

Article

Applying Address Encryption and Timing Noise to Enhance the Security of Caches

Dehua Wu ^{1,2}, Sha Tao ^{1,2} and Wanlin Gao ^{1,2,*}¹ College of Information and Electrical Engineering, China Agricultural University, Beijing 100083, China² Key Laboratory of Agricultural Information Standardization, Ministry of Agriculture and Rural Affairs, China Agricultural University, Beijing 100083, China

* Correspondence: wanlin_cau@163.com

Abstract: Encrypting the mapping relationship between physical and cache addresses has been a promising technique to prevent conflict-based cache side-channel attacks. However, this method is not foolproof and the attackers can still build a side-channel despite the increased difficulty of finding the minimal eviction set. To address this issue, we propose a new protection method that integrates both address encryption and timing noise extension mechanisms. By adding the timing noise extension mechanism to the address encryption method, we can randomly generate cache misses that prevent the attackers from pruning the eviction set. Our analysis shows that the timing noise extension mechanism can cause the attackers to fail in obtaining accurate timing information for accessing memory. Furthermore, our proposal reduces the timing noise generating rate, minimizing performance overhead. Our experiments on SPEC CPU 2017 show that the integrated mechanism only resulted in a tiny performance overhead of 2.9%.

Keywords: side-channel attacks; conflict-based cache side-channel attacks; cache structure



Citation: Wu, D.; Tao, S.; Gao, W. Applying Address Encryption and Timing Noise to Enhance the Security of Caches. *Electronics* **2023**, *12*, 1799. <https://doi.org/10.3390/electronics12081799>

Academic Editor: Aryya Gangopadhyay

Received: 4 March 2023

Revised: 29 March 2023

Accepted: 1 April 2023

Published: 11 April 2023



Copyright: © 2023 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

Modern processors apply cache to bridge the speed gap between the memory and the processor to speed up the execution of the memory access [1]. However, the cache is vulnerable to side-channel attacks which exploit the accessible physics information about the processor, such as power consumption [2] and timing [3–5] to leak private data. The attackers exploit many components in the processor to build the channel, among which the conflict-based cache side-channel is known to be the most dangerous one [6]. By modifying the cache, the attackers can leak private data with little effort.

To mitigate this, researchers have tried to block the building of the side channel. A lot of methods have been proposed. Making the mapping between the physical address and cache address invisible is considered a promising way. Since the pruning algorithm is complex, any obscuration in the mapping can cost the attackers more effort in finding the minimal eviction set. Therefore, much mitigation follows this method, such as Ceaser [7], Scattercache [8], RPCache [9], etc.

However, the analysis from the view of the probability model to find the minimal eviction set shows that the way to make the mapping between the physical address and cache address invisible is not to disable the side channel, but only to make it more difficult [10]. Even so, some protection methods that employ a dynamic mapping policy may still have vulnerabilities [11–13].

To address this issue, this paper proposes a mechanism integrating address encryption and a timing noise extension to resist the attacks. The address encryption mechanism we proposed can change the cipher of the encryption periodically. Unlike the design of Scatter-Cache, not all the ciphers need to change, only some of them. The basic component of the timing noise extension mechanism is the timing noise extension module which can generate

a threshold and monitor the cache access. When the memory accesses reach the threshold, the module generates a miss for the next cache access, rendering timing information erroneous and unsuitable for the algorithm that prunes the minimal eviction set.

To summarize, the main contributions of this paper are as follows.

1. By analyzing the probability of cache eviction caused by the attackers, we found that the encryption mechanism cannot prevent the attackers from pruning the eviction set.
2. This paper proposed a novel cache structure that integrates address encryption and a timing noise extension. The timing measurement can be interfered with due to the timing noise extension module being applied. We found the probability that the attackers would not meet a noise stays at a low level, which means the attackers almost certainly encounter timing noise.
3. Compared with other proposals, the timing noise extension module has a lower timing noise generating rate, which has little impact on the performance. By testing the SPEC CPU 2017, we found a very small cycles per instruction (CPI) slowdown of 2.9% was made.

The rest of this paper is organized as follows. Section 2 presents the related works in the literature. Section 3 offers the probability model of the minimal eviction set. We give a detailed introduction to the proposed cache structure in Section 4. Security analysis and performance evaluation are then carried out in Section 5, followed by a conclusion presented in Section 6.

