *w* as well. Finally, we can define the concept of a minimal eviction set.

Definition 3. A set of virtual addresses *S* is a minimal evic- tion set for a target address *x* if and only if x / S, S has w elements, and all the w elements are congruent with x [26]:
not address the same cache block, $cb(x)cb(y)$:

$$x \notin S \wedge |S| = w \wedge [x] \cap S = S \quad (3)$$

3 Find Eviction Sets in Ideal Caches

In this section, we answer the first question: **In theory, how fast can an adversary find a minimal eviction set?** A sys- tematic analysis of the existing algorithms has been done using an ideal cache. We have further reduced the latency up- per bound, revealed several parameters which can be tuned to reduce the average latency, and improved existing algorithms to boost their tolerance to noise.

An Ideal Cache

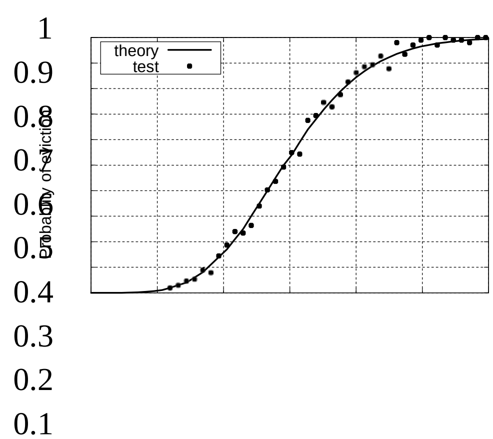
To conduct a systematic analysis of the complexity in finding eviction sets and reduce the noise introduced by the actual hardware implementation, an ideal cache model is adopted. It has the following characteristics:

Universal access latency: Accessing a cache block has the same latency disregarding to its location (level, set, way or slice) and status (hit or miss). This assumption allows measuring the search time by the number of memory accesses. **Ideal hit/miss status:** Search algorithms can directly and accurately inquire the status of a cache block (hit or miss) with no time penalty. This removes the errors of measuring the accessing latency on actual processors.

No TLB noise: The model assumes a uniformly random- ized mapping from virtual to physical addresses. Virtual ad- dresses are translated into physical addresses before cache accesses. TLB is not needed and page table entries are not cached.

LRU replacement: Adaptive or random replacement poli- cies can cause a large quantity of errors. To remove this effect, the model assumes a strict LRU replacement policy.

Randomized cache layout: The model adopts a fully randomized cache layout similar to the static version of CEASER [23]. Virtual addresses are randomly mapped to all cache sets; therefore, search algorithms cannot reduce



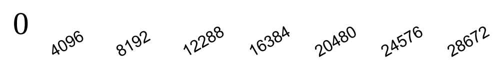


Figure 2: Probability of finding a candidate set as a function of its size. The cache has 1024 sets and 16 ways in each set. **Theory** shows the probability calculated from Equation 5. **Test** shows the test results using the ideal cache. Each result is averaged from 100 independent experiments.

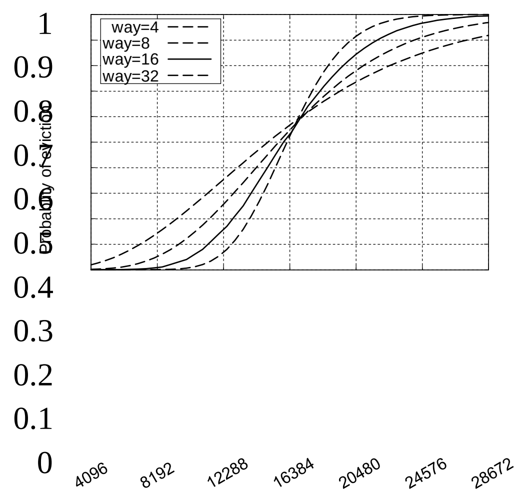


Figure 3: The probability of finding a candidate set in caches with the same size (16384 blocks) but different ways.

where p is the probability that a random virtual address is congruent with x. For a cache with s sets and w ways in each set, $a = w$ and $p = s^{-1}$:

their complexity by reversing the mapping.

$$Pr(X \geq w) = 1 - \sum_{i=0}^{w-1} \binom{n}{i} \frac{(s-1)^{n-i}}{s^n} \qquad (5)$$

Find a Candidate Set

The first step in finding a minimal eviction set is to create a large (non-minimal) eviction set, called a *candidate set*. Since the mapping from virtual addresses to cache sets is random- ized, a candidate set is acquired by randomly collecting a large collection of virtual addresses.

According to Definition 2, a collection of virtual addresses is a candidate set for the target address x if there are more than a congruent addresses in the collection. The probability that a collection of n virtual addresses is a candidate set for x can be calculated using the binominal distribution:

ISSN: 2278-4632
Vol-10 Issue-3 No.01 March 2020

Figure 2 depicts the probability of finding a candidate set as a function of its size. It is shown that the test results us- ing the ideal cache closely match the probability calculated from Equation 5. The probability of finding a candidate set increases with its size.

There is an interesting observation: When the size is around the number of blocks in a cache (16384 in this case), the probability of finding a candidate set is around 50%. Figure 3 depicts the probability of finding a candidate set in caches with the same number of blocks but different number of ways. It is shown that, independent of the set-way configuration, achieving a probability around 60% requires the same number

of virtual addresses which is just above he totanumber of

$$) = \; - \sum_{i=0} \quad (\; - \;)$$

$$(\; X \quad a$$

$$1 \quad a-1 \quad n \quad p^{i} \; 1$$

$$p^{\,n-i} \qquad (4)$$

blocks in the cache. It is possible to detect the size of the LLC by searching the size achieving the 60% probability.

75000
70000
65000
60000
55000
50000
45000
40000
35000
30000
25000
20000

Latency (memory accesses)

12288
28672
16384
20480
24576

Size of collection

ISSN: 2278-4632
Vol-10 Issue-3 No.01 March 2020

Algorithm 1: Baseline algorithm: prune_base(C,x)

```
Input: C, candidate set; x, target address.
Output: Minimal eviction set for x.
function prune_base(C,x)
    foreach c in C do
        if test(C \ {c}, x) then
            C ← C \ {c}
        end
    end
    return C
end
```

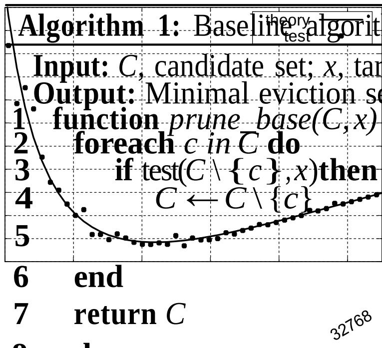


Figure 4: The average latency of finding a candidate set as a function of its size. The cache has 1024 sets and 16 ways in each set. **Theory** shows the probability calculated from Equation 6. **Test** shows the averaged latency of finding a candidate set in the ideal cache. Each result is averaged from 100 independent experiments.

To estimate the average latency of finding a candidate set, we assume the adversary to choose a fixed size for the candi- date set. She repeatedly acquires a collection and verifies it until a candidate set is found. In the verification, the adversary accesses all elements in the collection after first visiting the target address, and then checks whether the target address is evicted. Using the number of memory accesses as the mea- surement of time, the average latency of finding a candidate set with n elements can be calculated as $T_c(n)$:

$$T_c(n) = \frac{n}{Pr(X \geq w)} \qquad (6)$$

Figure 4 demonstrates the average latency of finding can- didate sets of different sizes. For a 1024-set 16-way cache, finding a candidate set of 20480 elements in around 25000 memory accesses is the quickest. Finding a smaller candidate set needs more retrials while larger sets suffer from longer verification latency.

Prune an Eviction Set

With a candidate set available, the rest is to prune it into a minimal eviction set. Algorithm 1 is the original pruning algorithm proposed by Liu *et al.* [18] and Oren *et al.* [21]. The algorithm takes the target virtual address x and a candidate set C as inputs. For each element c in the candidate set, the algorithm tests whether the candidate set is still an eviction set without c. If yes, *test*($C \setminus \{c\}, x$) returns true and c is then removed from the set. When all elements are tested, the remaining candidate set becomes a minimal one. As analyzed

$$T_{base}(n) \sim O(n^2) \qquad (7)$$

Proposition 4 can be proved by finding that both the upper and the lower bounds approach $O(n^2)$. The baseline algorithm is quadratic, which is slow for large caches. Nevertheless, it has a benefit that the algorithm does not need to know the number of ways. It can be used to detect this information.

An optimized algorithm was recently proposed by Vila *et al*. [26], as illustrated in Algorithm 2. Instead of removing at most one element in each iteration, multiple elements are removed in Algorithm 2. For each **while** iteration, the remain- ing candidate set is split into l groups $G_1,\ldots,G_l$. For each group G, it is tested whether the target x can be evicted with- out it. If yes, $test(C \setminus G, x)$ returns true and G is removed. The
success of the algorithm depending on the split parameter l.
In the worst scenario, elements of the minimal eviction set distribute evenly in all groups. If $l > w$, it is guaranteed that $l - w$ groups can be removed in each **while** iteration. Vila *et al*. set l to $w+1$ to maximize the group size [26].

