Feasibility Analysis and Performance Optimization of the Conflict Test Algorithms for Searching Eviction Sets

Zhenzhen Li[1,2], Zihan Xue[1,2], and Wei Song[1,2]

[1] Institute of Information Engineering Chinese Academy of Sciences, Beijing, China
[2] School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China
{lizhenzhen1,xuezihan,songwei}@iie.ac.cn

Abstract. Cache side-channel attacks have been widely utilized as an intermediate step in some comprehensive attacks. Eviction sets, especially the minimal eviction sets, are essential components of the conflict-based cache side-channel attacks. It is important to develop efficient search algorithms that incur the lowest latency with the highest success rate. Several fast search algorithms have been proposed in recent years, among which conflict test (CT) achieves the highest success rate with the lowest latency. In this paper, we have conducted the first systematic feasibility analysis of the CT algorithm. Besides failing on the commonly known cache architectures where the last-level cache (LLC) is exclusive or non-inclusive, CT is also found and verified failing on two inclusive LLC architectures if it is running in single-core mode. We have further explored three optimizations for improving the speed performance of the CT algorithm, two of which are newly proposed in this paper.

Keywords: Computer micro-architecture · Cache architecture · Cache side-channel attack · Eviction set construction

1 Introduction

As an effective way of obtaining sensitive information from the cache system [10, 18, 24], cache side-channel attacks have been widely utilized as an intermediate step in some comprehensive attacks, such as reconstructing cryptographic keys [1, 6, 11, 29, 30, 37], disarming the address space randomization [7, 8] in control-flow attacks, retrieving the leaked information at the end of a transient execution attack [14, 15], and constantly striking a row of the off-chip memory in a rowhammer attack [9].

Eviction sets, especially the minimal eviction sets [32], are essential components of the conflict-based cache side-channel attacks [34]. In such attacks, an attacker and her victim share the same cache space, typically certain cache sets in the last-level cache (LLC). The attacker needs to control the state of these shared cache sets to monitor the memory accesses of her victim, which are then used to infer security-critical information. To be specific, the attacker occupies (primes) a cache set by accessing an eviction set [9]; therefore, her victim's access

to this cache set must incur a cache miss, refilling of the missing cache block, evicting an address from the eviction set, and eventually a prolonged access. Both the address eviction and the prolonged access latency might be observable and used to infer the access of her victim.

All addresses in a minimal eviction set are *congruent* with (mapping to) the targeted cache set [32]. At least W addresses are required for a W-way set-associative cache. Obviously, the key for constructing an eviction set is to find enough congruent addresses. Unfortunately, this is not an easy task on modern processors. LLC is indexed by physical addresses but attackers control only virtual addresses. A complex addressing scheme is utilized by modern Intel processors [17] to randomize the mapping from physical addresses to LLC slices. Attackers are usually forced to search eviction sets at runtime from a large amount of random addresses. It is important to develop efficient search algorithms that incur the lowest latency with the highest success rate. Several fast search algorithms have been proposed in recent years, including *group elimination* (GE) [16, 27, 32], *prime, prune and probe* (PPP) [19, 22], *conflict test* (CT) [23] and *write-after-write* (W+W) [28]. Among these algorithms, CT achieves the highest success rate with the lowest latency (see Table 1), and becomes one of the most widely utilized search algorithms [20, 21]. However, there lacks a systematic analysis on the feasibility and the potential optimization of CT while similar analyses have been done for GE [27] and PPP [19].

In this paper, we have conducted the first systematic feasibility analysis of the CT algorithm. Besides sharing a commonly known limitation with other algorithms, that CT fails to work on exclusive or non-inclusive LLCs, CT also fails on two inclusive LLC architectures if the algorithm is running in single-core mode. Based on the result of the feasibility analysis, we have further explored three techniques for further optimizing the CT algorithm, two of which are newly proposed in this paper. Overall, this paper makes the following contributions:

- Conduct a systematic feasibility analysis on CT. For the first time, two inclusive cache architectures are identified as infeasible for single-core CT.
- Optimize the performance of CT by improving the efficiency of the cacheback technique and propose two new techniques.
- Practically evaluate the optimization techniques on both real processors and a behavioral-level cache model.

2 Background

Modern processors are multicore processors adopting a two/three-level cache structure. Each processing core contains a pair of private level-one (L1) instruction and data caches. Some processors, especially the Intel ones, equip each core with a uniformed level-two (L2) cache. A large LLC (L2 or L3) is shared among all cores. This LLC might be divided into multiple slices, whose mapping with physical addresses is decided by an undisclosed hash function (complex addressing scheme [17]) in Intel processors. All levels of caches are set-associative

writeback allocated caches. According to [31], all cache levels in the early generations (Haswell and earlier) and the L1 caches in recent Intel processors utilize the pseudo-LRU (PLRU) replacement policy [5], while L2 and LLC in recent Intel processors adopt some policies derived from RRIP [13]. The situation is similar for most other commercial processors, such as AMD ones. In some rare cases, random replacement policy is used in embedded-level processors [26]. In all cache architectures, LLC acts as the coherence hub. LLC and the private L1/L2 caches maintain either an inclusive relation (Intel's consumer processors), where all cache blocks in the private caches are also stored in the LLC, or a non-inclusive relation (Intel's Xeon and AMD's Ryzen), where cache blocks stored in private caches may not be concurrently stored in the LLC.

Cache side-channel attacks normally fall in two categories: *flush-based* and *conflict-based* attacks. Flush-based attacks use explicit flush instructions (`clflush` on x86 [36]) to invalidate a targeted data out of the cache architecture. These attacks are accurate but require the targeted data is accessible by the attacker, which is a rather strict requirement infeasible in most cross-process side-channel attacks. As an alternative, conflict-based attacks can achieve the similar effect. They evict the targeted data out of the LLC by occupying the corresponding LLC cache set with a collection of attacker's controlled cache blocks, typically called an *eviction set*. An eviction set is a collection of addresses (cache blocks) that contain enough addresses congruent with the targeted data. A sufficiently large number of addresses are also an eviction set as they can evict any cache block by priming the whole caches [33]. However, this type of untargeted eviction introduces undesirable noise [9] and brings down the attack speed [7]. What is really desirable is a minimal eviction containing only the congruent addresses. For simplicity, an "eviction set" beyond this point refers to a minimal eviction set. This paper concentrates on the algorithms for searching eviction sets.

Existing search algorithms for eviction sets can be classified into two categories: *pruning algorithms* which begin with an untargeted eviction set containing a large number of random addresses and prune it into a minimal one, and *inserting algorithms* which begin with an empty collection and gradually fill it with newly found congruent addresses until it becomes an eviction set.