2. Background

In this chapter, we introduce background knowledge and related works on the cache structure, the conflict-based cache side-channel attacks, minimal eviction set, mitigation strategies based on encrypting the address mapping relationship of physical-to-cache, and virtual memory translation.

2.1. The Cache Structure

The basic unit in the cache is the cache line, which stores the data transferred between the memory and the processor in fixed-length blocks. Three different mapping structures, fully associative mapping, set-associative mapping, and direct mapping, are used to place the cache line [1]. The fully associative mapping cache adopts the policy in which a block of the main memory can be mapped to any available cache line. Upon receiving a request, the cache needs to search for the correct data from the entire cache. This structure, however, greatly reduces the speed of location search. Different from the fully associative cache, the direct mapping cache sets the specific block of the main memory to be mapped to only one particular line of the cache, i.e., the mapping relationship from the physical address to the cache line is fixed.

Furthermore, the set-associative mapping cache is a trade-off between direct mapping and fully associative mapping. Several cache lines are grouped into sets. The direct mapping policy is applied between sets, whereas the fully associative mapping policy is employed within each set. In other words, a certain segment of memory in the main memory can only be mapped to fixed cache sets but not the fixed cache line. Compared with other cache structures, the set-associative mapping provides a higher performance and lower access latency.

As Figure 1 shows, the set-associative mapping cache can be imagined as an $n \times a$ matrix. The cache is divided into n sets and each set contains a cache lines. The same location in every set can form a new group named way. Many processor caches in today's designs are either direct-mapped, two-way set-associative, or four-way set-associative [14].

The address is usually divided into several parts in the set-associative mapping cache, that is *index*, *tag*, and *block_offset*. Each set includes several cache lines with the same *index* in each way. The *index* is then used to locate where the cache line is on each way. The *tag* is a part of the address stored in the cache. The *block_offset* is used to locate the requested data segment in each cache line.

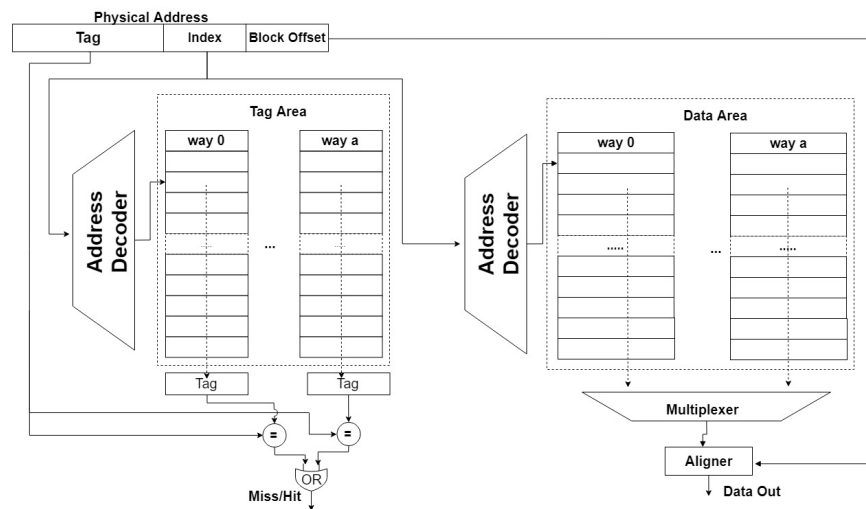


Figure 1. Serialized access of set-associative mapping cache.

When being accessed, a cache set can be found according to the *index*. If the *tag* does not match, it is a cache miss and the data is requested from the next level in the memory hierarchy. If the *tag* matches, it is a cache hit and the appropriate cache line is returned to the multiplexer. The multiplexer then chooses the correct bytes according to the *block_offset* and returns the data to the processor.

2.2. Conflict-Based Cache Side-Channel Attacks

To build the cache state-based side-channel, the attackers need to modify the state of the cache [10]. During the probe phase, the attackers measure timing differences to obtain leaked private data. There are two ways to modify the state of the cache [15–17]. One is to update the specified cache line, called updated-based cache side-channel attacks; the other is to initiate address conflict in the cache, called conflict-based cache side-channel attacks [18]. We are focused on the latter one.

