




Methods of Extracting Parameters of the Processor Caches

Sihao Shen^{1,2}, Zhenzhen Li^{1,2}, and Wei Song^{1,2}(✉) 

¹ State Key Laboratory of Information Security, Institute of Information Engineering, CAS, Beijing, China

songwei@iie.ac.cn

² School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China

Abstract. As attack scenarios and targets are constantly expanding, cache side-channel attacks have gradually penetrated into various daily applications and brought great security risks. The success of a cache side-channel attack relies heavily on the pre-knowledge of some important parameters of the target cache system. Existing methods for reading cache parameters have their limits. In this paper, a series of tests are proposed to extract cache parameters at runtime, which provides a method for launching existing cache side-channel attacks in some restricted cases and reduces the cost of attacks. They have been used to extract cache parameters on four processors using three different architectures, as well as in a restricted virtual machine environment. The extracted parameters match with the publicly available information, including some parameters unavailable from the CPUID instruction.

Keywords: hardware security · cache side-channel · micro-architecture

1 Introduction

Cache side-channel attacks have become an important way of leaking critical information in modern computer systems, especially after their employment in the Meltdown [1] and Spectre attacks [2]. The attack scenario has been broadened from a single core to multicore processors, virtual machines (VMs) [3] and trusted execution domains [4–6]. The targets of attacks also grow from just the secrets of crypto-algorithms to users' private data [7, 8], the mapping of virtual and physical address spaces [9] and manipulating data bits in memory [10].

The success of a cache side-channel attack relies heavily on the pre-knowledge of some important parameters of the target cache system. The *access latency* of the target cache is used as the time reference for inferring cache states [11]. The *size of a cache* and the *number of cache sets* affect the probability in finding an address conflicting with the target address [12]. Attacking using the minimal eviction set is crucial for a clean and stealth attack [13], while the size of this eviction set is decided by the *number of ways* in each cache set in set-associative caches [12, 13]. The *replacement policy* asserts significant impact on the way of

using eviction sets. For permutation-based policies [14], such as the widely used pseudo LRU (Least Recently Used) [15], sequentially accessing an eviction set is sufficient to dislodge the target. However, repeated and complicated accessing methods are required when scan-resistant policies, such as RRIP (Re-Reference Interval Prediction) [16], are adopted by modern processors [17]. In addition, it is found that the replacement policy also decides the optimal method for searching eviction sets [18].

Before launching the actual attack, attackers need to collect the aforementioned parameters with some investigation. Some parameters, such as the size of cache, might be publicly available if the processor information can be precisely identified through the `CPUID` instruction of x86-64 or the `lscpu` command on Linux. Other parameters, usually the access latency of individual cache levels, could be calculated by running tests on the target system [11, 13, 19]. However, the existing methods have some limitations. Not all architectures provide the `CPUID` instruction. Even when it is available to user land, it might be virtualized to mask the cache related information or even provide wrong information [20]. System commands, such as `lscpu`, might not be available as attackers have no method to open a shell. Testing the access latency at runtime might be problematic if all high-resolution timers, like the `RDTSC` of x86-64, are disabled [21]. Finally, attacks might be launched in a restricted environment [7] where attackers have almost no direct access to machine level instructions or resources.

To address these issues, this paper proposes a series of tests to extract the required cache parameters at runtime. These tests do not rely on accessing any of the files, commands and instructions leaking the cache information or the processor model. Instead of utilizing existing timing sources on the target system, a high-resolution timer is created and utilized to measure the access latency of all cache levels. *Consequently, these tests have the potential to be ported across different computer architectures and running in restricted environments, which provides a method for launching existing cache side-channel attacks in some restricted cases and reduces the cost of attacks.* In fact, we have already run the same tests on four processors over three different instruction sets (ISAs) including x86-64, AArch64 and RISC-V, as well as in a virtual machine environment. The tests have successfully extracted almost all the aforementioned cache parameters, including some parameters unavailable from the `CPUID` instruction.

2 Background

2.1 Cache Architecture

In modern processors, caches adopt a multi-level hierarchical structure. Taking the recent Intel processors as an example, level-one (L1) and level-two (L2) caches are private caches accessible only by the local core, while level-three cache (L3 \$), acting as the LLC (last-level cache), is shared by all cores. Normally, caches located near the processor core pipeline (inner caches), such as L1 \$, operate at a higher speed and smaller size than those far away from the core (outer caches), such as the LLC. A memory access always starts from the inner caches and inquires the outer caches only when data misses in the inner ones.

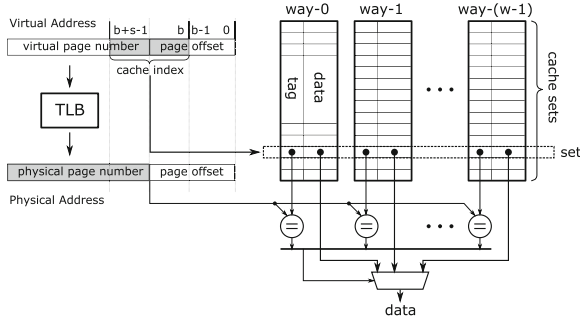


Fig. 1. A virtually indexed and physically tagged cache

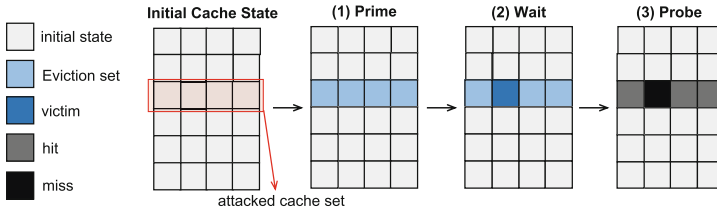


Fig. 2. Process of Prime+Probe attack

Almost all caches use a set-associative internal structure. The cache space is divided into cache sets and each set contains multiple ways of fixed sized cache blocks. Figure 1 depicts a virtually indexed and physically tagged cache normally used as the L1 \$. The 2^s cache sets are indexed by a segment of the virtual address (VA[b+s-1:b]) while the lower b bits (VA[b-1:0]) are used as the cache block offset and the higher bits (VA[63:12] assuming the 4 KB page size) are used by the translate lookaside buffer (TLB) for generating the physical page number also used as the tag for the cache way matching. Each cache set contains w cache blocks, i.e., w ways. When accessing a data, a cache set is selected by the VA and all cache blocks inside this set are simultaneously checked with the tag provided by the TLB. If the data is cached, one of the cache blocks would match with the tag; otherwise, the data is uncached (a miss) and will be fetched from the outer cache. Consequently, this missing block is stored in the cache set at either an unoccupied way or a cache block, chosen by a replacement policy, is evicted to the outer cache to make a room.