GE and PPP are the two widely utilized pruning algorithms. GE prunes the initial large eviction set in a multi-round process. In each round, the remaining N addresses are divided into $W + 1$ groups. Since a minimal eviction set contains only W addresses, at least one group contains none of the W addresses and should be removed. By sequentially testing whether the address collection remains an eviction set without a certain group, the removable group is found and removed. The prune process continues until a minimal set is produced. GE is robust in tolerating environment noise, as indicated by the high success rate shown in Table 1, but the multi-round prune is slow.

PPP reduces the prune latency by manipulating the PLRU replacement policy [22, 23]. It first tries to store addresses of the initial large eviction set into the LLC concurrently by gradually removing the addresses causing self-evictions. The resulted (reduced) eviction set is still untargeted but may fully occupy the

Table 1. Speed comparison of different search algorithms for eviction sets.

CPU	GE		PPP		W+W		CT	
	latency	rate	latency	rate	latency	rate	latency	rate
i7-3770	58 ± 32ms	74%	0.69 ± 1.7ms	8.8%	33 ± 41ms	6.1%	6.0 ± 3.4ms	69%
i7-6700	82 ± 66ms	79%	1.0 ± 2.9ms	0.9%	10 ± 5.3ms	5.3%	23 ± 21ms	16%
i7-9700	115 ± 92ms	85%	0.65 ± 0.68ms	11%	159 ± 8.5ms	2.0%	20 ± 17ms	21%
i7-11700	642 ± 586ms	24%	0.81 ± 0.04ms	7.0%	3 ± 1.4ms	0.4%	12 ± 4.4ms	2.1%

targeted cache set. Then the attacker incurs an eviction in the targeted cache set by accessing the targeted address, following with a timed re-access of the reduced eviction set. Due to the PLRU replacement policy, it is likely that exactly W addresses (just enough for an eviction set) are found missing in the LLC. However, the probability that the reduced eviction set occupying the targeted cache set is actually low in a large LLC with many cache sets. As shown in Table 1, the success rate of PPP is much lower than GE.

CT is the mostly utilized inserting algorithm. It was initially proposed only for LLCs adopting the random replacement policy [23]. In this case, a congruent address has a $1/W$ probability to evict the targeted cache block. As a random address is a congruent address by a probability of $1/S$, where S is the number of cache sets, one congruent address can be found by probing around SW random addresses. Finding eviction set with W congruent addresses therefore requires probing $\mathcal{O}(SW^2)$ random addresses. This algorithm is also effective for permutation-based replacement, such as LRU and RRIP. Instead of finding congruent addresses by detecting the eviction of the targeted cache block, detecting the prolonged write latency due to the LLC enforced serialization of parallel writes to the same cache set was also found effective [28]. The resulted algorithm, namely $W+W$, was claimed faster than the GE algorithm. However, the accuracy of such serialization detection is found extremely noisy and unstable, which results in low success rates as shown in Table 1.

To compare the speed performance of these algorithms, they are ported to four Intel processors and the result is shown in Table 1. CT seems to provide the most balanced performance in latency and success rate. The search latency is significantly lower than GE while the success rate is much higher than PPP (except for i7-11700). The search latency of W+W is shorter than CT only on i7-6700 and i7-11700 but the success rate is much lower on both processors. This paper concentrates on improving the CT algorithm.

3 Feasibility Analysis

This section conducts a systematic analysis on the feasibility of the CT algorithm on different cache architectures. For the first time, the CT algorithm is found infeasible on two inclusive cache architectures.

3.1 Threat Model

For an eviction set search algorithm, we define a successful attack as finding an eviction set. We assume that the search algorithm is run by an attacker in a restricted user mode environment with the following capabilities and limitations:

- The targeted LLC is shared between the attacker and her victim.
- The amount of memory acquirable by the attacker is not limited by the system, so the attacker can access an arbitrarily large range of addresses.
- The attacker either runs in the same core with her victim or occupies a separate core.
- The attacker can flush her own data out of the LLC.
- The attacker can accurately trick her victim into accessing a target address without incurring a large amount of noise.
- Some parameters regarding the cache system are made available, such as the replacement policy, the inclusiveness relation, and the number of sets and ways of each cache level, but neither the virtual to physical page mapping nor the Intel complex addressing scheme [17] is reverse-engineered.

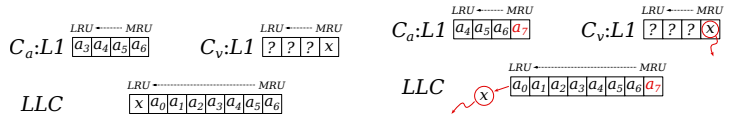
3.2 Necessary Working Conditions

Algorithm 1 illustrates the baseline CT algorithm. Different with other papers [20, 23], we explicitly specify the cores running the victim and the attacker. When $C_a = C_v$, the attacker and her victim are running on the same core or even in the same process/thread. This is the typical case for cache side-channel attacks that tries to break the user-mode address randomization [8], leak information through transient execution [14, 15], and constantly hammer a targeted DRAM row [9]. We call this the single-core case while the traditional cross-core (process) attack as the cross-core case. As we will soon discover in Section 3.3, CT may fail to work on some inclusive cache architectures when running in the single-core case while remains feasible for cross-core.

According to Algorithm 1, a random address a is found congruent with the targeted address x only if accessing a (line 6) causes a miss in the targeted cache

Algorithm 1: The baseline CT algorithm

```
   Input: x, target address; W, number of ways; (C_a, C_v), cores running the attacker and
          her victim.
   Output: E, an eviction set for x.
 1 function ct(x, W, C_a, C_v)
 2     E ← ∅ // eviction set
 3     C_v:access(x)
 4     while |E| < W do
 5         a ← random()
 6         C_a:access(a)
 7         if not C_v:probe(x) then
 8             E ⋃ {a}
 9         end
10     end
11     return E
12 end
```



(a) Before accessing a_7. (b) After accessing a_7.

Fig. 1. Purging x (cross-core case) after accessing a_7 by C_a in a 2-level inclusive cache architecture. ($W_{\text{L1}} = 4, W_{\text{LLC}} = 8$, all caches use LRU)

set and the cache block containing x is evicted for refilling a. In addition, the eviction of x can be observed by probing x (code highlighted in blue): a timed access of x. If the probe latency is longer than a pre-defined threshold, x is assumed missing and a is identified as congruent. Two necessary conditions for the success of CT can be derived from Algorithm 1:

Condition 1: *Inclusion victim effect.* When an LLC is the targeted cache, the targeted cache block stored in a private L1 cache (the potential inclusion victim [12]), such as the x stored in C_v:$L1$ depicted in Fig. 1a, must be purged from the cache architecture when its copy in the LLC is evicted due to a conflict, such as the access of a_7 by C_a shown in Fig. 1b. In other words, CT works only when the targeted LLC is inclusive. Note that this condition is required for the single-core case as well, since x is also purged by a conflict in the LLC.