Conflict-based cache side-channel attacks [19] attempt to evict a cache line from the cache by executing memory access instructions. In the preparatory phase, the attackers fill up a block of cache line with known data. In the attack phase, the attackers modify the cache line by loading new data and evicting the known data. During the probe phase, the attackers detect the evicted cache line to obtain the private data.

As an example, in [20], the authors proposed a method of bit-wise probing. In the preparation phase, the attackers fill up a block of cache lines with known data. During the attack phase, the attackers can modify a cache line by loading new data. If the first bit of the private data is 1, the specified cache line will be accessed and the known data will be evicted. If the first bit is 0, the specified cache line will not be accessed. In the probe phase, the attackers can determine whether the state of the specified cache has been changed. If it has, the modulated data is 1. If it has not, the modulated data is 0. By repeating this process multiple times, the attackers can obtain multi-bit private data.

To deduce private data, both updated- and conflict-based cache side-channel attacks require monitoring of the cache's state. It is also necessary to modify the state of the cache so that the attackers can get the leaked private data in the probing phase.

2.3. Minimal Eviction Set

To initiate a conflict-based cache side-channel attacks, the attackers must first induce an address conflict within their monitoring range. The minimal eviction set provides a suitable monitoring range. This is achieved by randomly selecting a set of cache lines, known as the candidate set, and then refining it until the minimal eviction set is reached.

Identifying a minimal eviction set is the most basic building block for the attackers to detect cache evictions. To accomplish this, the attackers must first load known data into

the minimal eviction set and then trigger an eviction within it. By carefully monitoring the minimal eviction set, the attackers can determine which cache line is being evicted during the attacks. This information is crucial for executing subsequent steps in the cache side-channel attacks.

The primary objective of finding a minimal eviction set is to induce an address conflict within the cache. When a cache miss occurs, the set-associative mapping cache selects a victim cache line to store the new or updated data based on the replacement algorithm. Specifically, the victim cache line is chosen from a cache set that shares the same index. If two physical addresses are mapped to the same set, a collision occurs, causing a conflict in the cache set during eviction.

2.4. Mitigation Strategies Based On Encrypting the Address Mapping Relationship of Physical-to-Cache

Encrypting the mapping from the physical address to the cache address has been widely used to resist conflict-based cache side-channel attacks [21,22]. By obfuscating the address mapping from the physical-to-cache, the random/encrypted mapping makes the data occupy the same cache line but has a different *index* so that the attackers cannot evict the specified cache line. This, in turn, makes it challenging for the attackers to identify and prune the eviction set.

Newcache [23] utilizes a randomized replacement strategy for the L1 cache. Newcache randomly selects a cache line as the victim to replace the new-updated data. During the cache addressing, all the cache lines should be accessed to determine the location of the destination data. Although the author of this paper uses a novel addressing structure called content addressable memory to reduce the addressing latency, it is still unsuitable for the large-capacity cache. It can only be used for structures with less cache capacity, such as the L1 cache.

Similar to Newcache, the random fill cache technique [21] utilizes a randomization strategy for selecting blocks to be cached. More precisely, it may not cache the requested block. The requested block is directly sent to the processor without being cached. To still benefit from the cache's performance, the random fill cache fills the cache with randomized fetches within a configurable neighborhood window of the missing memory blocks. Random fill cache requires both hardware and software changes.

RPCache [9] employs a permutation table that is indexed by the *index*. The physical-to-cache mapping for a process is stored in a permutation table. The index entry has a cache set index to which the address is mapped.

Ceaser [7] uses encryption to obscure the address mapping relationship between the physical and cache. The ceaser-s extends to the skewed associative cache. For attacks that require the virtual address to be mapped to the same set as the victim address on all partitions, remapping a line within every 100 accesses may only slow down by 1%.

Scattercache [8] enhances the cache structure of the skewed associative cache and uses the encrypted mapping function module to encrypt the physical address so that the attackers cannot determine the cache set to be filled.

2.5. Virtual Memory Translation

To achieve process security among the process, the operating system assigns an independent address space for every process used to address.