Algorithm 2: Prune with split: prune_split(C,x,w,l)

Input: C, candidate set; x, target address; w, number of ways; l, split parameter.
Output: Minimal eviction set for x.

```
function prune_split(C,x,w,l)
    while |C| > w do
        {G1,...,Gl} ← split(C,l)
        foreach G in {G1,...,Gl} do
            if test(C \ G,x) then
                C ← C \ G
            end
        end
    end
    return C
end
```

Proposition 5. Assuming $l > w$, the number of memory ac- cesses of using Algorithm 2 to prune a candidate set of n elements has an upper bound:
in Section 2.2, the size of the minimal eviction set should be

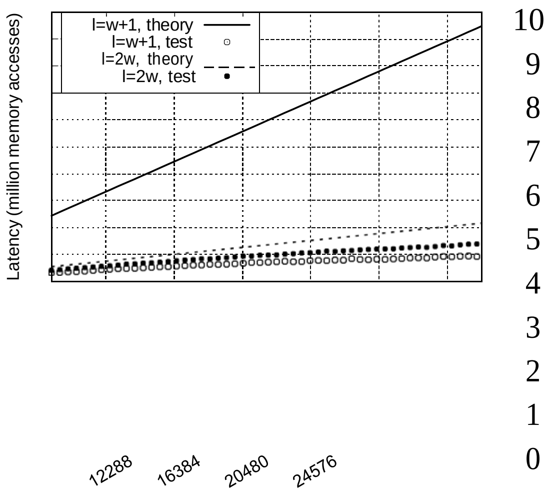
$$T_{split}(n) < \frac{l(l-1)n}{l-w} \quad (8)$$

w, the number of ways.

Proposition 4. Algorithm 1 prunes a candidate set of n ele-ments into a minimal eviction set in $O(n^2)$ memory accesses:

See Appendix A for a proof similar to the proof provided in [26]. When l is set to $w+1$, the upper bound is w^2n+wn

ISSN: 2278-4632
Vol-10 Issue-3 No.01 March 2020



Algorithm 3: Random split: prune_random(C,x,w,l)

Input: C, candidate set; x, target address; w, number of ways; l, split parameter.
Output: Minimal eviction set for x.

```
function prune_random(C,x,w,l)
    while |C| > w do
        G ← random_split(C,l)
        if test(C \ G,x) then
            C ← C \ G
        end
    end
    return C
end
```

Figure 5: The average latency of pruning a candidate set as a function of its size. The target cache has 1024 sets and 16 ways in each set. **Theory** shows the latency upper bound from Proposition 5. **Test** shows the average latency of prun-ing a candidate set using different split parameters in the ideal cache. Each result is averaged from 100 independent experiments.

approaching $O(w^2n)$ as described in [26]. However, this is not the optimal bound for $w \geq 4$. **The optimal upper boundis** $(4w-2)n$ **approaching** $O(wn)$ **when the split parameteris set to** $2w$ **(for caches with** $w \geq 4$**).**

Figure 5 reveals the average pruning latency compared with different upper bounds in a 1024-set 16-way cache. The opti-
mal upper bound using $l = 2w$ is significantly lower than the upper bound using $l = w + 1$. Both upper bounds are higher than the average latency from cache tests but **the optimal upper bound of** $(4w-2)n$ **is a more accurate estimator.**

Note that using an algorithm with a lower upper bound does not result in a lower average latency. The average latency of using $l = 2w$ is constantly higher than using $l = w + 1$. The reason is the early termination optimization of the inner **foreach** loop which is normally applied. When $l = w+1$, only one group is guaranteed removable in the worst case. The

foreach loop in Algorithm 2 can finish immediately when this group is found. However, nearly all groups are tested when $l = 2w$ because there are at least w removable groups. On actual processors, errors are inevitable due to hardware noise. Algorithm 2 might fail when removable groups are mistakenly found irremovable. Algorithm 3 is a more robust algorithm. It combines the **foreach** inner loop into the **while** outer loop. In every iteration, *random_split*(C,l) picks C/l random elements from C, equivalent to a group in Algorithm 2, and tests whether they are removable. Algorithm 3 takes the benefit of the early termination optimization while keeps try- ing when a removable group is mistakenly tested irremovable. Although we cannot derive a mathematical upper bound for Algorithm 3, its average latency performance is similar to Algorithm 2.

Figure 6a shows the overall latency of finding a minimal eviction set, including the latency in finding a candidate set

and pruning it. Algorithm 2 and 3 have similar latency when using the same split parameter. All algorithms achieve their lowest latency around size 11000 disregarding the algorithm and the split parameter, while $l = w + 1$ produces the lowest average latency of around 0.55 million memory accesses in all cases, which is around 50 times of the candidate size. However, neither $w + 1$ nor $2w$ is the optimal split parameter for Algorithm 3. As shown in Figure 6b, $l = 14$ produces an even lower latency for the 16-way cache. The long tail distribution of the overall latency is revealed in Figure 6c.

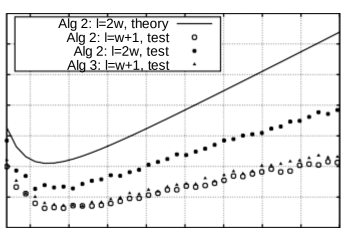
For defenses trying to prohibit an adversary from finding minimal eviction sets by dynamically remapping the cache layout, such as CEASER [23], **the remapping period should be shorter than the lowest overall latency of finding a min- imal set**. As shown in Figure 6c, the lowest latency is located at leftmost point of the distribution, which is smaller than the median latency. Figure 7 shows the 1st (1%) and 5th per- centile (5%) of the overall latency in different caches. To se- curely prohibit an adversary from finding a minimal eviction set with a probability of 99%, the remapping period is con- strained by the 1st percentile of the latency, which is roughly 40% of the optimal upper bound. For the 1024-set 16-way cache, the 1st percentile is roughly $25n$, where n is around 11500. It is only 0.2% of the naive bound of $O(n^2)$ used in CEASER [23], **which has significantly overestimated the time complexity in finding minimal eviction sets.**

4 Find Eviction Sets on Actual Processors

In this section, we answer the second question: **In practice, how fast can a minimal eviction set be found on modern processors?** As shown in Table 1, three different platforms are used in the evaluation. On all platforms, the search algo- rithm runs in user land without the root privilege. Algorithm 3 is used as the pruning algorithm due to its near optimal latency and tolerance to noise. Candidate sets are collected from a pre- allocated pool of 1GB in the virtual address space. By default, elements in the pool has the same page offset so that eviction sets are found at the granularity of pages. Later in Section 4.4, results for finding eviction sets comprised of elements with

Latency (million memory accesses)

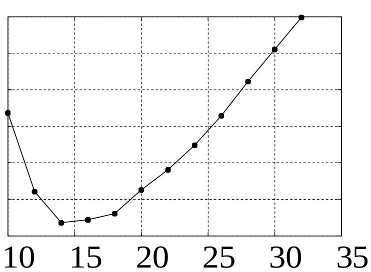
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4



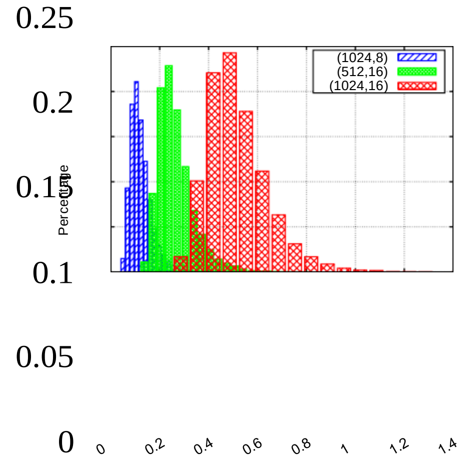
Latency (million memory accesses)

0.66
0.64
0.62
0.6
0.58
0.56
0.54

26624



ISSN: 2278-4632
Vol-10 Issue-3 No.01 March 2020



10240 12288 14336 16384 18432 20480 22528 24576

Size of collection

(a) The overall latency of finding minimal evic- tion sets in a 1024-set and 16-way cache as a function of the size of the candidate set. **Theory** shows the optimal upper bound while **test** shows the latency using different algorithms and split parameters.

Split parameter

(b) The overall latency of finding minimal evic- tion sets in a 1024-set and 16-way cache using Algorithm 3 with different split parameters.

Latency (million memory accesses)

(c) The long tail distribution of the overall la- tency in different (set, way) caches using Algo- rithm 3 with $l = w$.

Figure 6: Analyses for the overall latency in finding minimal eviction sets.