Condition 2: *Cache filter effect.* When the CT algorithm is used to target an LLC adopting LRU/RRIP replacement policies, the probing of x is observed by the LLC only after x is successfully evicted in the LLC, such as probing x after accessing a_7 as shown in Fig. 1b. The cache filter effect is a by-product of the hierarchical cache architecture where memory accesses hitting in private caches are invisible to the LLC. When the LLC adopts PLRU/RRIP replacement policies, the target address x is possible to be evicted by a fresh access of a new random address a only when x is pushed to the LRU position, as shown in Fig. 1a, by a number of accesses (random addresses) to the cache set after the previous access of x is observed by the LLC. According to Algorithm 1, x is accessed once in the probe for each random address. All of these accesses must be filtered from the LLC (served by private L1/L2 caches); otherwise, x is repeatedly accessed in the LLC and cannot be pushed to the LRU position. This is the first time that such condition has been discovered and we will show in the next section (Section 3.3) why CT fails on some inclusive cache architectures (satisfying condition 1) due to the lack of this cache filter effect.

3.3 Feasibility on Different Cache Architectures

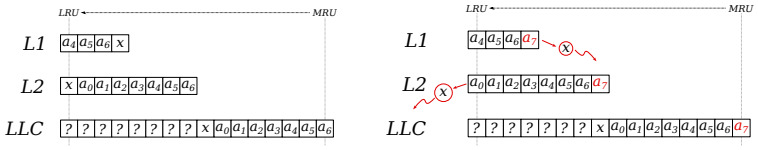
Utilizing the two necessary conditions discovered in Section 3.2, we have conducted a systematic survey on the feasibility of CT on different cache architectures. We consider the following cache parameters:

- **Cache levels**: cache architectures that have two or three levels of caches.
- **Inclusiveness**: inclusive ($L1 \subseteq LLC$), exclusive ($L1 \neq LLC$) or non-inclusive ($L1 \nsubseteq LLC$) relation between cache levels.

Table 2. Feasibility on different cache architectures.

Architecture	Example	Attack	Feasible
Exclusive or Non-inclusive LLC[a].	AMD Zen 2 and later (Ryzen-7 5700G)	cross-core single-core	No No
Inclusive LLC[b] with private caches using LRU/RRIP.	Intel Processors (i7-6700 and Xeon 4110[b])	cross-core single-core	Yes Yes
Three levels of inclusive caches using LRU/RRIP.	Early quad/hexa-core processors (Intel Dunnington [2, 25])	cross-core single-core	Yes No
Inclusive LLC using LRU/RRIP with private caches using random.	A customized Rocket-Chip processor (Section 5.1)	cross-core single-core	Yes No

[a] A non-inclusive LLC may adopt an inclusive directory and CT becomes feasible, such as the Intel Xeon processors [35]. These cache architectures are counted as inclusive LLCs without differentiating the directory from the cache.
[b] Include the non-inclusive LLCs adopting inclusive directories.



(a) Before accessing a_7. (b) After accessing a_7.

Fig. 2. A failing example of single-core CT in a 3-level inclusive cache using LRU. ($W_{\mathrm{L1}} = 4, W_{\mathrm{L2}} = 8, W_{\mathrm{LLC}} = 16$)

- **Cache sets and ways**: when the LLC is inclusive, it is assumed that the number of ways in the LLC (W_{LLC}) is no less than it in in private caches: $W_{\mathrm{LLC}} \geq W_{\mathrm{L2}}$ if $L1 \subseteq L2$ or $W_{\mathrm{LLC}} \geq W_{\mathrm{L1}} + W_{\mathrm{L2}}$ otherwise.
- **Replacement policy**: the replacement policy of individual cache can be independently selected among LRU, RRIP or random.
- **Attack scenario**: both cross-core and single-core attacks are considered.

In total, we have surveyed 168 different cache architectures (scenarios) and identified four categories of representative cache architectures as revealed in Table 2:

Exclusive or Non-inclusive LLC: When the LLC is exclusive or non-inclusive, the target x stored in L1 cannot become an inclusion victim and CT fails to work. Nearly all recent AMD Zen 2 and later processors fall in this category and are naturally immune to the CT algorithm. Intel Xeon processors adopt a non-inclusive LLC but utilize an inclusive directory. Due the inclusiveness of the directory, they are still vulnerable to CT. We count them as inclusive LLCs.

Inclusive LLC with private caches using LRU/RRIP: This is the common category for nearly all Intel processors. The inclusive LLC ensures the inclusion victim effect. As for the cache fitler effect, since LRU/RRIP is adopted by the private caches, the repeatedly probing of x ensures that x is pinned in the L1 and all accesses to x are invisible to LLC until x is evicted from the LLC. Note that we deliberately leave an exception here for simplicity. As described by the next category, a three-level inclusive LLC using LRU/RRIP can break the condition for the cache filter effect.

Three levels of inclusive caches using LRU/RRIP: This is an exception of the previous category. CT works in the cross-core case but fails in the

single-core case due to the lack of the filter effect. A failing example is presented in Fig. 2. After accessing seven congruent addresses (a_0 to a_6) and probing x, the state of the three-level cache is depicted in Fig. 2a. Note the repeated probing of x is filtered by L1 and invisible to both L2 and LLC. As a result, x is pushed to the LRU position in L2. As demonstrated in Fig. 2b, the following access of a_7 thus evicts x from L2, which consequently purges x also from L1 as it is an inclusion victim. As x is purges from both L1 and L2, the probing of x is observed by LLC, which moves x to the MRU position in LLC and CT fails. The rooting cause is that the probing of x is invisible to the inclusive L2 while $W_{\text{L2}} < W_{\text{LLC}}$. Early generations of the Intel quad/hexa-core multiprocessors, such as the Intel Dunnington architecture [2, 25] adopts such a three-level inclusive cache architecture. The L2 cache in later generations becomes exclusive, which unfortunately makes them vulnerable to single-core CT.

Inclusive LLC using LRU/RRIP with private caches using random: This architecture is uncommon as most L1 caches adopt LRU/RRIP replacement policies. However, the single-core CT fails in such an architecture as x is likely evicted from the private caches before it is evicted from the LLC due to the random replacement, which makes the following probing of x observed by the LLC. CT therefore fails due to the lack of the filter effect. In Section 5.1, we have configured the cache architecture of a dual-core Rocket-Chip accordingly as a demonstrative example for the failing of single-core CT.