The memory management unit (MMU) in the processor translates the virtual address (VA) to the physical address (PA). The translation adopts the policy of having a lookup table. The supervisor address translation and protection (*satp*) register holds the physical page number (PPN) of the root page table. A VA is partitioned into several virtual page numbers (VPNs) and page offset. We take the Sv39 as an example. The 27-bit VPN is translated into a 44-bit PPN via a three-level page table, while the 12-bit page offset is untranslated. The VA is translated into a PA as in Figure 2. PT1/PT2/PT3 represent different page tables. This process is called page walk [24].

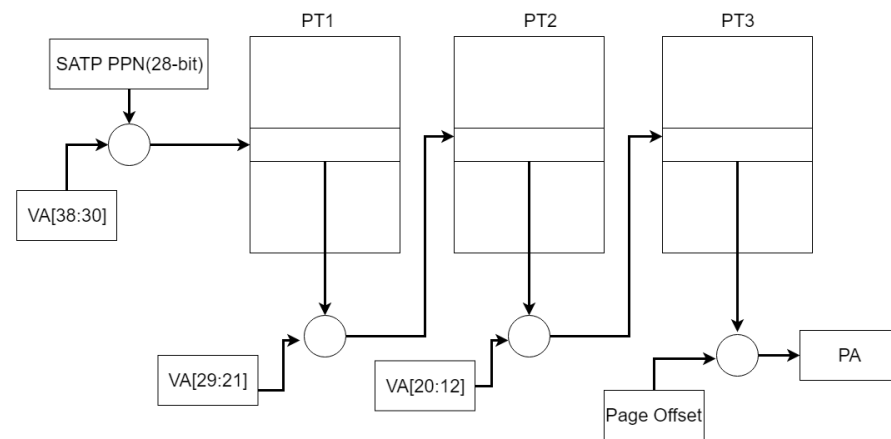


Figure 2. Virtual memory translation.

3. Probability Model of Finding the Minimal Eviction Set

In this section, we analyze the probability of obtaining a minimal eviction set.

3.1. Algorithms to Test the Eviction Set

Testing whether an eviction set exists in the candidate set is key to find the minimal eviction set. The literature [10] introduced three test algorithms as follows. The candidate set and the specified addresses are marked as *CS* and *X*, respectively.

Test 1: (1) The attackers measure the time to access a cache line, marking this latency as *Athreshold*. (2) Access *X*. (3) Access *CS*. (4) Access *X* again, measure the access latency of accessing *X*. If the latency exceeds *Athreshold*, the *CS* is an eviction set. This algorithm is aimed at the specified address.

Test 2: (1) The attackers measure the time to access cache lines with the same number as the candidate set. We marked this time as *Bthreshold*. (2) Access *CS*. (3) Access *CS*, again. If the access latency of (3) is larger than *Bthreshold*, then *CS* is an eviction set.

Test 3: (1) The attackers measure the time to access a cache line, marking this time as *Cthreshold*. (2) Access *CS*. (3) Access *CS* and measure access time of every address in *CS*. If the number of elements whose access latency is larger than *Cthreshold* is greater than the number of the cache way, then *CS* is an eviction set.

3.2. Pruning the Eviction Set

According to Vila's theory on cache-based side-channel attacks [10], the attackers can acquire the minimal eviction set through various methods, with "test-after-removing" being the most widely used approach. In this method, the attackers iteratively test all cache lines after removing a specific line. Firstly, they need to obtain a candidate set that ensures at least one eviction set can be found. Secondly, the attackers remove one cache line from the candidate set and test whether the eviction still exists in the remaining cache lines. If no eviction occurs, the removed cache line becomes one of the minimal eviction sets; otherwise, it is not. Finally, the attackers can obtain a minimal eviction set by performing these tests on all cache lines in the candidate set, one at a time.

3.3. Probability of Cache Eviction

The translation from the virtual address to the physical address can be viewed as a random function, which is the worst case from the attackers' point of view. However, since some address bits remain unchanged when translating the virtual address to a physical address, the randomness can be reduced by some technical means.

Let us focus on the bits located on $[P: 0]$ as shown in Figure 3. Physical memory is composed of multiple pages. The size of each page is 4 KB. For some large page tables, the size is 2 MB. To specify the position of the requested data on the page, the address bits located in $[P: 0]$ are used to indicate the offset of the requested data.