ISSN: 2278-4632
Vol-10 Issue-3 No.01 March 2020

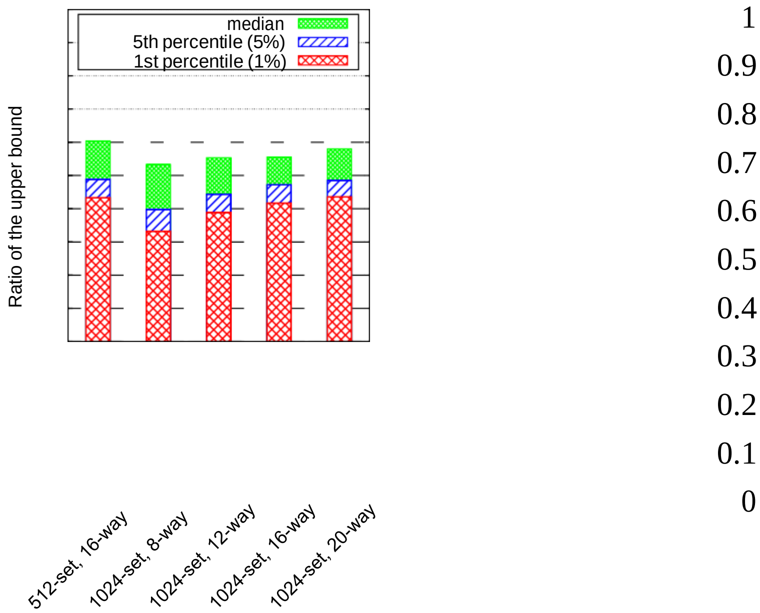


Figure 7: The overall latency of different cache configurations at the 1st (1%) and the 5th (5%) percentile, and the median. Results are normalized by their optimal upper bounds.

arbitrary page offset bits (at the granularity of cache blocks) are presented. The complex address scheme in the LLC is not reversed. Huge pages are not used unless explicitly noted. The initial size of candidate sets is chosen to achieve 50% of eviction using a sliding test similar to Figure 2, which is 2700 for i7-3770, and 3500 for both Xeon-4110 and i7-8700. The split parameter is set to the number of ways. The high resolution time stamp counter is used for time measurement [24] and the clflush instruction is used only in the automatic calibration of the LLC miss threshold.

Eviction Test

The key challenge in finding minimal eviction sets on actual processors is to quickly and accurately test whether a col- lection of virtual addresses is an eviction set. Algorithm 4

Table 1: Evaluation Platforms

	i7-3770	Xeon-4110	i7-8700
Architecture	IvyBridge	SkyLake	Coffee Lake
Cores	4	8	6
Threads	8	16	12
LLC Size	8MB	11MB	12MB
Cache Way	16	11	16
Memory	4GB	32GB	32GB
OS (Ubuntu)	16.04	16.04	18.04

illustrates a generic *test*() function used in the pruning al- gorithms. Instead of sequentially accessing a candidate set as on the ideal cache, the set is accessed using a dedicated *traverse*() function for multiple times. Normally the result is averaged from *b* repeated tests and the candidate set is tra- versed *d* times in each test. The latency of accessing the target address *x* is measured using a *time*() function.

Algorithm 4: Eviction test function

Input: *C*, candidate set; *x*, target address.

Output: Whether *C* evicts *x*.

Parameter : *h*, LLC eviction threshold; *b*, number of repeats; *d*, number of traverses; traverse(), traverse function.

```
function test(C,x)
    i 0, j 0
    while i < b do
        access(x)
        while j < d do
            traverse(C)
            j ← j + 1
        end
        record t ← time(access(x))
        i ← i + 1
        j ← 0
    end
    return t̄ > h
```

endime measurement and LLC eviction threshold: The access latency of *x* is used to tell whether *x* is evicted from the LLC. The accuracy in this measure affects the speed and accuracy of the pruning algorithm. In our experiment, the time stamp counter provided by the Intel processors is directly usedas an accurate time source [24]. On platforms where such time source is unavailable, it is possible to construct an accurate counter using a separate thread [6,24]. If the latency of access- ing *x* is larger than a threshold *h*, *x* is assumed evicted from the LLC. Such threshold is obtained from an automatic cali- bration [26] on each processor using the clflush instruction.For platforms where clflush is unavailable, measuring the data accessing latency by deliberately thrashing the LLC [29]

1200

1000

800

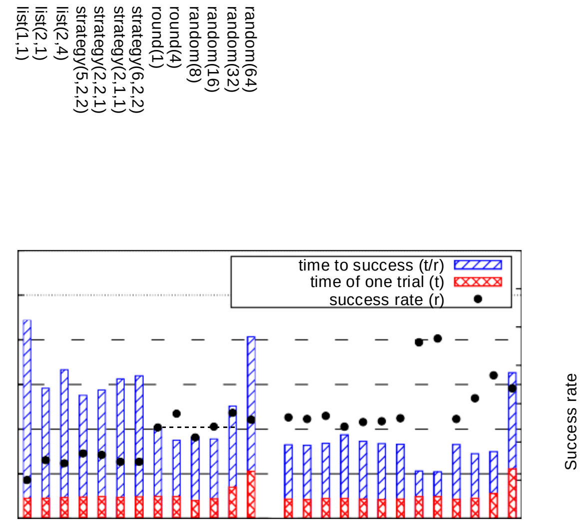
time (ms)

600

400

200

0



i7-3770

i7-8700

0.7

0.6

0.5

0.4

0.3

0.2

0.1

0

produces the same result.

Traverse function: The choice of traverse functions de- pends on the cache replacement policy of the LLC. For pro-

(a) The performance of different traverse functions on i7-3770 ($b = 4, d = 4$) and i7-8700 ($b = 4, d = 3$).

cessors using permutation-based replacement policies [2], sequentially accessing candidate sets would be sufficient. However, complex traverse functions become necessary when thrashing resistant replacement policies [12] are used, as on modern processors. Four types of traverse functions are ana- lyzed in our experiments:

List traverse *list*(m, n)**:** Traversing an eviction set using a moving window has been found effective on some processors [29]. For each $c_i \in C$, *list*(m, n) accesses $\{c_i \dots c_{i+m-1}\}$ for n times. It tries to hide its scan-like pattern by accessing the elements multiple times in short intervals. However, it might be ineffective when the evic- tion set C is not a minimal one, where $c_i \dots c_{i+m-1}$ might

1000

800

time (ms)

600

400

200

round(4) random(16) i7-3770

round(4) random(16) i7-8700

0.7

0.6

0.5

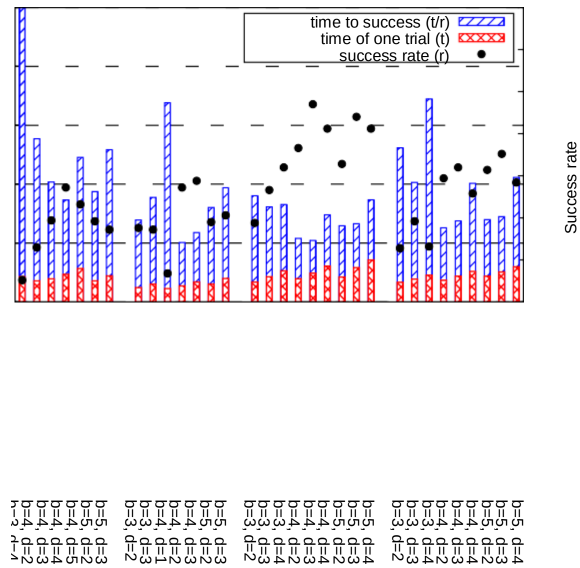
0.4

0.3

0.2

0.1

0



belong to different sets and most accesses to the LLC are filtered by the inclusive L1 cache.

Strategy traverse *strategy*(m, n, δ)**:** Cache eviction strategy was first proposed in [7]. Stragtegy traverse takes a more generic form than list traverse as *list*(m, n) is equivalent to *strategy*(m, n, 1) in theory. The parameter δ controls the incremental step of the outer loop. To be specific, if c_i is traversed using *list*(m, n) in the current iteration, $c_{i+\delta}$ would be traversed in the next iteration. When $\delta > 1$, *strategy*(m, n, δ) reduces the total number of memory accesses by a factor of δ compared to *list*(m,n).

Round trip traverse *round*(n)**:** First proposed in [18], a round trip traverse sequentially accesses an eviction set forward and then backward. For each $c_i \in C$, it is ac- cessed n times. This guarantees that the access pattern seen by each LLC set is a round trip rather than a simple scan.

Random traverse *random*(n)**:** This is a new traverse func- tion proposed in this paper. For each $c_i \in C$, it is placed in a buffer along with $n-1$ previously accessed and randomly selected elements. All the n elements in the buffer are accessed instead of c_i alone. This random pat- tern is likely to trigger repeated accesses to the LLC and therefore break its scan-like pattern.

Figure 8a demonstrates the results of finding minimal evic-

(b) The performance of using different (b,d) combinations.

Figure 8: Experiments using different configurations. Each result is averaged from 500 independent trials. *Time to success* refers to the estimated average latency of successfully finding a minimal eviction set, which is calculated as *time of one trial* (t) divided by the *success rate* (r).

tion sets using different traverse functions. We use *time to success* as the main criterion in comparing different algo- rithms. As an estimation of the average latency of success- fully finding a minimal eviction set, it is calculated as the *time of one trial* divided by the *success rate* averaged from all trials. On i7-3770 (IvyBridge), *round*(4) and *random*(16) perform equally well while *round*(4) performs the best on the latest i7-8700 (Coffee Lake). Interestingly, on Xeon-4110 (Skylake), no eviction set is found no matter which function is used.