4 Performance Optimization

This section begins with a performance analysis of the baseline CT algorithm. Based on the analysis, three optimization techniques are proposed to improve the efficiency of the CT algorithm.

4.1 Performance Analysis of the Baseline Algorithm

Let us consider a cross-core attack on a two-level inclusive cache using the LRU replacement policy. The latency (L) of searching one eviction set of W congruent addresses can be estimated as:

$$L = (N_{\text{RA}} + W) \cdot t_{\text{mem}} + (N_{\text{v}} - W) \cdot t_{\text{L1}} + N_{\text{v}} \cdot \Delta_{\text{cross}} \tag{1}$$

where N_{RA} and N_{v} are the numbers of accessing random addresses and the victim address x, respectively, while t_{mem}, t_{L1} and Δ_{cross} are the time for one memory access, the time for one access hitting in L1, and the time overhead for one cross-core access, respectively. The total number of LLC misses is $N_{\text{RA}} + W$ and $N_{\text{v}} - W$ times of probing x should hit in L1 due to the perfect filter effect.

Due to the LRU replacement policy, the target address x is evicted from the LLC every time when W congruent random addresses are accessed. A total of W^2 congruent random addresses are searched before obtaining an eviction set.

We call this number N_{CA}. Since random address is a congruent address with x by a probability of $1/S$, N_{RA} and N_{v} can be estimated as:

$$N_{\mathrm{RA}} = N_{\mathrm{v}} = N_{\mathrm{CA}} \cdot S = SW^2 \tag{2}$$

where S is the number of LLC sets. Using Equation 1, L is rewritten to:

$$L = (SW^2 + W) \cdot t_{\mathrm{mem}} + (SW^2 - W) \cdot t_{\mathrm{L1}} + SW^2 \cdot t_{\mathrm{cross}} \tag{3}$$

$$= SW^2 \cdot [t_{\mathrm{mem}} + (t_{\mathrm{L1}} + \Delta_{\mathrm{cross}})] + W \cdot (t_{\mathrm{mem}} - t_{\mathrm{L1}}) \tag{4}$$

$$= S \cdot N_{\mathrm{CA}} \cdot (t_{\mathrm{mem}} + t_{\mathrm{v}}) + W \cdot \Delta_{\mathrm{miss}} \tag{5}$$

where t_{v} and Δ_{miss} are the time for one (cross-core) probing of x and the time overhead of one cache (both L1 and LLC) miss, respectively. According to Equation 5, the key for reducing L is to decrease N_{CA}, the number of congruent random addresses requiring to be accessed, as all others are constants.

Equation 5 holds true for cross-core attacks on all feasible cache architectures, even when the LLC adopts the random replacement policy. In this case, Equation 2 remains the same as a random address is a congruent address with x by a probability of $1/S$, accessing a congruent address evicts x by a probability of $1/W$, and x is evicted for W times during the whole search. Consequently, Equation 4 and 5 remain untouched

For single-core attacks, Equation 5 remains valid as long as the L1 adopts LRU/RRIP replacement policies because LRU/RRIP guarantees the perfect filter effect. t_{v} is reduced to t_{L1} as the cross-core overhead is removed. When both L1 and LLC adopt the random replacement policy, accessing a random address evicts x from the L1 cache by a probability of $1/(S_{\mathrm{L1}} \cdot W_{\mathrm{L1}})$. Therefore, extra latency is introduced in Equation 4 and 5:

$$L = SW^2 \cdot (t_{\mathrm{mem}} + t_{\mathrm{L1}}) + W \cdot (t_{\mathrm{mem}} - t_{\mathrm{L1}}) + \frac{SW^2}{S_{\mathrm{L1}} \cdot W_{\mathrm{L1}}} \cdot (t_{\mathrm{LLC}} - t_{\mathrm{L1}}) \tag{6}$$

$$= S \cdot N_{\mathrm{CA}} \cdot (t_{\mathrm{mem}} + t_{\mathrm{v}}) + W \cdot \Delta_{\mathrm{miss}} + \frac{S \cdot N_{\mathrm{CA}}}{S_{\mathrm{L1}} \cdot W_{\mathrm{L1}}} \cdot \Delta_{\text{L1-miss}} \tag{7}$$

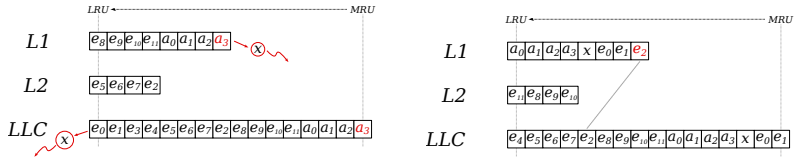
where $t_{\mathrm{v}} = t_{\mathrm{L1}}$ and $\Delta_{\text{L1-miss}} = t_{\mathrm{LLC}} - t_{\mathrm{L1}}$, which is the time overhead of accessing LLC when L1 misses. Similarly, the key for reducing L is to decrease N_{CA} as all others are constants.

4.2 *Cacheback*: Reducing the Number of Random Accesses

Cacheback is an optimization capable of reducing N_{CA} when the LLC adopts an LRU/RRIP replacement policy. In the baseline CT algorithm, every time the target address x is evicted from the LLC, a total of W congruent addresses are accessed but only the last one is identified by the algorithm, because it finally evicts x. When a number of congruent addresses are identified and stored in $\mathcal{E}$ (line 8 in Algorithm 1), these addresses can be used to push x to the LRU position and reduce the total number of congruent addresses (N_{CA}) needed in the

Algorithm 2: Cacheback after a successful probe

```
1 if not C_v:probe(x) then
2     E ⋃ {a}
3     for e in E do
4         C_a:access(e)
5     end
6 end
```



(a) After accessing a_3. (b) After accessing e_2.

Fig. 3. Problem of cacheback when the order observed by LLC (L2) mismatching with the program order. (According to i7-6700, $W_{\text{L1}} = 8, W_{\text{L2}} = 4, W_{\text{LLC}} = 16$, L2 is exclusive, L2 and LLC adopt RRIP)

search. This cacheback procedure is described in the code extraction of the probe illustrated in Algorithm 2 (replacing the code highlighted blue in Algorithm 1) with the optimization highlighted in red. After the i-th congruent address (e_i) is identified by CT, the number of congruent addresses needed for identifying the next one is reduced to $W - i$. Therefore, N_{CA} is reduced to:

$$N_{\text{CA}} = \sum_{i=0}^{W-1} (W - i) = \frac{W^2 + W}{2} \tag{8}$$

Compared with Equation 2, the total number of congruent addresses needed in the search is roughly reduced by half, so does the search latency.