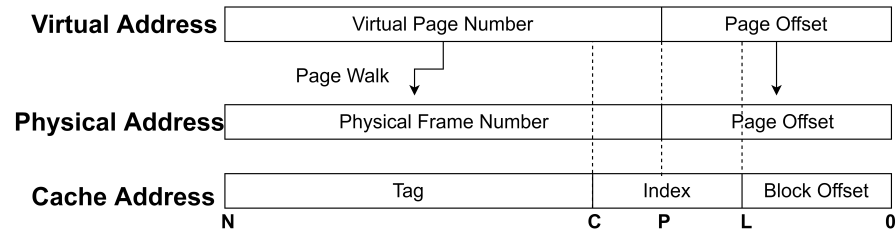


Figure 3. Address conflict.

It can be seen from Figure 3 that the *index* located in [C: L] is sharing the partial bits with the *Page Offset*. Since the *Page Offset* is identical in the address translation, not all of the virtual address bits need a translation. Note that the address bits are located in [P: L]. These address bits are located in the *Page Offset* of the virtual and physical addresses. Therefore, several *index* bits remain unchanged in the translation. These bits are referred as the controlled bits. The number of controlled bits significantly influences the success of finding the eviction set.

When trying to compute the probability, one can only consider the number of the bits located in [C: P]. The probability that two virtual addresses are mapped to the same cache set can be computed by counting the uncontrolled address bit since only the uncontrolled address bits can bring uncertainty to find the eviction set. We mark the number of address bits located in [C: 0] as C. With very large pages, for example, the number of bits located in [P: 0] is 21. Therefore, the probability of two virtual addresses mapping to the same cache set is $\frac{1}{2^{C-21}}$. The address bits located in [N: C] are the *tag* bits that are used to verify.

To illustrate the efforts of the attackers in obtaining a minimal eviction set, we analyzed the probability of cache conflict being triggered on purpose.

The attackers can target the specified address to prune the eviction set, the process of which can be considered as a binomial probability distribution. Therefore, aiming at X, the probability of getting a conflict in a random-formed candidate set P_a is given by:

$$P_a = 1 - \sum_{k=0}^{a-1} \binom{N}{k} P_t^k \times (1 - P_t)^{N-k} \quad (1)$$

where P_t is the probability that one cache line can collide with X, a is the *associativity*, and N is the size of the candidate set. The k is the number of cache lines colliding with X.

Furthermore, the attackers can target the arbitrary address. We refer to the calculation methods in [10], which briefly introduces the computation of the probability of targeting an arbitrary address. It relies on the Poisson distribution to estimate the probability as:

$$P_a(\exists i | N_i > a) = 1 - P(N_1 \leq a, \dots, N_B \leq a) \quad (2)$$

where N_i represents the number of addresses mapping to the i -th cache set, B is the number of the cache set, and a is the *associativity* of the cache.

3.4. Security of Address Encryption Mechanism

With the help of the address encryption mechanism, the probability of address conflict can be lower. However, it is never zero. The address encryption mechanism aims to obscure the mapping between the physical addresses and cache addresses, so the address bit the attackers can control is 0 in the worst case. We mark the number of address bits located in [C: L] as CL , while the minimal probability for address conflict is $\frac{1}{2^{CL}}$. Obviously, this probability relied on the structure parameters of the cache itself. So we give the figure about the probability of cache eviction with different structural parameters, as shown in Figures 4 and 5.

However, these figures show that the attackers have different probabilities of obtaining the minimal eviction set on different caches. Having a larger cache set, the cache can give

the attackers more difficulty in obtaining the minimal eviction set, while a smaller cache set does not. Furthermore, all of these caches can reach the peak of 1, which means the attackers can get one eviction set finally.

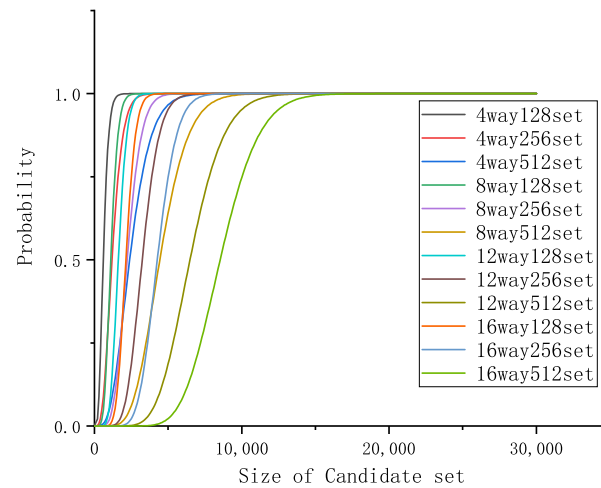


Figure 4. The probability of cache eviction with different cache structural parameters targeting specific addresses.