2.2 Cache Side-Channel Attacks

Cache side-channel attacks are based on the difference in time, where the latency is small when an attacker accesses a cached address but large when this address is evicted from the cache [22, 23]. Attackers can obtain a lot of sensitive information from this time difference, which leads to information leakage.

Commonly used cache side-channel attacks are mostly divided into two categories: flush-based attacks and conflict-based attacks. The flush-based attacks [24–26] require explicit cache control instructions to invalidate the target cache block, such as the `clflush` [24] on x86, in addition to requiring the target cache block must be shared between the attacker and the victim. This type of attack is simple and accurate but it relies too much on memory sharing and cache control instructions, making this attack unsuccessful in many restricted situations. If either of the above conditions is not satisfied, the attacker could launch conflict-based attacks to achieve similar effect [12]. This type of attack exploits the fact that each cache set holds only a fixed number of cache blocks, and blocks mapped to the same set conflict with each other [27–30]. The attacker can thus control the state of a cache set by occupying it completely. After the victim program is executed, victim information can be inferred by rechecking whether the cache set is still fully occupied.

Prime+Probe [31] is the most classic conflict-based cache side-channel attack. The attack process can be roughly divided into three stages, as shown in Fig. 2: **(1) Prime:** The attacker accesses a pre-prepared eviction set to occupy the target cache set to evict all victim data. **(2) Wait:** The attacker waits for a period of time, during which the victim executes the program and reoccupies the cache. **(3) Probe:** The attacker accesses the eviction set again and records the access latency. If the victim accesses the target cache set while waiting, some of the attacker’s cache blocks are evicted from the target cache set. They must be reloaded from memory during probe resulting prolonged access latency.

In a conflict-based cache side-channel attack, an important step is to construct the eviction set, which consists of a collection of (virtual) addresses that are all congruent to each other with the target address [12, 13], i.e., all mapping to the same cache set.

Definition 1. If and only if two virtual addresses x and y map to the same cache set, $Set(x) = Set(y)$ [13], but are not on the same cache block, $Cb(x) \neq Cb(y)$, then the addresses x and y are said to congruent to each other:

$$Congruent(x, y) \iff Set(x) = Set(y) \wedge Cb(x) \neq Cb(y) \quad (1)$$

Definition 2. $[x]$ denotes the collection of all congruent addresses with address x . Suppose the number of ways for the cache set is w . For a target address x , a collection of virtual addresses S is an eviction set for x if $x \notin S$, and at least w addresses in S are congruent with x [13]:

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

3 Threat Model

We assume unprivileged attackers with the ability to launch a multi-thread program on the target system and allocate consecutive memory in the virtual memory space. All files, commands and instructions that might leak the cache

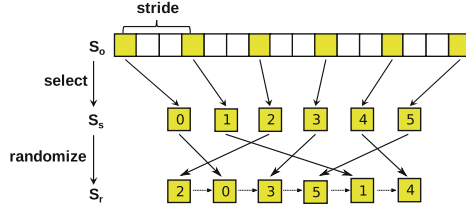


Fig. 3. Construct the randomized sequence of addresses

Algorithm 1: Latency measurement

```

1 function latency( $S_r$ )
2   start = timer()
3   foreach  $p$  in  $S_r$  do  $p = *p$ 
4   return (timer() - start)/len( $S_r$ )
5 end

```

information or the processor model has been disabled on the target system, attackers cannot get these information directly by reading files or executing commands such as `lscpu`. Attackers cannot directly launch a flush-based attack. The parameters of the target platform are not known to attackers in advance and need to be obtained through actual measurements. Attackers may be in a virtual environment. Meanwhile, sources of high-resolution timers, such as the RDTS of x86-64, might be removed or made unusable.

4 Measuring Cache Access Latency

The access latency of a cache is the foremost crucial parameter required for time side-channels while also the easiest one to obtain. It is therefore chosen as the first cache parameter to be extracted.

4.1 Random Cache Scan

The main idea of estimating the access latency of a cache is by measuring the overall latency of accessing a pre-constructed sequence of addresses. In order to accurately measure the access latency while effectively circumventing the various optimization implemented in modern processors, the access latency is averaged from the overall time of traversing a long and randomized sequence of addresses S_r constructed according to Fig. 3 [17]. A consecutive memory space S_o is initially allocated from the virtual address space. According to a predefined stride, a consecutive address sequence S_s is constructed from S_o and then randomized to form the final S_r . A final step is to link S_r into a linked-list by storing the next address in the memory pointed by the current address, which is the key in disabling instruction level parallelism as described by Algorithm 1.

Before actually extracting the cache access latency using Algorithm 1, S_r is accessed for multiple times to ensure the maximum number of addresses of

S_r have already been cached. The final round of traverse is a timed run. Inside the traverse, the next address is decided by reading the content of the current address (line 3); therefore, the processor pipeline cannot accurately predict the next address and the overall traverse time is an accumulation of individual memory accesses. Naturally, the averaged cache access latency is averaged from the overall time. Conceptually, this latency can be considered as the optimal cache performance for a certain size of data (S_o) after the cache system is properly warmed. The detailed method to extract the cache access latency of individual cache levels will be revealed in Sect. 5.