Repeat parameters: The number of repeats b and the num- ber of traverses d in each repeat also affect the results. Fig- ure 8b reveals the performance of using the best traverse func- tion with different repeat parameters (b,d). On i7-3770, the best time to success is produced by $b = 4, d = 2$, *random*(16), while it is $b = 4, d = 3$, *roud*(4) on i7-8700.

Multithread traverse: To circumvent the thrashing re- sistant cache replacement policies used in modern proces- sors [12], we propose to use concurrent multithread execution to do parallel rather than sequential traverse. The idea is sim- ple. Although different traverse functions try to break the scan-like pattern by making multiple accesses to the same cache block, the inclusive L1 cache filters most of the extra accesses. Modern processors are usually multiprocessors containing four to eight cores (double for hardware threads). Meanwhile, attackers are capable of motivating multiple threads even in a JavaScript sandbox thanks to the latest support of web work- ers and JavaScript's SharedArrayBuffer[5, 6, 9].[1] When

```
void worker() {
  bool work = false;
  while (true) {
    lock();  // critical section begin
    if(jobs > 0) {
      jobs--;
      work = true;
    } else {
      work = false;
    }
    unlock(); // critical section end

    if(work) {
      traverse(); // the chosen traverse func
      lock();     // critical section begin
      done++;
      unlock();   // critical section end
    }
  }
}
```

ISSN: 2278-4632
Vol-10 Issue-3 No.01 March 2020

and the modification of both atomic variables are guarded by locks.

Listing 2: Worker Thread

a cache block is concurrently accessed by multiple cores, [10]

ISSN: 2278-4632
Vol-10 Issue-3 No.01 March 2020

the LLC almost always receives multiple requests unless the block has been cached in all the L1 caches of all cores. This would effectively break the scan-like pattern and significantly increases the success rate of eviction.

Instead of doing the *d* traverses sequentially as described in Algorithm 4, the candidate set is traversed concurrently using *d* worker threads. It was found that creating and destroying threads cause significant amount of noise. Worker threads are created beforehand and kept in idle waiting status until traverse jobs are broadcast through global atomic variables.

Listing 1 shows the creation of worker threads (worker()). The traverse jobs are scheduled using the two global atomic variables: *jobs* (the number of traverse jobs to be claimed) and *done* (the number of traverse jobs being done). The create_thread() function is called only once at the be- ginning to initialize the global atomic variables, create several worker threads, and make them detached from the main thread. The number of workers depends on the chosen number of tra- verses (*d*).

Listing 1: Worker Initialization

```
//global variables
atomic<int> jobs;
atomic<int> done;
int d = number of traverse;

void create_threads() {
  jobs = 0;
  done = 0;
  for(int i=0; i<d; i++) {
    thread t(worker);
    t.detach();
  }
}
```

Listing 2 explains the internal procedure of worker threads. For each worker (worker()), it constantly checks whether there are unclaimed jobs (jobs > 0). If yes, the worker con- sequently claims a job (jobs--), does the traverse, and in- creases the job count (done++). To avoid race conditions among workers and the main thread, the reading of jobs

1 As a mitigation of the Spectre attack, SharedArrayBufferwas dis- abled by default in Firefox from 52 ESR [1].

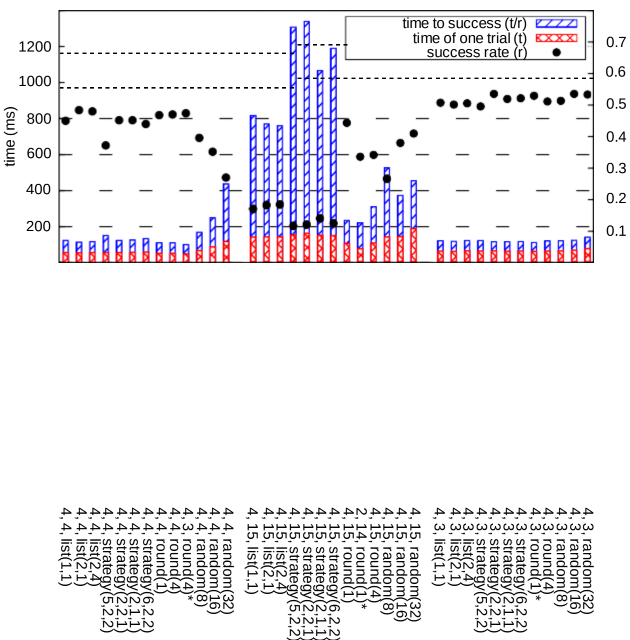
The sequential traverse in Algorithm 4 (line 5–8) is then replaced by the code segment as shown in Listing 3. traverse.cfg() broadcasts the chosen traverse function and the candidate set to all workers. Consequently, the main thread triggers all workers by clearing the job count (done =0) and setting the number of unclaimed jobs to the number of traverses (jobs = d). To avoid race conditions, the modifi- cation of jobs is guarded by locks. The multithread traverse finishes when all jobs are claimed and done (jobs == 0 &&done == d).

Listing 3: Multithread Traverse

```
traverse.cfg() // set the traverse functi

done = 0;      // clear job count
lock();        // critical section begin
jobs = d;      // trigger works
unlock();      // critical section end

while(jobs != 0 || done != d); // detect
```

```
on

end
```

The results of using multithread traverse are shown in Fig- ure 9. The success rate of using multithread traverse is signif- icantly higher than that of using a single thread. The time for each trial is also shortened as traverses are done in parallel. These two factors together result in a much lower time to success compared with the results in Figure 8a. When the best configuration of traverse function and repeat parameter is used, multithread traverse reduces the time to success by 50% and 52% on i7-3770 (IvyBridge) and i7-8700 (Coffee Lake) respectively. Finding minimal eviction sets also be- comes possible on Xeon-4110 (Skylake) by using as many as 15 concurrent workers. The best configuration is found to be
$b = 2, d = 14, round(1)$.



1400

0

0.8

Success rate

0

800

600

time (ms)

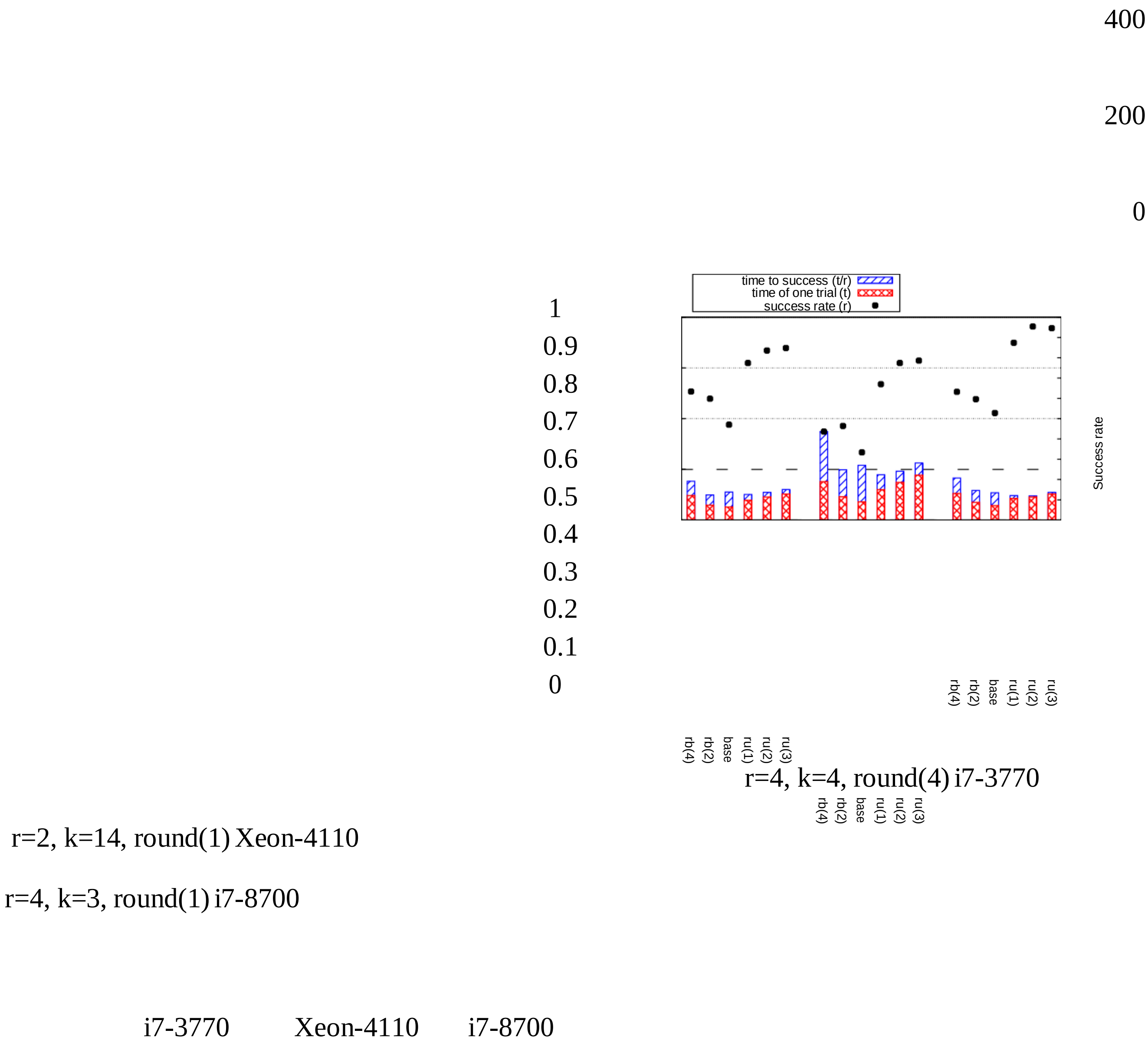


Figure 9: Experiements using concurrent multithread tra- verse. The parameter of each configuration is shown as (*b*, *d*, traverse). The configuration with the lowest time to success is labelled with '*'.