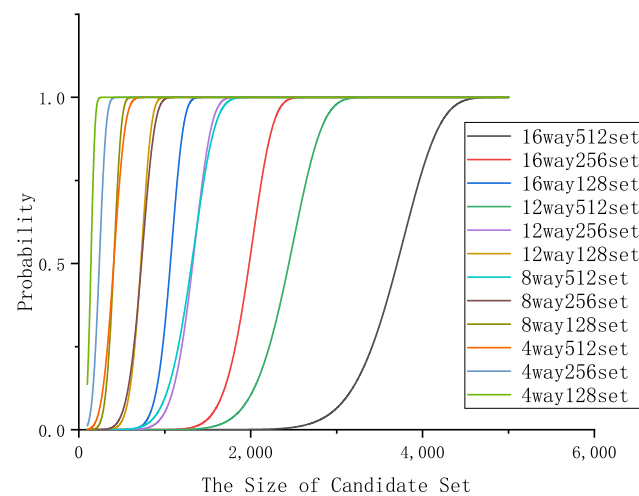


Figure 5. The probability of cache eviction with different cache structural parameters targeting arbitrary addresses.

4. Proposed Cache Structure

This section gives a detailed introduction to the cache structure integrated with the address encryption mechanism and timing noise extension mechanism. Since this cache structure retrofits the address encryption mechanism, we first introduce the address encryption mechanism we designed. Then, we give a detailed explanation of the timing noise extension mechanism.

4.1. The Address Encryption Mechanism

The address encryption mechanism we designed can encrypt all the physical addresses except the block_offset. To improve the encryption speed, the address encryption mechanism applies the XOR algorithm in the encryption.

The address encryption module, shown as $Encryption_0, \dots, Encryption_M$ in Figure 6, is configured for each cache way. When data at address x is requested, the address is encrypted to generate a new index E_{Index} , as follows:

$$E_{Index} = F_{encryption}(x) \quad (3)$$

To minimize the risk of the attackers obtaining the minimal eviction set, it is important to eliminate address bits that are under control. This can be achieved by dividing all requested address bits into T cells of the same granularity as the *index* bits, denoted as $\{E_1, E_2, \dots, E_T\}$. In addition, a series of random ciphers are generated by a random number generator, denoted as $\{E_{r1}, E_{r2}, \dots, E_{ra}\}$, where a is the associativity of the cache. This prevents the attackers from manipulating specific address bits to obtain the minimal eviction set.

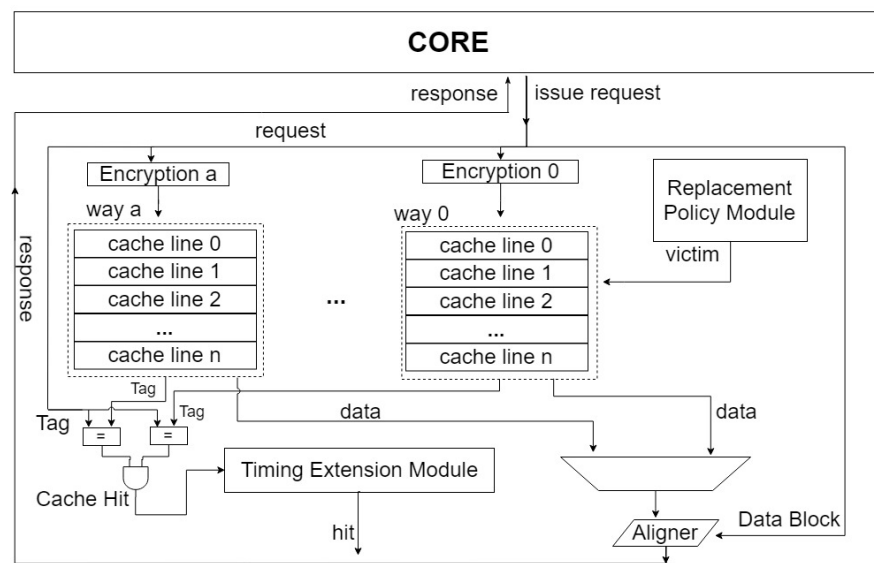


Figure 6. The structure of address encryption and timing noise extension mechanism.