4.2 A Portable Timer

The latency of cache accesses ranges from a couple to several hundreds of nanoseconds [11]. To accurately measure this latency, especially for the L1 caches, we need high-resolution timers. On x86-64 processors, such a timer can be conveniently built from the RDTSC instruction. Other processor architectures are nevertheless lack of such high-resolution source of time in user land. We summarize the applicable architectures of commonly used timer resources and their approximate accuracy in Table 2 in Sect. 6.

In order to achieve the portability across architectures, we choose to construct a virtual time stamp (VTS) as firstly introduced in [32] for all processors. The detailed method is illustrated in Algorithm 2. Assuming the processor under test is a multicore processor, a separated child timer thread is attached to a unique core, which does nothing else but constantly increases a global counter *cnt*. In the main thread, the latency measurement process then utilizes *cnt* as a wall clock for timing. Since self-increasing is usually faster than memory accesses, this wall clock should be quick enough as long as it is not disturbed by context switching.¹ Additionally, each time the global variable *cnt* is incremented, it requires accessing memory twice. Actually, it is possible to reduce the number of memory accesses by executing the self-increment operation directly through assembly instructions [28]. This means that the global variable *cnt* can be incremented faster in the same time, thus improving the resolution. The actual resolution of this virtual time stamp is evaluated in Sect. 6.

Furthermore, we did not attach threads to a certain core (without using CPU affinity) in the actual experiments. According to our observations, the probability of threads being migrated to other cores is very low and is a small probability event. If the counting thread has core migration, we believe that there will be an impact on the clock accuracy within a short period of time when the migration occurs, but these effects will be averaged over multiple samples in the experiment and have little effect on the final result. Of course, using CPU affinity to attach the counting thread on one core will improve the accuracy of the algorithm, but it will also inevitably reduce the cross-platform capability of the algorithm.

¹ Such context switching can be detected by software as it usually leads to outstanding measurement errors.

Algorithm 2: Virtual time stamp

```

1 global variable  $cnt \leftarrow 0$ 
2 //child timer thread
3 while(true) do  $cnt ++$ 
4
5 //main thread
6 function  $latency'(S_r)$ 
7    $start = cnt$ 
8   foreach  $p$  in  $S_r$  do  $p = *p$ 
9   return  $(cnt - start)/len(S_r)$ 
10 end

```

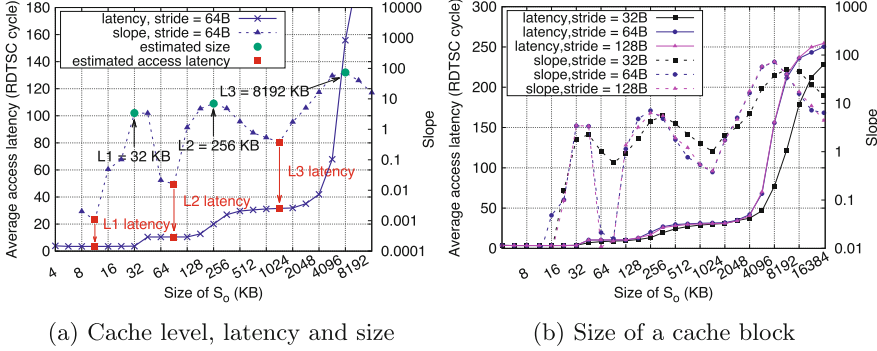


Fig. 4. Extracting basic parameters using random cache scan

5 Methodology of Extraction

This section describes the series of tests used to extract individual cache parameters. To better illustrate the details of each test, we provide actual test results collected from an Intel i7-3770 using RDTSC as the timing source. In addition, the virtual time stamp are used as the timer to detect replacement policies in Figs. 6 and 7 since ARM and RISC-V architecture processors are involved. The experimental results of using the virtual time stamp and running on other more recent processor architectures are revealed in Sect. 6.

5.1 Cache Size and Latency of All Levels

The parameter extraction starts with a series of cache scans using a relatively small stride (such as 64 B) but with different sizes of S_o . An exemplary test on an Intel i7-3770 is depicted in Fig. 4a. The number of cache levels, the access latency and the size are the first batch of parameters to be extracted.

When S_o is smaller than the size of L1 \$, the access latency l denotes the L1 access latency as all accesses hit in L1 \$. When S_o grows well beyond the size of L1 \$, nearly all access miss in L1 because the long scan pattern leaves no locality for the L1 \$ to explore. Consequently, all accesses are served by the L2 \$ and l equates to the access latency of the L2 \$. Similarly, we can extract the

L3 access latency using an even larger S_o . However, we need to first figure out the sizes of individual cache levels.

According to Fig. 4a, the latency l increases with S_o at a varying speed. When S_o grows just surpassing the size of a cache, l jumps from the access latency of the current level to the next. The number of cache levels can be extracted by counting the number of these latency jumps. It is found that such jumps can be clearly detected by analyzing the slope curve, which measures the first order of derivative of l calculated as:

$$f(i) = \frac{l_{i+1} - l_i}{2} + \frac{l_i - l_{i-1}}{2} \quad (3)$$

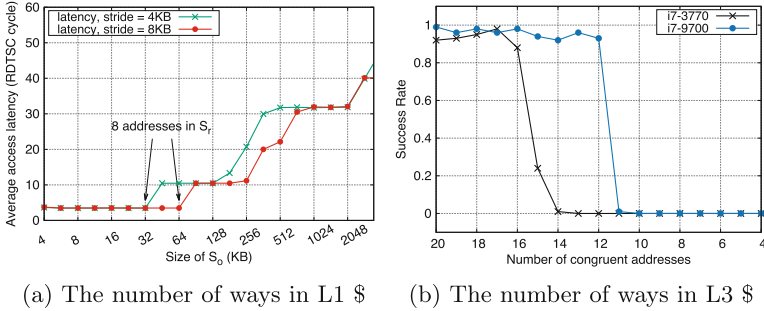
where $f(i)$ denotes the increasing speed of l at x-axis location i . Note that the x-axis and the y-axis for the slope curve in Fig. 4a are both logarithmized. We use x-axis location i as the function input while the corresponding S_o and l are l_i and $S_{o,i}$ respectively. A value of l is sampled every time that S_o is increased by $\sqrt{2}$.² As shown in the slope curve, three peaks unambiguously reveal the existence of three levels of caches. More interestingly, the peaks locate exactly in the vicinity of the sizes of individual caches. This is because when the size of S_o exceeds the cache size, the cache generates a large number of capacity misses [33] and the average access latency of the sequence increases sharply. Using the related S_o of a peak as a rough estimation and correcting it using common sense, such as the number of sets should be 2ⁱpower, we can infer the sizes of individual cache levels. Moreover, the latency of a cache level can be estimated using the latency l_i at location of the lowest $f(i)$ related to the cache level.

5.2 Size of a Cache Block

A cache block is the smallest portion of data being communicated between caches. Although almost all modern processors adopt a uniformed block size of 64 bytes to ease the implementation of cache coherence, some processors use non-64 uniformed block size or even different block sizes across cache levels. We cannot simply assume that the block size is 64 bytes universally.

The way to extract the block size at a certain cache level is to pinpoint a match between the block size and a stride. If the chosen stride is smaller than the block size, each cache block has multiple addresses contained in S_r while only one address is contained if the stride is equal to or larger than the block size. When S_o grows just beyond the cache size, part of cache accesses begin to miss and the average latency starts to rise. In this situation, the average access latency using a smaller stride is lower than using a larger stride. Since multiple addresses of the same cache block is contained in S_r using a small stride and a whole cache block is refilled when missed, each cache refill is effectively a prefetch

² Introducing extra samples in between each pair of basis points ($\times 2$) sharpens the peaks in the slope curve, which makes the peaks easy to detect but leads to long running time. As a trade-off, only one extra sample is added at the middle ($\sqrt{2}$) of the basis points on the logarithmized x-axis.



(a) The number of ways in L1 \$ (b) The number of ways in L3 \$

Fig. 5. Extracting the number of ways

for the remaining addresses not yet accessed to the same block, which then leads to the reduced latency. If we sweep stride from a small value to a value larger than the block size, a gradual rise of the latency curve should be observable until the stride is equal to the block size. Any further increase on stride results in a similar latency curve. Consequently, the first stride fails to raise the latency curve is equal to the block size.

In our test, the stride is gradually doubled from 8 to 256 bytes. The latency curves for strides from 32 to 128 bytes are depicted in Fig. 4b. The Intel i7-3770 adopts a uniformed block size of 64 bytes. The latency using a stride of 32 bytes is indeed lower than the latency of stride 64 and 128 bytes while the latency curve of the latter mostly identical. However, it is difficult to check whether two curves are identical by a program. Instead, we check whether the peaks of two slope curves are co-located with a small error. Also shown in Fig. 4b, the peaks of the slope curves of stride 64 and 128 bytes perfectly co-located for all cache levels, which reveals that all cache levels use the same block size of 64 bytes. The extra benefit of using the slope curve is the enlarged distance between peaks. For the peaks using stride less than 64 bytes, the height of the peak is noticeably lower and the location is pushed rightwards, thanks to the much milder latency jumps produced by them.

5.3 Number of Cache Ways and Sets

The random cache scan can be used to extract the number of ways in a L1 cache set provided the L1 \$ is set-associative. As shown in Fig. 5a, the latency curve moves rightwards when the stride grows beyond 4KB. This is because all the addresses in the 32KB S_r (stride = 4KB) are congruent [13] and mapped to the same cache set due to the hardwired cache set index $\text{VA}[\mathbf{b}+\mathbf{s}-1:\mathbf{b}]$ as illustrated in Fig. 1, and they are just enough to fill the whole set. When the stride increased to 8KB, the number of addresses is halved. To fill the whole set then requires a S_r covering 64KB. Note that the size S_r divided by the stride is both 8 for the two cases, revealing the number of ways in the L1 \$ is 8. We can explain it from another angle. Since the L1 \$ is virtually indexed, by choosing addresses with the same stride, we effectively create an eviction set for a set. Detecting the shift

of curve thus reveals the minimum number of addresses required for an eviction set, which is exactly the number of ways for set-associative caches [13, 34].

However, this method is only suitable for L1 \$ because all outer caches are physically indexed and addresses apart from the same stride on longer guaranteed of mapping to the same set. For the outer caches, we extend the group elimination search algorithm [13, 34] to search for congruent addresses instead of using the random cache scan. At the beginning, the number of congruent addresses a is large enough to fill the whole set to create an eviction set. However, it cannot create an eviction set anymore when a is less than the number of ways. By gradually reducing the number of congruent addresses in an eviction set, we can derive the minimum number of addresses, which is also the number of ways. Figure 5b shows the results of extracting the number of ways of the L3 \$ on both i7-3770 and a latest i7-9700. When the number of addresses is set to less than the minimum number (the number of ways), the success rate of finding an eviction set immediately drops to zero. The result clearly reveals that the numbers of ways are 16 and 12 for the L3 \$ on Intel i7-3770 and i7-9700 respectively.

The detailed steps is illustrated in Algorithm 3. The input candidate set C is divided into $a + 1$ groups (a is the number of congruent addresses). Since the eviction set contains a congruent addresses, there must be a certain group among the $a + 1$ groups that does not contain the addresses in the eviction set. For each group G , if the target address x can still be evicted after removing it from the candidate set C , it means that the addresses in the group G are irrelevant to the eviction set, then remove the group G . Conversely, keep the group G and continue to detect whether the next group G can be removed. Until a group G that can be removed from the candidate set C is found, then the current round of detection is ended. The remaining candidate set C continues to be divided into $a + 1$ groups to start the next round of detection until the number of congruent addresses in C is exactly equal to a , thus the eviction set S is successfully obtained.