Algorithm Optimizations

It is also possible to improve the performance by further optimizing Algorithm 3. We analyze two methods in this paper: One is the *rollback support* first utilized in [26] and the other one is to *reuse the failed set*, a new optimization proposed by this paper.

Rollback support: When some elements are mistakenly removed from a candidate set, Algorithm 3 may fail to prune it into a minimal one. To tolerate this type of false positive errors, the algorithm can roll back by adding the removed elements back and try again. In practice, the algorithm keeps a limited number of recently removed groups of elements. When a predefined maximal number of retries is reached, the latest removed group is added back and the algorithm redoes the test. The algorithm ultimately fails when all kept groups are added back but the maximal number of retries is still reached.

Reuse the failed set: When a trial fails to find a minimal eviction set, the remaining collection is normally significantly smaller than a new candidate set and has a high density of the irremovable elements for a minimal eviction set. Instead of throwing the collection away, it is kept as a residue set. If the next trial fails as well, the new remaining collection and the stored residue set is combined as the new candidate set for the next trial. Using this combined candidate set has a significantly higher chance in producing a minimal one than a normal candidate set. If it still fails, the remaining collection becomes the new residue set and the whole process starts again.

Figure 10 depicts the improvement achieved by the two optimizations. To produce a fair comparison, the time of a single trial for cases reusing the failed set is the accumulated time of all retries targeting the same virtual address. On all

gure 10: Experiments using algorithm optimizations. Mul- tithread traverse is used. $rb(i)$ denotes the maximal level of allowed rollbacks is i. $ru(i)$ denotes the maximal number of reused trials is i. Both rollback and reuse are disabled in *base*.

processors, reusing the failed set boosts the success rate sig- nificantly with a moderate latency penalty. As for rollback, the improvement on the success rate is moderate but the la- tency overhead becomes significant when the maximal level of rollbacks is more than four. A combination of reusing the failed set and a shallow rollback should reduce the time to success while boosting the success rate.

Other Optimizations

Besides all the aforementioned techniques, there are extra op-timizations that can be utilized to reduce the time to success. **TLB preload:** Accessing a large candidate set may trigger false positive errors when TLB entries are mistakenly evicted from the TLB. To reduce this effect, Genkin *et al.* proposed to access another virtual address inside the same page with the target address before checking its cache status [5]. This effectively preloads the TLB entries and therefore reduces false positive errors.

Use huge pages: The underlying reason for the TLB noise is that the large number of pages visited by a candidate set over-stress the comparatively small TLB. Allocating candi- date sets from huge pages [26] would significantly reduce the number of pages visited in traversing a candidate set, which thus reduces false positive errors.

Increase retry limit: Allowing extra retries for each can- didate set increases the chance of finding removable elements. However, this also increases the time wasted on the candidate sets which would fail ultimately.

Tune the size of candidate sets and split parameters: As described in Section 3.3, finding the optimal candidate size and split parameter would reduce the time of a single trial, which then leads to reduced time to success.

Figure 11 demonstrates the results of using preloaded TLB and huge pages. When multithread traverse is not utilized, preloading TLB indeed improves the success rate, which is significant on i7-3770 (IvyBridge). However, its benefit is

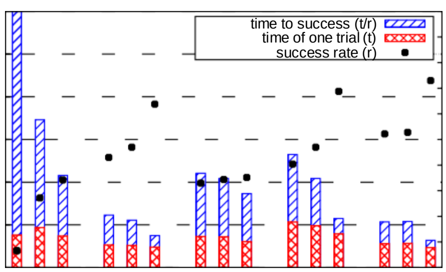
ISSN: 2278-4632
Vol-10 Issue-3 No.01 March 2020

600
500
400
300
200
100
0

time (ms)

st, 4k st, pl, 4k st, pl, 2m mt, 4k mt, pl, 4k mt, pl, 2m mt, 4k mt, pl, 4k mt, pl, 2m st, 4k st, pl, 4k st, pl, 2m

r=4, k=4, round(4)
r=2, k=14, round(1) r=4, k=3, round(1)



1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0

mt, 4k mt, pl, 4k mt, pl, 2m

Table 2: Space of Parameters

Parameter	Good Setting	Proposed by
prune algorithm	Algorithm 3	this paper
repeat parameter	$b = 4, d = 4$	[7, 26]
multithreading	enable	this paper
traverse function	*round*(1)	[18, 26]
rollback	*rb*(2)	[26]
reuse failed set	*ru*(1)	this paper
TLB preload	enable	[5]
huge page	enable	[26]
retry limit	*rt*(4*w*)	this paper
candidate set	$n = s \cdot w/64$	this paper

i7-3770

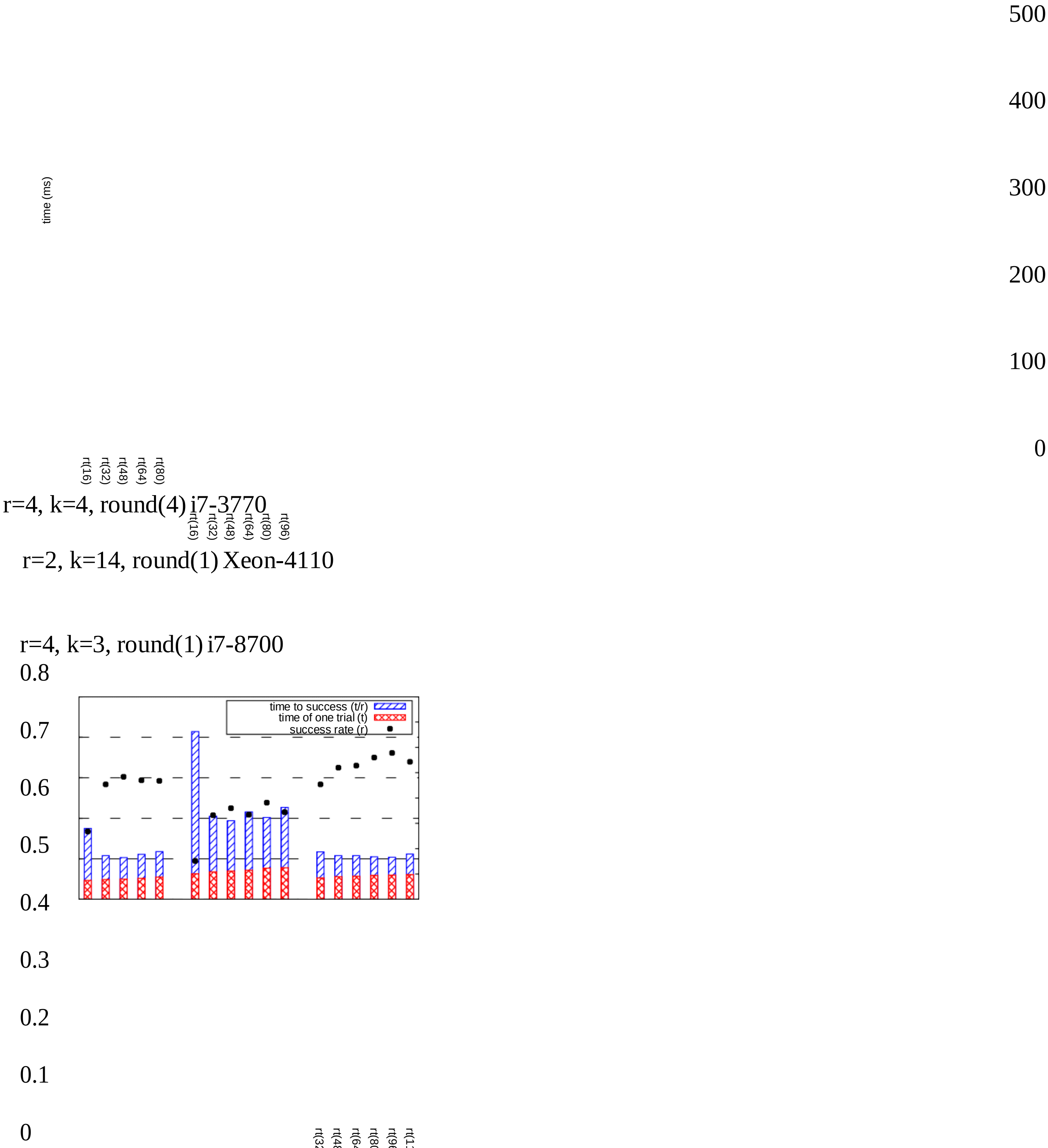
ISSN: 2278-4632
Vol-10 Issue-3 No.01 March 2020

Xeon-4110
7-8700

split parameter $l = w$ this paper

Figure 11: Experiments with preloaded TLB and huge pages. *st* (*mt*) denotes the tests using a single (multiple) thread(s) in traversing. *pl* denotes the TLB entries are preloaded before the eviction test. 4*k* and 2*m* denote the page size. So 2*m* means huge pages are enabled.