This optimization is first proposed in the Prime+Scope attack [20]. By further investigation, we find out that the optimization works but not as effective as it should be. There are two reasons for this reduced efficiency: *mismatching access order* and *broken filter effect*. Let us consider an example of single-core attack on a three-level cache depicted in Fig. 3. The access order observed by the LLC might not match with the access order issued by the program. As a result, when the target address x is evicted by accessing address a_3 in Fig. 3a, the access order observed (more importantly the replacement order) by L2 and LLC mismatches with the program order for address e_2,[3] assuming 12 congruent addresses (e_0 to e_{11}) have been identified, stored in $\mathcal{E}$, and used for cacheback. a_3 is identified as a congruent address and stored in $\mathcal{E}$ after probing x (refill x to L1 and LLC as well). According to the cacheback optimization, e_0 to e_{11}, along with a_3 are accessed

[3] Many reasons can cause the mismatch in access order. The filter effect itself is a potential cause as soon shown in Fig. 3b. The imperfect pseudo-LRU used in hardware and the RRIP derivative policies used in L2 and LLC [31] also cause mismatching replacement order and access order. Finally, the L2 in this case (also in modern Intel processor) is exclusive, whose replacement order is also affected by the block swapping between L2 and L1 when a block hits in L2.

Algorithm 3: Flush before cacheback

```
1  if not C_v:probe(x) then
2      E ⋃ {a}
3      for e in E do
4          C_a:flush(e)
5      end
6      for e in E do
7          C_a:access(e)
8      end
9  end
```

Algorithm 4: Interleavedly re-access target during cacheback

```
1  if not C_v:probe(x) then
2      E ⋃ {a}
3      for e in E do
4          C_a:flush(e)
5      end
6      for e in E do
7          C_a:access(e)
8          C_v:access(x) // single-core, C_v = C_a
9      end
10 end
```

to push x towards the LRU position in LLC. When the cacheback proceeds to e_2, this address hits in L2 and is swapped to L1 rather than accessing from LLC due to the order mismatch. Consequently, the access of e_2 is invisible to LLC, reducing the effectiveness of the cacheback and the access order in L2 and LLC diverse further away from the program order. To avoid the mismatching access order, we propose to flush all the addresses stored in $\mathcal{E}$ before caching them back, as highlighted in the code extraction of the probe part illustrated in Algorithm 3 (replacing the code highlighted blue in Algorithm 1). In this way, each accessing of e_i forces an insertion at the MRU position in the LLC.

The other problem is the broken filter effect in single-core CT attack when the number of addresses in $\mathcal{E}$ is larger than the associativity of the inner caches: $|\mathcal{E}| \geq W_{\text{L1}} + W_{\text{L2}}$ for the cache architecture shown in Fig. 3. Let us consider the situation after the cacheback process is finished, the target x is actually evicted from L1 and L2, because the total number of addresses in $\mathcal{E}$ becomes 13 after adding a_3. As a result, LLC observes a re-access of x soon after probing for the next random address. This would put x to the unfavorable MRU position if LLC adopts the LRU replacement. It is even worse for the RRIP replacement policy as a re-access of x promotes it to higher replacement priority [13], which would fail the CT algorithm. To avoid such problem, we propose to interleavedly re-access the target address x during the cacheback process, as shown in Algorithm 4 (replacing the code highlighted blue in Algorithm 1). In this way, CT ensures that x is never evicted from L1.

4.3 *Extended Probing*: Increasing the Probability of Probing

Cacheback is effective only when the LLC adopts an LRU/RRIP replacement policy. If the policy is random, the probability of evicting the target address x is independent for every random address being tested. Cacheback is therefore

Algorithm 5: Extended probing

```
1 r = TRUE
2 r = r and C_v:probe(x)
3 for e in E do
4       r = r and C_a:probe(e)
5 end
6 if not r then
7       E ⋃ {a}
8 end
```

useless. In this situation, we propose to directly improve the probability of identifying a congruent address in the probing. Instead of probing only the target address x, an attacker can additionally probe all the found congruent addresses stored in $\mathcal{E}$, as they all stored in the same LLC cache set. Algorithm 5 demonstrates the probe (code highlighted blue in Algorithm 1) optimized with the extend probing. If any of the target address x or the addresses stored in $\mathcal{E}$ is probed missing in the LLC, r becomes `FALSE`, and the random address a is then identified as congruent and added to $\mathcal{E}$.

Assuming the size of $\mathcal{E}$ is $|\mathcal{E}|$, the probability of identifying a congruent address increases from $\frac{1}{W}$ to $\frac{1+|\mathcal{E}|}{W}$, which approaches to 64% when $|\mathcal{E}| = 15$ for a 16-way LLC. Consequently, N_{CA} is reduced from 256 to 54.1, achieving a 79% reduction. However, the search latency does not drop proportionally to the reduction of N_{CA}. In fact, the latency benefit eventually drops to negative with the increasing of $|\mathcal{E}|$, because the total number of accesses issued by probes rises proportionally to $|\mathcal{E}|$. They incur a significant latency overhead when $|\mathcal{E}| \to W$. When $|\mathcal{E}| < W_{\mathrm{L1}}$, addresses in $\mathcal{E}$ likely hit in L1. The extra accesses introduced by the extended probing are served by L1 and the latency overhead is small. When $|\mathcal{E}| \geq W_{\mathrm{L1}}$, the extended probing begins to experience significant amount of L1 misses. The latency overhead would gradually becomes intolerable. There should be an optimal number of addresses applied with the extended probing.

4.4 *Surrogate Targets*: Reducing Victim Accesses

The final optimization is related to reduce the number of probing the target address x. In certain attack scenarios, tricking the victim to probe the target address x (normally cross-core) is a time consuming and noisy procedure ($\Delta_{\mathrm{cross}} \gg t_{\mathrm{L1}}$ in Equation 4), especially when the victim is non-cooperative or the victim probe is likely bulky (containing unrelated code). As a result, the total time required for constructing an eviction might not be decided by the complexity of the search algorithm but largely by the number of victim accesses [22].

According to Equation 2, the number of victim access $N_{\mathrm{v}} = N_{\mathrm{CA}} \cdot S$, which is a fairly large number. We would like to significantly reduce N_{v}. Instead of probing the target address x, an attacker can replace x with a found congruent address as the surrogate target, such as e_0 stored in $\mathcal{E}$. The number of total victim access N_{v} is therefore reduced to the number of victim accesses required for identifying the first congruent address, which is only $S \cdot W$. Note that this effectively convert an originally cross-core attack into a single-core one. Therefore, it is viable only

Table 3. Cache misses incurred by testing 1000 random addresses.