Finally, since cache size equates the production of number of sets, number of ways and block size, it is straightforward to calculate the number of sets once the other three parameters are extracted, i.e. $cache\ size = set * way * block\ size$.

5.4 Replacement Policy

In a cache side-channel attack, dislodging the target address by traversing an eviction set is literally a thrashing access pattern [16] whose effectiveness is closely related to the replacement policy adopted by the target cache. A couple of traverses are usually enough for *permutation-based* policies, such as LRU [35]. Increasing the number of traverses is sufficient to defeat *random* replacement policies. Complicated traverse algorithms [12] would be required for *scan-resistant* policies, such as RRIP [16, 36]. Instead of detecting the exact types of policies [19], this paper tries to classify replacement policies into three categories: permutation-based, random and scan-resistant policies.

It is relatively easy to differentiate permutation and non-permutation policies. Figure 6 depicts the jump of access latency when S_r grows beyond the size of the L1 \$ on Intel i7-3770 and the HiFive unleashed board (RISC-V processor). The virtual time stamp is used to measure the access latency. As indicated

Algorithm 3: Group elimination search

Input: C , candidate set; x , target address; a , number of congruent addresses.
Output: S , eviction set for x .

```

1 function group_reduction( $s, x, a$ )
2   while  $|C| > a$  do
3      $G_1, \dots, G_{a+1} \leftarrow \text{split}(C, a + 1)$ 
4      $i \leftarrow 1$ 
5     while  $\neg \text{TEST}(C \setminus G_i, x)$  do
6        $i \leftarrow i + 1$ 
7     end
8      $C \leftarrow C \setminus G_i$ 
9   end
10   $S \leftarrow C$ 
11  return  $S$ 
12 end

```

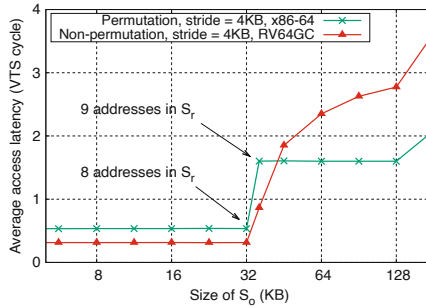
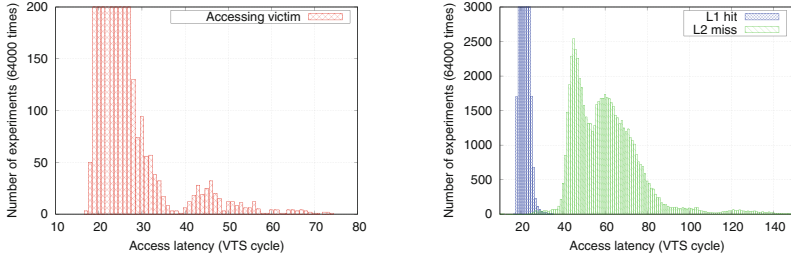


Fig. 6. Differentiating permutation and non-permutation policies

by the result, the two L1 \$ certainly adopts different replacement policies. Intel i7-3770 likely uses a permutation-based policy because the access latency suddenly increases when S_r just grows beyond the cache size (32 KB), indicating the cache scan with 9 addresses can easily dislodge all the 8 cache blocks. As for the RISC-V processor, much more congruent addresses are required for evicting the whole cache set, denoting the use of a non-permutation policy.

To further differentiate scan-resistant and random policies, we have done a modified cache scan as described by Fig. 6 in [17] on the RISC-V processor whose caches adopting non-permutation policies. The sequence S_r is divided into a short and a long sequence. The short sequence should fit in the target cache and are initially traversed multiple times to mimicking a access pattern with temporal locality. The whole sequence is then used in a normal cache scan but only the access latency of the short sequence is measured. If a scan-resistant policy is adopted, addresses belonging to the long sequence are replaced before the short ones, and the latency curve is pushed rightwards, as described in [17].

In summary, permutation and non-permutation policies are detected using a normal cache scan. A permutation policy is used if the latency curve shows a narrow and sharp jump at the size of the cache; otherwise, a non-permutation policy is used. A modified cache scan is then applied. If the latency curve of the short sequence is noticeably pushed rightwards from the size of the cache, a scan-resistant policy should be used; otherwise, it is likely to be a random one.



(a) Detecting if victim is evicted from L2 \$

(b) Obtaining the latency of accessing memory

Fig. 7. Detecting the random replacement policy by latency distributions

Furthermore, there is a way to verify the use of a random policy by calculating whether the substitution rate is completely random through a large number of repeated experiments. Take the L2 \$ on the Jetson Nano (ARM architecture) as an example, the number of its ways is 16. If the L2 \$ adopts a random policy, then the probability of successfully evicting the target address to memory should be only $1/16$ when using a congruent address to evict the target address. By repeating the experiment we can obtain this probability and thus infer whether the cache adopts a random policy. We first need to find two addresses called attacker and victim that map to the same cache set on both L1 and L2 \$. Then the eviction set of attacker and victim is constructed. Each address in the eviction set must be in the same set as these two addresses in L1 \$ but in a different set in L2 \$, in order to ensure that no additional noise is introduced when evicting the target address to the L2 \$.

First, access the eviction set to evict the victim to the L2 \$, and then re-access the eviction set to evict the attacker to the L2 \$ as well. Since attacker and victim are congruent with each other, victim may be evicted from the L2 \$ to memory during this process. Finally, the state of the victim in the cache is inferred according to its access latency. The above experiments were performed 64,000 times and the access latency of the victim was counted. This experiment uses the virtual time stamp as the timer and the results are shown in Fig. 7a. According to the analysis, the probability that the attacker successfully evicts the victim to memory is $1/16$ (the reciprocal of the numbers of ways for the L2 \$) if the L2 \$ adopts a random policy. In this case, the access latency of victim is the time it takes to fetch data from memory.