The experiments with differently sized candidate sets and split parameters show similar trends with the results on the ideal cache as shown in Figure 6a and 6b in Section 3.3. The optimal sizes of candidate sets and split parameters (n, l) are



found to be (2700, 12), (3500, 9) and (4500, 12) for i7-3770, Xeon-4110 and i7-8700 respectively.

Best Practice and Summary

Table 2 lists all the parameters that can be tuned to improve the speed and accuracy of finding

ISSN: 2278-4632
Vol-10 Issue-3 No.01 March 2020

minimal eviction sets. Ob- viously, this is a large parameter space to search, especially for a new processor. To reduce the search effort, the "Good Setting" column provides a known good or nearly optimal setting for each parameter. Note that the size of the candidate set is provided assuming eviction sets are found at the page

Figure 12: Experiments with different retry limits. Multi- thread traverse is used. $rt(i)$ denotes the maximal number of retries for each level is i.

marginal when multithread traverse is used. In this case, the traverse of candidate sets is done by worker threads rather than the main thread. The TLB entries for the main thread are therefore less disturbed than worker threads and preloading TLB has little effect on the testing results. Using huge pages substantially boosts the success rate in all cases except for Xeon-4110 (Skylake). Another benefit of using huge pages is the reduced time for a single trial as the penalty introduced by page faults is reduced. Overall, both techniques can effec- tively reduce TLB noise. Preloading TLB is necessary when multithread traverse is not used and huge pages is generally beneficial for all scenarios.

Generally speaking, increasing the number of retries im- proves the success rate, as shown in Figure 12. However, it also prolongs the time of a single trial and thus raises the time to success for some cases. It looks like, setting the retry limit to four times of the number of ways provides comparatively good results. In our experiments, the best settings are 48, 48 and 96 for i7-3770, Xeon-4110 and i7-8700 respectively.

granularity (constant page offset). For finding eviction sets at the granularity of cache blocks, $n = s \; w$ is a good start.

In this paper, we have done a manual search of the optimal settings for finding eviction sets on the three processors un- der evaluation and Table 3 presents the best results we have achieved so far. When finding eviction sets at the granular- ity of pages, we seek to find the best setting for the shortest time to success for four different scenarios (combination of the availability of huge pages and multithreading). On Xeon- 4110, we failed to find minimal eviction sets when multithread traverse is not used. We consider the scenario with multithread traverse but without huge pages as the common case. For the common case, it is possible to find minimal eviction sets in just 0.085s, 0.170s and 0.095s on i7-3770 (IvyBridge), Xeon- 4110(SkyLake) and i7-8700 (Coffee Lake) respectively.

We have also estimated the highest success rate achievable by increasing the number of reuses to 50 (except for i7-8700 as it is already high). It is shown that the highest rates are 99.2%, 97.0% and 95.4% on i7-3770, Xeon-4110 and i7-8700
respectively.

To our best knowledge, we are the first to successfully find minimal eviction sets at the granularity of cache blocks, which means doing the search with totally random addresses. The results are also provided in Table 3 with *granularity* set as

Table 3: The Current Best Results of Finding Minimal Eviction Sets

	Granularity	Multithread	Huge Page	Candidate Size	Repeat Parameter (b, d)	Traverse Function	Retry (rb)	Split Parameter (l)	rollack (rb)	Reuse Failed Set (ru)	TLB Preload	Success rate	Time of a Single Trial	Time to Success
i7-3770	4KB	**N**	N	2700	(4,2)	*random*(16)	48	16	2	0	Y	0.340	0.051s	0.150s
i7-3770	4KB	**N**	**Y**	2700	(4,2)	*random*(16)	48	16	2	0	Y	0.508	0.046s	0.091s
i7-3770	4KB	Y	N	2700	(4,3)	*round*(4)	48	16	2	0	Y	0.646	0.055s	**0.085s**
i7-3770	4KB	Y	N	2700	(4,3)	*round*(4)	48	16	2	50	Y	**0.992**	0.116s	0.117s

ISSN: 2278-4632
Vol-10 Issue-3 No.01 March 2020

i7-3770	4KB	Y	**Y**	2700	(5,3)	*round*(4)	64	16	6	3	Y	0.960	0.058s	0.060s
i7-3770	**64B**	Y	N	162000	(5,7)	*round*(4)	80	16	2	10	Y	0.390	43.98s	1.88m
Xeon-4110	4KB	Y	N	3500	(2,14)	*round*(1)	48	9	2	1	Y	0.758	0.129s	**0.170s**
Xeon-4110	4KB	Y	N	3500	(2,14)	*round*(1)	48	9	2	50	Y	**0.970**	0.320s	0.330s
Xeon-4110	4KB	Y	**Y**	3500	(2,14)	*round*(1)	48	9	2	1	Y	0.760	0.102s	0.134s
Xeon-4110	**64B**	Y	N	281600	(5,18)	*round*(1)	48	9	2	50	Y	0.980	1.70m	1.74m
i7-8700	4KB	**N**	N	3500	(4,3)	*round*(4)	80	14	0	1	Y	0.782	0.158s	0.202s
i7-8700	4KB	**N**	**Y**	3500	(4,3)	*round*(4)	64	14	0	1	Y	0.860	0.106s	0.123s
i7-8700	4KB	Y	N	3500	(4,3)	*round*(1)	64	16	0	2	Y	**0.954**	0.091s	**0.095s**
i7-8700	4KB	Y	**Y**	3500	(4,3)	*round*(1)	64	16	2	2	Y	0.966	0.059s	0.061s
i7-8700	**128B**	Y	N	112000	(4,3)	*round*(4)	64	16	2	10	Y	0.100	63.3s	10.6m

Table 4: Improvement against the State-of-the-Art [26]

			State-of-the-Art [26]			This Paper			Improvement		
Huge Page	Traverse Function	Number of repeat (*b*)	Success rate	Time of a Single Trial	Time to Success	Success rate	Time of a Single Trial	Time to Success	Success rate	Time of a Single Trial	Time to Success
i7-3770N	round	7	0.475	0.226s	0.477s	0.646	0.055s	0.085s	36.0%	−75.7%	−82.1%
i7-3770Y	round	1	0.530	0.116s	0.219s	0.960	0.058s	0.060s	81.1%	−50.0%	−72.6%
i7-8700N	round	1	0.617	0.151s	0.244s	0.954	0.091s	0.095s	54.6%	−39.7%	−61.1%
i7-8700Y	round	1	0.500	0.093s	0.186s	0.966	0.059s	0.061s	93.2%	−36.6%	−67.2%

64B. We have managed to succeed on both i7-3770 and Xeon- 4110, and achieved a surprisingly high success rate of 98% on Xeon-4110. We believe the number of cores on the server level processors (Xeon-4110) contributes to the high success rate because they allow more threads to traverse concurrently compared with the desktop level processors.