Scenario	C_0 L1 miss	C_0 L2 miss	C_1 L1 miss	C_1 L2 miss	LLC miss
cross-core	1000 ± 0.0	1000 ± 0.0	1.3 ± 0.5	1.3 ± 0.5	1001 ± 0.5
single-core	1016 ± 1.3	1016 ± 1.3	0 ± 0	0 ± 0	1000 ± 0.03

for the cache architectures feasible for the single-core case. It is also worthwhile to point out, this technique is universally effective for all inserting algorithms.

5 Performance Evaluation

The performance of CT with various optimizations is evaluated by running them on actual processors whenever possible. The two assumed failing cache architectures for the single-core case (Section 3.3) are first verified. Consequently, the speed benefits of the optimizations proposed in Section 4 are measured.

5.1 Feasibility Verification

It is widely understand that CT fails on exclusive/non-inclusive cache architectures. This section concentrates on verifying of the two inclusive cache architectures identified in Table 2 (Section 3.3) where the single-core CT fails.

For the **three levels of inclusive caches using LRU/RRIP**, we verify the failing single-core case using a behavioral cache model [27] as we do not have any of the early Intel machines or any open processor implementation adopting a three-level cache architecture. The cache model is configured with two cores. $(C_a, C_v) = (C_0, C1)$ for the cross-core case. Each core contains a 64-set 8-way L1 and an private 512-set 8-way L2, while a 4096-set 12-way LLC is shared between cores. All caches adopt the LRU replacement policy.

The baseline CT is used to test 1000 random addresses for both cross-core and single-core cases. Complying with the analysis provided in Table 2, 1∼2 congruent addresses are found in the cross-core case but none in the single-core case. Table 3 reveals the cache misses recorded in all caches. In the cross-core case, testing 1000 random addresses incurs ∼1001 misses in the LLC, where the extra 1∼2 misses are caused by the eviction of the target addresses x from the LLC, which is confirmed by the matching missing number on core C_1 (C_v). In the single-core case, the number of cache misses incurred by testing 1000 random addresses is ∼1016 on L1 but exactly 1000 on LLC. The 16 extra misses on L1 is caused by the eviction of the target address x from the L2, which would lead to re-accessing x on the LLC (broken filter effect). As a result, x is never pushed to the LRU position in the LLC, and CT fails.

For the **inclusive LLC using LRU/RRIP with private caches using random**, we manage to configure a dual-core Rocket-Chip [3,4] with a two-level cache architecture where the 1024-set 16-way LLC is inclusive using LRU while the 64-set 8-way L1 uses random. The Rocket-Chip is ported to a FPGA dev board, runs at 50MHz, and boots with a Linux kernel (ver. 5.11.0).

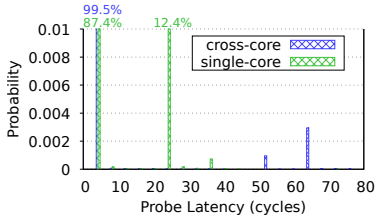


Fig. 4. The latency distribution of probing the target address x.

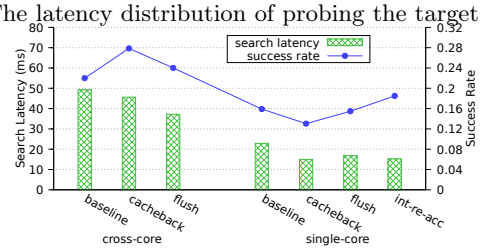


Fig. 5. Search latency and success rate on Intel i7-6700 when various cacheback optimizations are applied.

The baseline CT algorithm runs on this dual-core Rocket-Chip for both cross-core and single-core cases. The cross-core case successfully finds eviction sets with a probability of 13% while the single-core case fails, complying with Table 2. To verify this result, the latency distribution of probing the target address x has been collected and depicted in Fig. 4. For the cross-core case, 99.5% probes hit in L1 ($\sim$4 cycles), while $\sim$0.4% probes miss in LLC ($>$45 cycles). The tested CT algorithm uses random addresses sharing the same page offset with the target address x, providing a theoretical conflicting rate of 1/256 (0.391%). The 0.4% LLC miss rate matches perfectly with the theory. For the single-core case, only 87.4% probes hit in L1, 12.4% probes hit in LLC ($\sim$25 cycles), and none misses in the LLC. Due to the random replacement policy used in L1, the target address x shall be evicted from the L1 by a probability of 1/8 (12.5%), in theory. This matches with the 12.4% probes hitting in the LLC. Due to this effect, x is never pushed to the LRU position in LLC, and CT fails.

5.2 Speed Optimization Results

Cacheback (Section 4.2) reduces N_{AC} along with the search latency in inclusive LLCs adopting LRU/RRIP replacement policies. We use Intel i7-6700 as our default processor for analyzing the different techniques for improving the efficiency of cacheback while the final performance of CT with the optimized cacheback is compared on all the four Intel processors.

Fig. 5 demonstrates the search latency and success rate on Inel i7-6700 when different optimization techniques are applied to the cacheback process. In the cross-core case, applying the basic cacheback alone without *flushing before cacheback* (labeled as "flush") or *interleavedly re-access* (labeled as "int-re-acc") already raises the success rate from 22% to 28% and reduces the search latency from 49ms to 46ms. Since the target address x is accessed by C_v rather than

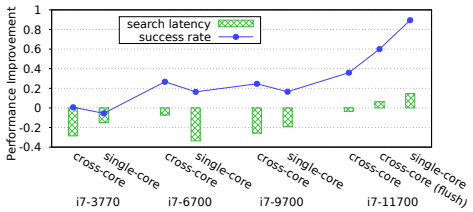


Fig. 6. Search latency and success rate on Intel processors when cacheback is applied.

Table 4. Cross-core CT on Intel processors using cacheback and surrogate targets.

CPU	Baseline		Cacheback		Surrogate Target			
	latency	rate	latency	rate	latency	rate	victim-acc.	reduction
i7-3770	13 ± 4.5ms	80%	9.5 ± 7.6ms	81%	6.4 ± 2.6ms	64%	$3.4K$	89%
i7-6700	49 ± 47ms	22%	46 ± 54ms	28%	31 ± 44ms	16%	$44K$	68%
i7-9700	44 ± 39ms	20%	33 ± 39ms	24%	29 ± 36ms	22%	$43K$	62%
i7-11700	72 ± 54ms	4.8%	69 ± 58ms	6.6%	63 ± 50ms	2.3%	$125K$	37%

C_a, caching back $\mathcal{E}$ would not evict x out of the C_v:L1 and the thrashing access pattern observed by the private L1 and L2 caches means the benefit of *flush* is marginal. As shown in Fig. 5, *flush* reduces the search latency but also incurs a drop on the success rate. *int-re-acc* is unnecessary for the cross-core case. We therefore choose the basic cacheback (without *flush* or *int-re-acc*) as the default cacheback optimized CT algorithm. In the single-core case, caching back $\mathcal{E}$ has a much higher probability to evict the target address x out of the private L1 and L2 caches than in the cross-core case. Consequently, applying cacheback itself leads to an substantial drop on the success rate. By applying both *flush* and *int-re-acc*, the success rate is raised from 16% to 19% while the search latency drops from 23ms to 15ms. We consequently define the cacheback with both *flush* and *int-re-acc* as the default cacheback optimized CT algorithm for single-core.