To obtain the access latency of memory, we need to count the cache access latency in different states. A certain target address is accessed multiple times to make it cached, and then it is evicted from the cache through the cache flush instruction. These two different states of access latency are recorded and

the results are shown in Fig. 7b. We can intuitively obtain the access latency of memory (L2 miss) is about 30 or more.³

We then calculated the frequency of access latency greater than 30 in Fig. 7a, which is about 1/18 of the total number of experiments (the practically possible number of ways closest to this value is 16). Only the random policy has a replacement rate of 1/16, so we can infer that the L2 \$ does adopt a random policy based on this result. This experiment exploits the law of large numbers, i.e., repeating the experiment many times under the same conditions, the frequency of a random event will approximate its probability. That is why we need a sufficient number of experiments to ensure accuracy, and this also brings a long time-consuming problem. How to detect the random policy more quickly and accurately is also one of our subsequent research goals.

6 Experiment Results

We have chosen four representative processors using three different architectures to conduct the experiments. The processor information is illustrated in Table 1. Besides the relatively old i7-3770, a latest i7-9700 processor is also tested. We have also managed to run the tests on two non-x86 processors which we have access to. One is a Jetson Nano board mounted with an Arm Cortex-A57 processor and the other one is a HiFive Unleashed board mounted with a SiFive u540 processor. All processors run a Linux operating system while tests are compiled with the default GNU GCC compiler. In order to further verify the usability of this method in some restricted environments, such as cloud computing, browser sandboxes, etc., we installed a virtual machine on the i7-9700 processor and performed the same cache parameter extraction experiments in the virtual machine environment.

The methods of measuring time varies across architectures and the commonly used time resources are shown in Table 2. The resolution may vary within the same architecture due to extra factors such as dynamic frequency scaling. Among them, the RDTSC register has the highest precision, which can reach 0.3 ns on the i7-9700 processor, but it is only applicable to the x86 architecture. The ARM architecture can use the `cntvct_e10` register for timing, which has an accuracy of about 52 ns on the Jetson Nano processor. Both the `time` and `cycle` registers can be used for timing on the RISC-V architecture, and their accuracy is 1 μ s and 1 ns respectively on HiFive Unleashed processor. While the virtual time stamp we used is applicable to all three architectures above.

We verify the resolution of the virtual time stamp by calculating the increase of the global variable `cnt` during a certain runtime period, the details are as follows: In the main thread, we accurately control the running time through a sleep function and record the increment of the global variable `cnt` in the child timer thread during this period. Dividing the running time by the increment produces the resolution of the virtual time stamp, which indicates how long

³ This latency is not consistent with Table 3 as extra delay is caused by the operations to clean states at the beginning of each test.

Table 1. Processor information and timer resolution

	Intel	Intel	Intel (VM)	Jetson Nano	Unleashed
Processor	i7-3770	i7-9700	i7-9700	Cortex-A57	SiFive u540
Arch.	x86-64	x86-64	x86-64	ARMv8.0-A	RV64GC
OS	Ubuntu 16.04	Ubuntu 18.04	Ubuntu 16.04	Ubuntu 18.04	OpenEmbedded
GCC ver.	5.4	5.4	5.4	7.5	10.2
Resolution	1.9ns	1.2ns	1.2ns	5.0ns	11.0ns

Table 2. Comparison of time resources under different architectures

	RDTSC	cntvct_e10	time	cycle	virtual time stamp (VTS)
Arch.	x86	ARM	RISC-V	RISC-V	x86/ARM/RISC-V
Resolution	0.3ns	52.0ns	1.0us	1.0ns	1.2-11.0ns

it takes for the global variable *cnt* to increase by one unit. In addition, the child timer thread will increase the single-core CPU overhead to over 90%, thus this timer runs at the highest frequency. The resolution achieved by the virtual time stamp method is revealed on the final row of Table 1. It is shown that the virtual time stamp achieves nanosecond resolution on all processors. All of the experiment results provided in this section are collected from tests using this virtual time stamp.

Taking the virtual time stamp as the precise timer, we extract the access latency at all cache levels by scanning the random address sequence on the four processors. The specific latency as well as its slope variation is shown in Fig. 8 and Table 3 along with the extracted cache parameters. Although the L1 access latency on all processors is less than the resolution of the virtual time stamp, the latency of the L2 \$ is always longer than 1. Note that the we only need to differentiate a L1 hit from miss, as long as the measured difference between the L1 and L2 latency is longer than 1, the resolution of the virtual time stamp is high enough.

Intel processors normally adopt a three-level cache hierarchy but only two levels are found on the two non-x86 processors. The proposed tests successfully produce an estimation on all cache parameters except for the numbers of sets and ways for the L2 \$ on Intel processors. These L2 \$ caches are found to be physically indexed caches. As described in Sect. 5.3, the test extracts the number of ways by trying to figure out the minimum number of congruent addresses needed by an eviction set. However, the group elimination algorithm [12] suffers from significant error rate and fails to produce any eviction sets. With some investigation, we suspect the L2 \$ on these Intel processors might be non-inclusive with regarding to the L1 \$.