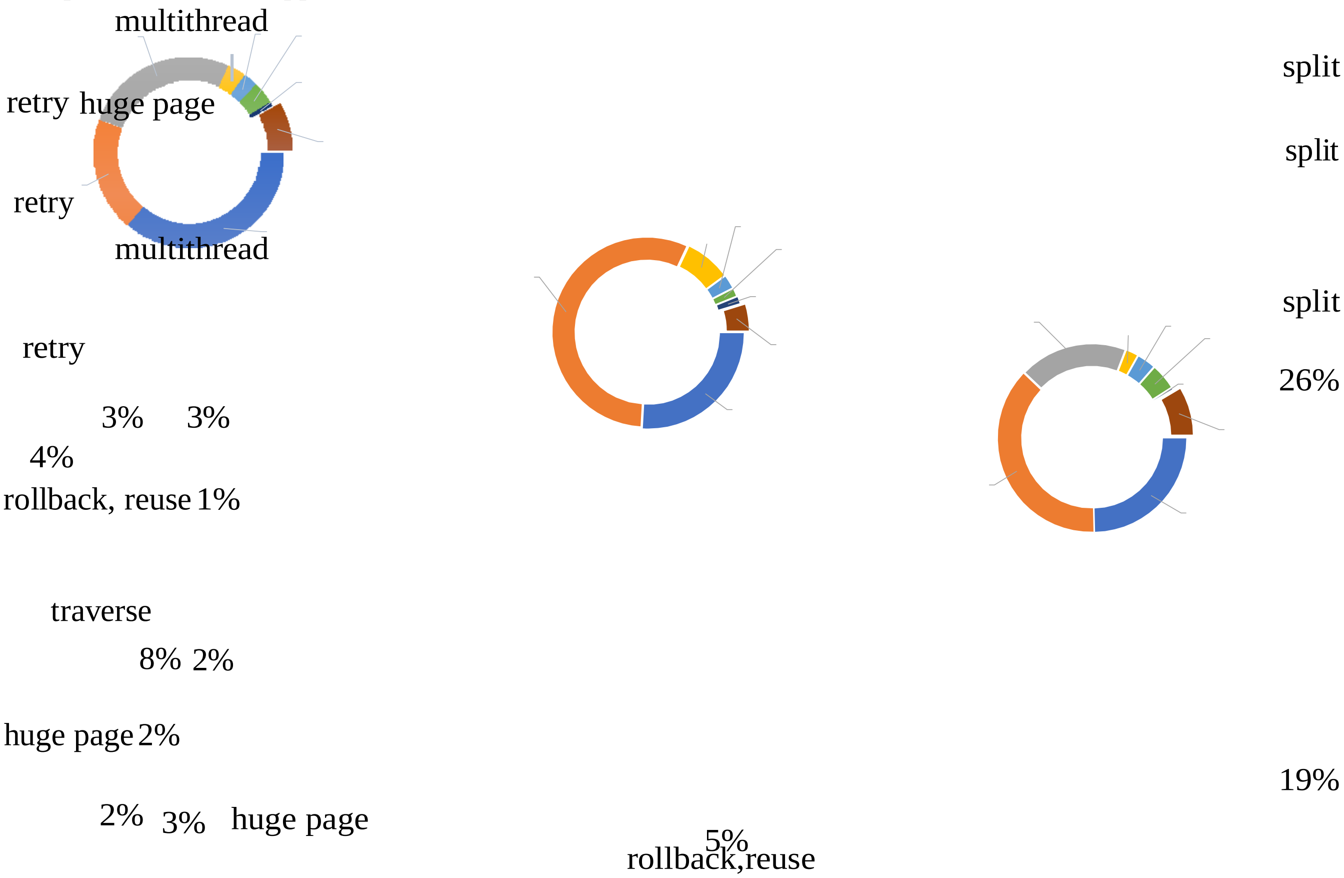
Unfortunately, we failed on i7-8700. The best result is to find eviction sets at the granularity of 128B (2 cache blocks) at a success rate of 10%. The intermediate data show that the pruning algorithm tends to remove the first group of elements (G_1 in Algorithm 3) with abnormally high rates when the size of the candidate set is large. This indicates that the rate of false positive errors in the initial rounds of pruning is too high for the algorithm to tolerate. Although the traverse function *round*(4) achieves the best success rate, the size of the residue sets is rather large (thousands of elements) when it fails. Using *random*(16) results in much smaller residue sets (less than a hundred) but slightly lower success rate and longer latency. We think there might be new changes to the replacement policies inside the latest Coffee Lake processors. The re-accessing of the target address during the eviction test (line 4 in Algorithm 4) might be recognized as a scan leading to its amateur eviction. Improving the techniques in re-accessing the target address and further optimizing traverse functions should bring down the rate of false positive errors. These are our future works.

To evaluate the improvement we have made against the state-of-the-art method [26], we have rerun it on the same platforms used in this paper. Table 4 reveals the comparison results. Since the state-of-the-art method fails to find eviction sets on Xeon-4110, no comparison is made on this platform. On i7-3770 and

ISSN: 2278-4632
Vol-10 Issue-3 No.01 March 2020

i7-8700, evaluations are made both with and without huge pages.

We have manually tested and selected the best traverse function and the best number of repeats (*b*) for both platforms. The size of the initial candidate set is set to 4000, as same as in [26], while all other parameters are set to their optimal val- ues (after manual tuning). Note that both the *success rate* and the *time to success* are significantly better than the reported rate (around 20%) and latency (0.75s) in [26]. According to our discussion with Vila, the reported low success rate was collected when no optimization was applied, while the re-



traverse func 19%

set size 36%

final 8% func

56%

rollback, reuse 1%

final
5%

set size 26%

traverse func 37%

0%

final 9%

set size 25%

(a) i7-3770: from 0.732s to 0.060s (8.2%).

(b) Xeon-4110: from 2.838s to 0.134s (4.7%).

(c) i7-8700: from 0.715s to 0.061s (8.5%).

Figure 13: Contribution of individual optimization techniques. Configuration for the initial case: *granularity*= 4KB, *multithread* disabled (except for Xeon-4110), *huge page* disabled, *candidate size*= 4000, *traverse function*= *list*(1, 1), $(b, d) = (4, 4)$, *retry*= 32, *split parameter*= 32, *rollback* disabled, *reuse failed set* disabled and *TLB preload* enabled. The order of optimization: *candidate size*; *traverse function* and (b, d); *multithread*; *split parameter*; *retry*; *huge page*; *rollback* and *reuse failed set*.ported long latency was due to a very conservative setting of 50 to the number of repeats. The performance boost obtained from optimizing the state-of-the-art method clearly demon- strates that we can significantly reduce the overall latency by tuning parameters.

Even compared with the boosted results of the state-of-the- art method, the improvement achieved by using the techniques proposed in this paper is still significant. As shown in Table 4, *time to success* has been reduced by more than 60% in all scenarios while the success rate has been greatly boosted.

We have also evaluated the contribution of individual opti- mization techniques towards the reduction of *time to success*. The results of all platforms are shown in Figure 13. It is found that the order used in tuning parameters has a strong impact on the contribution. Optimization techniques are not indepen- dent with each other and some techniques are closely related. From our observation, the repeat parameter (b,d) is closely related to the traverse function. They are thus tuned together. Reducing the split parameter too early while the success rate is low might actually hurt performance. The split parameter is also related to the retry parameter. Overall, applying the various optimization methods reduces the *time to success* by more than 90% on all platforms.

We tried to find eviction sets on an AMD Ryzen 3 2200G processor but both ours and the state-of-the-art method [26] failed. According to a recent research [30] on the side-channel attacks on non-inclusvie LLCs, the recent AMD processors are suspected to use a snoopy-based cache coherence protocol. Since evicting a cache block from a non-inclusive LLC does not purge the block from L1 caches, the assumption of $a = w$, as used in Equation 5, no longer holds and none of the pruning algorithms described in this paper can reduce the size of the candidate set to w. Dynamically finding eviction sets targeting non-inclusive LLCs is still an open question, especially when snoopy-based coherence protocols are used.

5 Related Work

Search algorithms: It was found by Hund *et al*. [10] that accessing a large enough collection of virtual addresses is sufficient to evict any data from the LLC. Liu *et al*. [18] and Oren *et al*. [21] proposed the first pruning algorithm which prunes a large eviction set into a minimal one in $O(n^2)$ memory accesses. Eviction sets found in this way were used to construct eviction sets without fully reversing the virtual to physical address mapping [18,20], and launch attacks inside a sandbox [5,7,21] or an SGX enclave [25] without huge pages. Recently Vila *et al*. [26] managed to reduce the upper bound of the pruning algorithm to $O(w^2n)$. We reduce it further to $O(wn)$ in this paper.

Traverse functions: Starting from the Intel IvyBridge, thrashing resistant cache replacement policies are adopted [12,29]. Simply repeatedly and sequentially accessing an eviction set no longer guarantees an eviction. A *dual pointer chase* method was found effective on IvyBridge processors [29]. Gruss *et al*. [7] generalized the method with *eviction strate- gies*. The *round trip* method was first introduced by Liu *etal*. [18]. *Rondom traverse* and *multithread traverse* are intro- duced in this paper.

TLB noise: TLB noise was explained by Genkin *et al*. [5]. They discovered that *preloading the TLB entry* [5] can effec- tively reduce such noise. Also allocating candidate sets from huge pages is effective as well [26].

Existing defenses: *Cache partitioning* is a promising de- fense against cache side-channel attacks [4, 14, 17]. However, it cannot defend the cases when the target and the eviction set cannot be separated by partitions, such as reversing the complex addressing scheme [13], rowhammer inside a sand- box [25], and circumventing the user level or kernel space ASLR [6, 10]. *Randomizing the cache layout* is another type of effective defenses, which hinders the computing of mini- mal eviction sets by randomizing the cache set index. Wang *et al*. [27,28] proposed to apply randomization on the L1 caches, which is ineffective to thwart attacks against the LLC. Qureshi recently proposed CEASER [23], which dynamically random- izes the LLC using a low latency block cipher. If adopted, this would be a strong defense against all conflict-based attacks against the LLC.

6 Conclusion

In this paper, we have reduced the upper bound of the latency in finding minimal eviction sets from $O(w^2n)$

to $O(wn)$ and shown that recent defenses have significantly over-estimated such latency. Practical experiments are produced on three modern processors. Using multiple new techniques proposed in this paper, including using concurrent multithread execu-tion to circumvent the thrashing resistant cache replacement policies, we demonstrate that minimal eviction sets can be found within a fraction of a second on all processors, includ-ing a latest Coffee Lake one. We also show that it is possible to find minimal eviction sets with totally random addresses without fixing the page offset bits.