The next step is to XOR the divided cells with the corresponding random ciphers to obtain the final result. Therefore, the calculation of E_{Index} for the i -th way is as follows:

$$E_{index_i} = (E_1 \oplus E_{ri}) + (E_2 \oplus E_{ri}) \cdots + (E_n \oplus E_{ri}) \quad (4)$$

Each component of the requested address is associated with an encrypted address using a specific algorithm. Even a slight change in the address bits can result in a different encrypted address. As the random cipher remains in an unknown state, the attackers are unable to access the encrypted cipher E_{index} and decipher it.

With the exception of the newly added encryption module, the other structures are similar to those of the original set-associative mapping cache. As shown in Figure 6, the address encryption mechanism proposed in this paper is integrated into the set-associative mapping cache. This mechanism divides cache lines into several sets and ways. Unlike traditional caches, the cache with the address encryption mechanism incorporates an encryption module in each way, with each module using a unique cipher.

However, the XOR operation may introduce some delays. As shown in Figure 7, due to the need for multiple XOR calculations in the encryption module circuit, this encryption process can cause a delay of two clock cycles. When requests are issued to the cache, their requested addresses are divided into multiple parts. In the encryption module, each part of these address bits is XORed with the random cipher generated by the random number

generator, which can be done in a single clock cycle. Then, the operation result is encrypted by ADD, which requires an additional clock cycle.

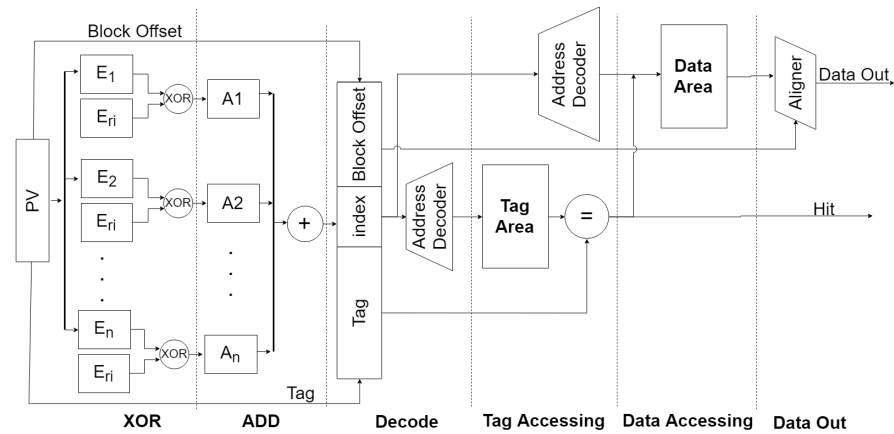


Figure 7. Serialized access of the address encryption mechanism.

The values $\{E_{r1}, E_{r2}, \dots, E_{rn}\}$ change periodically. Unlike the ScatterCache design, not all of the random ciphers need to be changed; only some of them do. The address encryption module randomly selects some address encryption modules and changes their random ciphers, as follows:

$$\{E_{r1c}, E_{r2c}, \dots, E_{rkc}\} = F_{random}(\{E_{r1}, E_{r2}, \dots, E_{rn}\}) \quad (5)$$

where the $\{E_{r1c}, E_{r2c}, \dots, E_{rkc}\}$ are the random ciphers that need to be changed and k is the number of random ciphers needs to be changed.

4.2. Timing Noise Extension Mechanism

Based on the above analysis, encrypting the mapping relationship between the physical address and the cache address can make it more difficult for the attackers to identify the minimal eviction set but it cannot completely prevent them from pruning the eviction set. As stated in [11], caches that use address encryption and cipher updates remain vulnerable to attacks. This is because some attacks do not require the use of a minimal eviction set and, even if they do, a 99% eviction rate may not be necessary. This undoubtedly reduces the difficulty for the attackers to implement attacks. According to the viewpoint proposed by [11], they believe that caches with address encryption and cipher updates still have security vulnerabilities. This mechanism generates cache misses randomly, resulting in unpredictable noise. For the attackers, such noise is a fatal error that can lead to incorrect conclusions [10].