The non-inclusive structure means that when the data in the upper-level cache is evicted, this evicted data will be written back to the next-level. It ensures that the current cache only holds data that is not in the upper-level cache. The design and implementation of non-inclusive cache are more complex but improve security by making the eviction set construction much more difficult [37]. The current trend in cache design is a shift from inclusive to non-inclusive, such

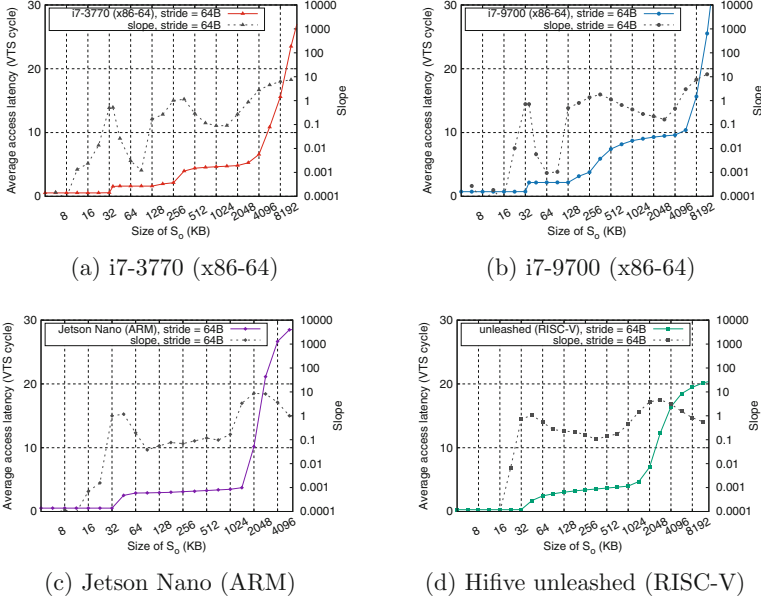


Fig. 8. Extracting cache latency through virtual time stamp (VTS)

as Intel’s Skylake architecture, which has designed the L3 \$ as a non-inclusive structure. Although it is reported possible to construct eviction sets for L3 \$ using directory-based coherence policy [37], it is unlikely for a non-inclusive and private L2 \$ to use directory. Finding eviction sets on it thus remains an open question requiring further research.

In the virtual machine environment, we extracted the same cache parameters as the i7-9700 processor in the normal environment except for the number of ways for the L3 \$. This is because the two-layer address translation mechanism in the virtual machine (VM VA to host VA, and then from host VA to host PA) leads to an increase in TLB pressure and a significant increase in miss rate. Previous studies have shown that this noise can significantly affect the success rate of the eviction set search algorithm [38]. It was found that the existing opensourced algorithms, such as the group elimination algorithm [13] and the random algorithm [12], cannot work directly in the virtual environment for the time being. How to address such problems in a restricted environment is also one of our future work.

We have compared the extracted parameters against the information available from CPUID and `lscpu`. All the parameters match with the publicly available information while the correctness on the extracted types of replacement policies remains unclear. It is partially verified by a separate research [19] that the Intel processors do adopt scan-resistant policies on the L3 \$ and even the L2 \$ for recent processors. Whether the L2 \$ of i7-3770 indeed adopting a scan-resistant policy would need further investigation. Some counter-intuitive results

Table 3. Extracted cache parameters

	i7-3770	i7-9700	i7-9700 (VM)	Jetson	Unleashed
Latency	0.59	0.72	0.73	0.52	0.31
Size	32KB	32KB	32KB	32KB	32KB
L1 Block	64B	64B	64B	64B	64B
Set/Way	64/8	64/8	64/8	256/2	64/8
Replace	permu.	permu.	permu.	permu.	random
Latency	1.76	2.18	2.21	3.34	3.63
Size	256KB	256KB	256KB	2MB	2MB
L2 Block	64B	64B	64B	64B	64B
Set/Way	?	?	?	2048/16	1024/32
Replace	scan-res.	scan-res.	scan-res.	random	permu.
Latency	5.58	8.83	8.94		
Size	8MB	12MB	12MB		
L3 Block	64B	64B	64B		
Set/Way	8192/16	16384/12	?		
Replace	scan-res.	scan-res.	scan-res.		

are found on the RISC-V processors as it uses a random replacement policy on the L1 \$. Since the L1 \$ has high performance requirements, permutation-based replacement policies (such as LRU, etc.) are usually adopted. We have double-checked our experiment result. The opensourced implementation of the SiFive u540 (Rocket-Chip) does show the possibility to set the policy to random for L1 \$ but it is still an odd choice for performance concerns.

7 Conclusion

A series of tests have been proposed in this paper to extract the cache parameters crucial for cache side-channel attacks. With the help of a virtual time stamp timer, the proposed tests have the potential to be ported across different computer architectures and running in restricted environments, which provide a method for launching existing cache side-channel attacks in some restricted cases and reduces the cost of attacks. We have conducted experiments on four representative processors using three different architectures, as well as in a virtual machine environment. Nearly all cache parameters have been extracted except for the number of ways of the L2 \$ on Intel processors because these caches are suspected non-inclusive, which makes the construction of the eviction set extremely difficult. All the extracted parameters match with the publicly available information using `CPUID` and `lscpu`. How to effectively construct an eviction set in a non-inclusive cache or virtual machine environment is currently a chal-

lence in the field of cache side-channel attacks, which is also one of our next research goals.

Acknowledgements. The HiFive Unleashed board was kindly borrowed from Xiongfei Guo. This work was partially supported by the National Natural Science Foundation of China under grant No. 62172406 and No. 61802402, the CAS Pioneer Hundred Talents Program, and internal grants from the Institute of Information Engineering, CAS. 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. Lipp, M., et al.: Meltdown: reading kernel memory from user space. In: Proceedings of the USENIX Security Symposium, August 2018, pp. 973–990 (2018)
2. Kocher, P., et al.: Spectre attacks: exploiting speculative execution. In: Proceedings of the IEEE Symposium on Security and Privacy, May 2019, pp. 19–37 (2019)
3. 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, November 2009, pp. 199–212 (2009)
4. Brasser, F., Müller, U., Dmitrienko, A., Kostianen, K., Capkun, S., Sadeghi, A.-R.: Software grand exposure: SGX cache attacks are practical. In: Proceedings of the USENIX Workshop on Offensive Technologies, August 2017
5. Hähnel, M., Cui, W., Peinado, M.: High-resolution side channels for untrusted operating systems. In: Proceedings of the USENIX Annual Technical Conference, July 2017, pp. 299–312 (2017)
6. Schwarz, M., Maurice, C., Gruss, D., Mangard, S.: Fantastic timers and where to find them: high-resolution microarchitectural attacks in JavaScript. In: Proceedings of the International Conference on Financial Cryptography and Data Security, January 2017, pp. 247–267 (2017)
7. Oren, Y., Kemerlis, V.P., Sethumadhavan, S., Keromytis, A.D.: The spy in the sandbox: practical cache attacks in JavaScript and their implications. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, October 2015, pp. 1406–1418 (2015)
8. Page, D.: Theoretical use of cache memory as a cryptanalytic side-channel. Cryptology ePrint Archive (2002)
9. 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, February 2017
10. Kim, Y., et al.: Flipping bits in memory without accessing them: an experimental study of DRAM disturbance errors. In: Proceedings of the International Symposium on Computer Architecture, June 2014, pp. 361–372 (2014)
11. Hund, R., Willems, C., Holz, T.: Practical timing side channel attacks against kernel space ASLR. In: Proceedings of the IEEE Symposium on Security and Privacy, May 2013, pp. 191–205 (2013)
12. 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 Recent Advances in Intrusion Detection, September 2019, pp. 427–442 (2019)