Acknowledgments

This work was supported by the CAS Pioneer Hundred Tal- ents Program, the National Natural Science Foundation under grant No. 61802402, and internal grants from the Institute of Information Engineering, CAS. The authors would like to thank the anonymous reviewers for their invaluable comments, Kaveh Razavi for kindly shepherding this paper, and Pepe Vila for a detailed explanation of his paper. The authors also express gratitude to Wenhao Wang, Xiaoxin Li, Xiaofei Fu and Yifei Sun for discussions related to cache randomization and cache eviction.

Availability

The ideal cache model described in in Section 3.1 can be accessed at https://github.com/comparch-security/cache-model. The test programs on actual processors can be accessed at https://github.com/comparch-security/smart-cache-evict.

References

[1] MFSA 2018-01: Speculative execution side-channel at- tack ("Spectre"), Jan. 2018. https://www.mozilla.org/en-US/security/advisories/mfsa2018-01/.

[2] Andreas Abel and Jan Reineke. Reverse engineering of cache replacement policies in Intel microprocessors and their evaluation. In *Proceedings of the IEEE Interna- tional Symposium on Performance Analysis of Systems and Software (ISPASS'14)*, pages 141–142. IEEE Com-puter Society, Mar. 2014.

[3] Christoph Berg. PLRU cache domino effects. In *Proceedings of the International Workshop on Worst- Case Execution Time Analysis (WCET'06)*, Dagstuhl, Germany, 2006. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. http://drops.dagstuhl.de/opus/volltexte/2006/672.

[4] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael B. Abu-Ghazaleh, and Dmitry Ponomarev. Non- monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Transactions on Archi- tecture and Code Optimization*, 8(4):35:1–35:21, 2012.

[5] Daniel Genkin, Lev Pachmanov, Eran Tromer, and Yu- val Yarom. Drive-by key-extraction cache attacks from portable code. In *Proceedings of the International Con- ference on Applied Cryptography and Network Security (ACNS'18)*, pages 83–102. Springer, 2018.

[6] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the line: Practical cache attacks on the MMU. In *Proceedings of the Network and Distributed System*

Security Symposium (NDSS'17). Internet Society, 2017.

[7] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A remote software-induced fault attack in JavaScript. In *Proceedings of the Conference on Detection of Intrusions and Malware, and Vulnerabil- ity Assessment (DIMVA'16)*, pages 300–321. Springer, 2016.

[8] Marcus Hähnel, Weidong Cui, and Marcus Peinado. High-resolution side channels for untrusted operating systems. In *Proceedings of the USENIX Annual Tech- nical Conference (ATC'17)*, pages 299–312. USENIX Association, Jul. 2017.

[9] Lars T. Hansen. Shared memory and atomics for EC- MAscipt, Feb. 2017. https://github.com/tc39/ecmascript_sharedmem.

[10] R. Hund, C. Willems, and T. Holz. Practical timing side channel attacks against kernel space ASLR. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'13)*, pages 191–205. IEEE, May 2013.

[11] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$a: A shared cache attack that works across cores and defies VM sandboxing – and its application to AES. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'15)*. IEEE, May 2015.

[12] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely Jr., and Joel S. Emer. High performance cache replacement using re-reference interval prediction (RRIP). In *Pro- ceedings of the International Symposium on Computer Architecture (ISCA'10)*, pages 60–71. ACM, Jun. 2010.

[13] Mehmet Kayaalp, Nael Abu-Ghazaleh, Dmitry Pono- marev, and Aamer Jaleel. A high-resolution side- channel attack on last-level cache. In *Proceedings of the Annual Design Automation Conference (DAC'16)*. ACM Press, 2016.

[14] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTHMEM: system-level protection against cache- based side channel attacks in the cloud. In *Proceedings of the USENIX Security Symposium (Security'12)*, pages 189–204. USENIX Association, Aug. 2012.

[15] Paul Kocher, 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 *Proceed- ings of the IEEE Symposium on Security and Privacy (S&P'19)*, pages 19–37, May. 2019.

[16] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémen- tine Maurice, and Stefan Mangard. ARMageddon: Cache attacks on mobile devices. In *Proceedings of the USENIX Security Symposium (Security'16)*, pages 549–564. USENIX Association, Aug. 2016.

[17] Fangfei Liu, Qian Ge, Yuval Yarom, Frank McKeen, Car-los V. Rozas, Gernot Heiser, and Ruby B. Lee. CATalyst: Defeating last-level cache side channel attacks in cloud computing. In *Proceedings of the International Sym- posium on High Performance Computer Architecture (HPCA'16)*, pages 406–418, 2016.

[18] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *Proceedings of the IEEE Symposium on Security and Privacy*

(S&P'15). IEEE, May 2015.

[19] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Re- verse engineering Intel last-level cache complex address- ing using performance counters. In *Proceedings of the International Symposium on Research in Attacks, Intru-sions, and Defenses (RAID'15)*, pages 48–65. Springer, 2015.

[20] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the other side: SSH over robust cache covert channels in the cloud. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS'17)*, Mar. 2017.

[21] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadha- van, and Angelos D. Keromytis. The spy in the sandbox. In *Proceedings of the ACM SIGSAC Conference on Com- puter and Communications Security (CCS'15)*. ACM Press, 2015.

[22] Colin Percival. Cache missing for fun and profit, 2005. https://papers.freebsd.org/2005/cperciva-cache_missing/.

[23] Moinuddin K. Qureshi. CEASER: Mitigating conflict- based cache attacks via encrypted-address and remap- ping. In *Proceedings of the Annual IEEE/ACM Inter- national Symposium on Microarchitecture (Micro'18)*, pages 775–787. IEEE, 2018.

[24] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic timers and where to find them: High-resolution microarchitectural attacks in JavaScript. In *Proceedings of the International Con- ference on Financial Cryptography and Data Security (FC'17)*, pages 247–267. Springer, Jan. 2017.

[25] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clé- mentine Maurice, and Stefan Mangard. Malware guard extension: Using SGX to conceal cache attacks. In *Pro- ceedings of International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'17)*, pages 3–24. Springer, Jul. 2017.

[26] Pepe Vila, Boris Köpf, and Jose Morales. Theory and practice of finding eviction sets. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'19)*. IEEE, 2019.

[27] Zhenghong Wang and Ruby B. Lee. New cache de- signs for thwarting software cache-based side channel attacks. In *Proceedings of the International Symposium on Computer Architecture (ISCA'07)*, pages 494–505. ACM, Jun. 2007.

[28] Zhenghong Wang and Ruby B. Lee. A novel cache architecture with enhanced performance and security. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'08)*, pages 83–93. IEEE, 2008.

[29] Henry Wong. Intel Ivy Bridge cache replacement policy, Jan. 2013. http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/.

[30] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrel- las. Attack directories, not caches: Side-channel attacks in a non-inclusive world. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'19)*. IEEE, 2019.

[31] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel

w elements are found irremovable.

n

$n-w$

attack. In *Proceedings of the USENIX Security Sympo-sium (Security'14)*, pages 719–732. USENIX Associa-

$\sum_{i=w+1}$

$i+w^2 \quad T(n) \quad wn+\sum(i+w)$

$i=1$ $\leq$ $\leq$ n n

tion, 2014.

$\sum i < T(n) < \sum n$

[32] Xiaokuan Zhang, Yuan Xiao, and Yinqian Zhang. Return-oriented flush-reload side channels on ARM and

$i=1$

$\frac{n^2+n}{2}$

$i=1$

their implications for Android devices. In *Proceedings of the ACM SIGSAC Conference on Computer and Com- munications Security (CCS'16)*, pages 858–870. ACM Press, Oct. 2016.

[33] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS'12)*, pages 305–316. ACM, Oct. 2012.

A Proof of Propositions

Proposition 4. Algorithm 1 prunes a candidate set of n ele-ments into a minimal eviction set in $O(n^2)$ memory accesses:

$$2 \quad < T(n) < n$$

Since $n \quad w > 1$, $T_{\text{base}}(n) \sim O(n^2)$. □

Proposition 5. Assuming $l > w$, the number of memory ac- cesses using Algorithm 2 to prune a candidate set of n ele- ments has an upper bound:

$$T_{\text{split}}(n) < \frac{l(l-1)n}{l-w}$$

Proof. In the worst case, the w elements of the minimal evic- tion set is evenly distributed in w of the l groups; therefore, only $l-w$ groups are removed in each iteration. $l-1$ groups are tested in the worst case because only $l-w-1$ groups are

$-$

Tbase

$(n) \sim \ O(n^2)$

found removable in the l 1 tests. The upper bound of T(n) can be calculated as:

Proof. For a candidate set with n elements, at most one ele-ment is removed in each iteration of the **foreach** loop. In total $n-w$ elements are removed. The minimal number of memory

$T(n) \leq (l-1)n + (l-1)n\frac{w}{l} + \cdots + (l-1)n(\frac{w}{l})^k$

k i

$-$ $\frac{w}{l}$

accesses is reached when one element is removed for the first $n-w$ iterations while the maximal is reached when the first

$= (l-1)n\sum(\ )$

$i=0$

where the termination condition is $n(\frac{w}{l})^{k+1} \leq w$.

$$T(n) \leq (l-1)n \frac{1-(\frac{w}{l})^{k+1}}{1-\frac{w}{l}} \leq \frac{l(l-1)n}{l-w}$$