Fig. 6 demonstrates the performance improvement of cacheback optimized CT compared with the baseline CT on all the four Intel processors. The detail performance figures are also revealed in Table 4 for the cross-care case and Table 5 for the single-core case. The success rate is improved substantially on the more recent processors (later than the 6th generation) and this increase is most visible for the latest i7-11700 where the success rate is raised by 90% for the single-core case. As for the search latency, cacheback is able to reduce the search latency for all processors earlier than the 9th generation. Overall, cacheback is able to significantly improve the speed performance of CT on all the four tested Inel processors.

The **surrogate targets** (Section 4.4) optimization can significantly reduce the number of victim accesses by replacing the probing target from the target address x to the first found congruent address e_0 stored in $\mathcal{E}$. We have tested the CT using surrogate targets on the four Intel processors and the detailed result is revealed in the right-most columns in Table 4. The number of victim accesses is reduced by 37% to, as high as, 89%. This reduction proves the effectiveness of the optimization. The search latency is also significantly reduced to the range achieved by the single-core case. The reason is simply, once the probe target is

Table 5. Single-core CT on Intel processors using cacheback.

CPU	Baseline		Cacheback	
	latency	rate	latency	rate
i7-3770	6.0 ± 3.4ms	69%	5.1 ± 5.8ms	65%
i7-6700	23 ± 21ms	16%	15 ± 19ms	19%
i7-9700	20 ± 17ms	21%	16 ± 18ms	25%
i7-11700	12 ± 4.4ms	2.1%	13 ± 9.8ms	3.9%

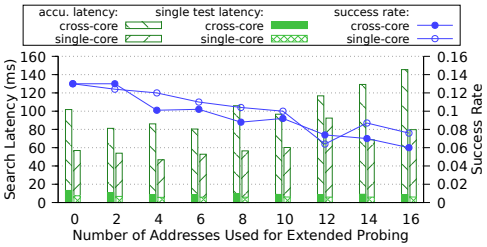


Fig. 7. Success rate and search latency (both single-test and accumulated) of CT running a dual-core Rocket-Chip (L1 LRU and LLC random) with extended probing.

replaced with e_0, the time consuming cross-core probe becomes the much faster single-core probe. However, the success rate drops to slightly lower than the single-core case. The success rate of single-core case is typically lower than the cross-core case due to its higher noise level. In addition, probing the surrogate targets suffers from a slightly reduced success rate as the found e_0 might not be congruent with x by a small probability due to false-positive errors.

Finally, we demonstrate the performance benefit of the **extended probing** (Section 4.3) again using a dual-core Rocket-Chip and configuring the replacement policies of the L1 cache to LRU and the LLC to random. Fig. 7 depicts the success rate and the search latency when the probing target is extended with 0 to 16 found congruent addresses stored in $\mathcal{E}$. The search latency is labeled as the "single-test latency" while the accumulated latency for eventually finding an eviction set (latency divided by success rate) is labeled as the "accu. latency". For both cross-core and single-core cases, extending the probe with found congruent addresses reduces the single-test search latency by increasing the success probability of probes. However, the success rate gradually drops with the number of extended probed addresses due to the increased probability of self-evicting the probe targets. The overall impact of applying extended probing is better presented by the accumulated search latency for finding an eviction set. Extending the probe with 2 to 6 addresses reduces the accumulated latency by around 20% for the cross-core case while extending the probe with 4 addresses reduces the accumulated latency by 18% for the single-core case. The result confirms that extending the probe with a small number of found congruent addresses can improve speed when the LLC adopts the random replacement policy.

6 Conclusion

In this paper, we have conducted the first systematic feasibility analysis of the CT algorithm. Besides the commonly known failing case where the LLC is exclusive or non-inclusive, two inclusive cache architectures are identified and verified as failing cases for the single-core CT. Three optimizations have been studied. The performance of the cacheback optimization has been significantly improved (especially for the single-core CT) by introducing flushing before cache back and interleaved re-access during the cacheback. The other two are newly proposed in this paper. Extended probing is effective in reducing the search latency by increasing the success probability of probes on cache architectures where the LLC adopts the random replacement policy. Surrogate targets is effective in reducing the number of victim accesses, which is hugely beneficial when the cross-core probing of the victim address is time consuming.

Acknowledgements. This research was supported by the National Natural Science Foundation of China (No. 62172406) and the CAS Pioneer Hundred Talents Program. Any opinions, findings, conclusions, and recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

1. Acıiçmez, O., Schindler, W., Koç, Ç.K.: Cache based remote timing attack on the AES. In: Topics in Cryptology – CT-RSA 2007. pp. 271–286 (2007)
2. Alfs, G., Knupffer, N.: Intel fact sheet: Intel corporation's multicore architecture briefing (Mar 2008), https://www.intel.com/pressroom/archive/releases/2008/20080317fact.htm
3. Amid, A., Biancolin, D., Gonzalez, A., Grubb, D., Karandikar, S., Liew, H., Magyar, A., Mao, H., Ou, A., Pemberton, N., Rigge, P., Schmidt, C., Wright, J., Zhao, J., Shao, Y.S., Asanović, K., Nikolić, B.: Chipyard: Integrated design, simulation, and implementation framework for custom SoCs. IEEE Micro **40**(4), 10–21 (2020)
4. Asanović, K., Avizienis, R., Bachrach, J., Beamer, S., Biancolin, D., Celio, C., Cook, H., Dabbelt, D., Hauser, J., Izraelevitz, A., Karandikar, S., Keller, B., Kim, D., Koenig, J., Lee, Y., Love, E., Maas, M., Magyar, A., Mao, H., Moreto, M., Ou, A., Patterson, D.A., Richards, B., Schmidt, C., Twigg, S., Vo, H., Waterman, A.: The Rocket chip generator. Tech. Rep. UCB/EECS-2016-17, University of California, Berkeley (Apr 2016)
5. Berg, C.: PLRU cache domino effects. In: Proceedings of the International Workshop on Worst-Case Execution Time Analysis (WCET) (2006)
6. Bernstein, D.J.: Cache-timing attacks on AES (2005), https://cr.yp.to/antiforgery/cachetiming-20050414.pdf
7. Genkin, D., Pachmanov, L., Tromer, E., Yarom, Y.: Drive-by key-extraction cache attacks from portable code. In: Proceedings of the International Conference on Applied Cryptography and Network Security (ACNS). pp. 83–102. Springer (Jul 2018)