13. Vila, P., Köpf, B., Morales, J.: Theory and practice of finding eviction sets. In: Proceedings of the IEEE Symposium on Security and Privacy, May 2019 (2019)
14. Jain, A., Lin, C.: Cache Replacement Policies. Morgan & Claypool Publishers, San Rafael (2019)
15. Berg, C.: PLRU cache domino effects. In: Proceedings of the International Workshop on Worst-Case Execution Time Analysis, June 2006
16. Jaleel, A., Theobald, K.B., Steely, S.C.Jr., Emer, J.S.: High performance cache replacement using re-reference interval prediction (RRIP). In: Proceedings of the International Symposium on Computer Architecture, June 2010, pp. 60–71 (2010)
17. Wong, H.: Intel Ivy Bridge cache replacement policy, January 2013. <http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/>
18. Qureshi, M.K.: New attacks and defense for encrypted-address cache. In: Proceedings of the International Symposium on Computer Architecture, June 2019, pp. 360–371 (2019)
19. Vila, P., Ganty, P., Guarnieri, M., Köpf, B.: CacheQuery: learning replacement policies from hardware caches. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, June 2020, pp. 519–532 (2020)
20. Nakajima, J., Mallick, A.K.: Hybrid-virtualization – enhanced virtualization for Linux. In: Linux Symposium, vol. 2, June 2007, pp. 87–96 (2007)
21. Martin, R., Demme, J., Sethumadhavan, S.: TimeWarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In: Proceedings of the International Symposium on Computer Architecture, June 2012, pp. 118–129 (2012)
22. Deng, S., Xiong, W., Szefer, J.: A benchmark suite for evaluating caches’ vulnerability to timing attacks. In: Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems, 2020, pp. 683–697 (2020)
23. Ge, Q., Yarom, Y., Cock, D., Heiser, G.: A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *J. Cryptogr. Eng.* **8**(1), 1–27 (2016). <https://doi.org/10.1007/s13389-016-0141-6>
24. Yarom, Y., Falkner, K.: FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack. In: Proceedings of the USENIX Security Symposium, 2014, pp. 719–732 (2014)
25. Zhang, X., Xiao, Y., Zhang, Y.: Return-oriented flush-reload side channels on arm and their implications for android devices. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, 2016, pp. 858–870 (2016)
26. Zhang, Y., Juels, A., Reiter, M.K., Ristenpart, T.: Cross-tenant side-channel attacks in PaaS clouds. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, 2014, pp. 990–1003 (2014)
27. Osvik, D.A., Shamir, A., Tromer, E.: Cache attacks and countermeasures: the case of AES. In: Pointcheval, D. (ed.) CT-RSA 2006. LNCS, vol. 3860, pp. 1–20. Springer, Heidelberg (2006). https://doi.org/10.1007/11605805_1
28. Lipp, M., Gruss, D., Spreitzer, R., Maurice, C., Mangard, S.: ARMageddon: cache attacks on mobile devices. In: Proceedings of the USENIX Security Symposium, 2016, pp. 549–564 (2016)
29. Yan, M., Gopireddy, B., Shull, T., Torrellas, J.: Secure hierarchy-aware cache replacement policy (SHARP): defending against cache-based side channel attacks. In: Proceedings of the ACM/IEEE Annual International Symposium on Computer Architecture, pp. 347–360. IEEE (2017)

30. Irazoqui, G., Eisenbarth, T., Sunar, B.: 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, pp. 591–604. IEEE (2015)
31. 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, May 2015, pp. 605–622. IEEE (2015)
32. Percival, C.: Cache missing for fun and profit. In: BSD Conference Ottawa (2005)
33. Smith, A.J.: Cache memories. *ACM Comput. Surv.* **14**(3), 473–530 (1982)
34. Song, W., Li, B., Xue, Z., Li, Z., Wang, W., Liu, P.: Randomized last-level caches are still vulnerable to cache side-channel attacks! But we can fix it. In: Proceedings of the IEEE Symposium on Security and Privacy, May 2021
35. Abel, A., Reineke, J.: Reverse engineering of cache replacement policies in intel microprocessors and their evaluation. In: Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, pp. 141–142. IEEE (2014)
36. Qureshi, M.K., Jaleel, A., Patt, Y.N., Steely, S.C., Emer, J.: Adaptive insertion policies for high performance caching. *ACM SIGARCH Comput. Arch. News* **35**(2), 381–391 (2007)
37. 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, May 2019, pp. 888–904 (2019)
38. Genkin, D., Pachmanov, L., Tromer, E., Yarom, Y.: Drive-by key-extraction cache attacks from portable code. In: Preneel, B., Vercauteren, F. (eds.) ACNS 2018. LNCS, vol. 10892, pp. 83–102. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-93387-0_5