8. Gras, B., Razavi, K., Bosman, E., Bos, H., Giuffrida, C.: ASLR on the line: Practical cache attacks on the MMU. In: Proceedings of the Network and Distributed System Security Symposium (NDSS). Internet Society (Feb 2017)
9. Gruss, D., Maurice, C., Mangard, S.: Rowhammer.js: A remote software-induced fault attack in JavaScript. In: Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMV), pp. 300–321. Springer (Jul 2016)
10. İnci, M.S., Gulmezoglu, B., Irazoqui, G., Eisenbarth, T., Sunar, B.: Cache attacks enable bulk key recovery on the cloud. In: Proceedings of the International Conference on Cryptographic Hardware and Embedded Systems (CHES). pp. 368–388. ICAR (Aug 2016)
11. Irazoqui, G., Inci, M.S., Eisenbarth, T., Sunar, B.: Wait a Minute! A fast, Cross-VM Attack on AES. In: Stavrou, A., Bos, H., Portokalidis, G. (eds.) Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID). pp. 299–319 (2014)
12. Jaleel, A., Borch, E., Bhandaru, M., Jr., S.C.S., Emer, J.: Achieving non-inclusive cache performance with inclusive caches: Temporal locality aware (TLA) cache management policies. In: Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE (May 2020)
13. Jaleel, A., Theobald, K.B., Steely Jr., S.C., Emer, J.S.: High performance cache replacement using re-reference interval prediction (RRIP). In: Proceedings of the International Symposium on Computer Architecture (ISCA). pp. 60–71. ACM (Jun 2010)
14. Kocher, P., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y.: Spectre attacks: Exploiting speculative execution. In: Proceedings of the IEEE Symposium on Security and Privacy (S&P). pp. 19–37 (May 2019)
15. Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., Hamburg, M.: Meltdown: Reading kernel memory from user space. In: Proceedings of the USENIX Security Symposium (Security). pp. 973–990. USENIX Association (Aug 2018)
16. Liu, F., Yarom, Y., Ge, Q., Heiser, G., Lee, R.B.: Last-level cache side-channel attacks are practical. In: Proceedings of the IEEE Symposium on Security and Privacy (S&P). IEEE (May 2015)
17. Maurice, C., Scouarnec, N.L., Neumann, C., Heen, O., Francillon, A.: Reverse engineering intel last-level cache complex addressing using performance counters. In: Proceedings of the International Symposium on Research in Attacks, Intrusions, and Defenses (RAID). pp. 48–65. Springer (Nov 2015)
18. Percival, C.: Cache missing for fun and profit (2005)
19. Purnal, A., Giner, L., Gruß, D., Verbauwhede, I.: Systematic analysis of randomization-based protected cache architectures. In: Proceedings of the IEEE Symposium on Security and Privacy (S&P). pp. 987–1002. IEEE (May 2021)
20. Purnal, A., Turan, F., Verbauwhede, I.: Prime+Scope: Overcoming the observer effect for high-precision cache contention attacks. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS). pp. 2906–2920. ACM (Nov 2021)
21. Purnal, A., Turan, F., Verbauwhede, I.: Double trouble: Combined heterogeneous attacks on non-inclusive cache hierarchies. In: Proceedings of the USENIX Security Symposium (Security). pp. 3647–3664. USENIX Association (Aug 2022)
22. Purnal, A., Verbauwhede, I.: Advanced profiling for probabilistic Prime+Probe attacks and covert channels in ScatterCache. ArXiv: cs.CR (Aug 2019)

23. Qureshi, M.K.: New attacks and defense for encrypted-address cache. In: Proceedings of the International Symposium on Computer Architecture (ISCA). pp. 360–371. ACM (Jun 2019)
24. Ristenpart, T., Tromer, E., Shacham, H., Savage, S.: Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In: Proceedings of the ACM Conference on Computer and Communications Security (CCS). pp. 199–212. ACM (Nov 2009)
25. Savage, J.E., Zubair, M.: A unified model for multicore architectures. In: Proceedings of the International Forum on Next-Generation Multicore/Manycore Technologies (IFMT). p. 12. ACM (Nov 2008)
26. Shen, S., Li, Z., Song, W.: Methods of extracting parameters of the processor caches. In: Proceedings of the International Workshop on Security. pp. 47–65. Springer (Aug 2022)
27. Song, W., Liu, P.: Dynamically finding minimal eviction sets can be quicker than you think for side-channel attacks against the LLC. In: Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID). pp. 427–442. USENIX Association (Sep 2019)
28. Thoma, J.P., GÃijneysu, T.: Write me and I'll tell you secrets — write-after-write effects on Intel CPUs. In: Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID). ACM (Oct 2022)
29. Tromer, E., Osvik, D.A., Shamir, A.: Efficient Cache Attacks on AES, and Countermeasures. Journal of Cryptology **23**(1), 37–71 (Jan 2010)
30. TÃşth, R., Faigl, Z., Szalay, M., Imre, S.: An advanced timing attack scheme on RSA. In: Networks 2008 - The 13th International Telecommunications Network Strategy and Planning Symposium. vol. Supplement, pp. 1–9 (2008)
31. Vila, P., Ganty, P., Guarnieri, M., KÃűpf, B.: CacheQuery: learning replacement policies from hardware caches. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). ACM (Jun 2020)
32. Vila, P., Köpf, B., Morales, J.F.: Theory and practice of finding eviction sets. In: Proceedings of the IEEE Symposium on Security and Privacy (S&P). pp. 39–54. IEEE (May 2019)
33. Wong, H.: Intel Ivy Bridge cache replacement policy (Jan 2013), http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/
34. Yan, M., Gopireddy, B., Shull, T., Torrellas, J.: Secure hierarchy-aware cache replacement policy (SHARP): defending against cache-based side channel atacks. In: Proceedings of the Annual International Symposium on Computer Architecture (ISCA). pp. 347–360. ACM (Jun 2017)
35. Yan, M., Sprabery, R., Gopireddy, B., Fletcher, C.W., Campbell, R.H., Torrellas, J.: Attack directories, not caches: Side-channel attacks in a non-inclusive world. In: Proceedings of the IEEE Symposium on Security and Privacy (S&P). pp. 888–904. IEEE (May 2019)
36. Yarom, Y., Falkner, K.: FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In: Proceedings of the USENIX Security Symposium (Security). pp. 719–732. USENIX Association (Aug 2014)
37. Zhou, Y., Feng, D.: Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing (2005), http://eprint.iacr.org/2005/388