Figure 6 illustrates the infrastructure of the timing noise extension mechanism, which consists of an independent circuit module within the cache. This module monitors all requests sent to the cache and blocks cache access once the number of memory accesses exceeds a certain *threshold*. The request is then forwarded to the next level of the memory unit. To reduce circuit resource costs, the cache reuses former cache lines to store the requested data. The *threshold* value is generated using a random number generator, which can generate the value randomly to control the frequency of the noise generation.

In Figure 8, the *counter* is utilized to keep track of the number of cache hits. Once the number of cache hits reaches the *threshold* generated by the *random number generator*, the *counter* triggers a cache miss and resets itself. However, random noise can cause the data to be reloaded, which can be resource-intensive. To address this issue, a *record table* has been designed to record the cache line and its corresponding physical address. This table enables the cache to reuse the cache line when the data is reloaded by looking up the *record table*.

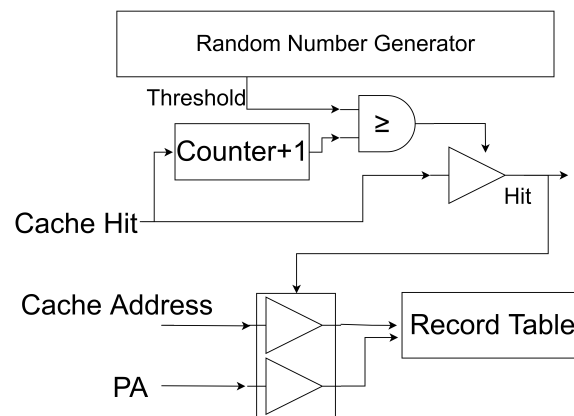


Figure 8. The timing noise extension mechanism.

4.3. The Frequency of Timing Noise Generating

The timing noise extension mechanism involves adding timing noise to memory access, which is essential for improving security. However, the frequency at which timing noise is generated must be carefully balanced against cache performance. If the frequency is set too high, it may result in a significant number of cache misses, which can adversely affect processor performance. Conversely, if the frequency is set too low, the attackers can exploit simple evasion attacks to eliminate the noise [11,25].

Figures 4 and 5 indicate that there is a greater probability of meeting one cache conflict when targeting an arbitrary address as opposed to a specific address. As a result, this paper only focuses on the attack model of targeting arbitrary addresses, which is considered to be the most dangerous scenario.

To determine the appropriate threshold for the timing noise extension mechanism, we need to estimate the number of memory accesses that the attackers may use. To be conservative, we assume that only one attempt is needed to determine the timing of eviction, rather than the typical tens or hundreds of attempts. Therefore, we can calculate the expected number of memory accesses (S) that the attackers may use as follows:

$$S = \frac{N}{P} \quad (6)$$

where N represents the size of the candidate set and P is the probability of obtaining an eviction in a randomly formed candidate set of size N . Using Formula (6), we have created a figure that shows the expected value of S for several different cache configurations, which is presented below:

As shown in Figure 9, the expectation S reaches a minimal value. To determine this minimal value, we take the derivative of S with respect to N :

$$S' = \begin{cases} -\frac{1}{P^2}, & P < 1 \\ 1, & P = 1 \end{cases} \quad (7)$$

From Formula (7), we can see that, when $P < 1$, then $S' < 0$; S decreases as N increases. Furthermore, since $P < 1$, then $S > N$. Furthermore, when $P = 1$, then $S = N$ while, in the case of $P = 1$, N gets the minimal value $N_{min, P=1}$, S also has the minimal value S_{min} , i.e., $N_{min, P=1} = S_{min}$. To make a trade-off between cache performance and security, we set the $threshold = S_{min}$. The reasons why we come to this conclusion are below.

1. To ensure that timing noise can exist in every test on the candidate set, the interval between two timing noises must be less than or equal to the minimal size of memory access during the execution of a test algorithm.

2. To ensure that the frequency of the timing noise generated is consistent with the cache evictions. The timing noise generated is within *threshold* accesses, while the attackers may encounter a cache eviction in S_{min} accesses.

Under these considerations, even if there is a memory access delay, the attackers cannot determine whether the delay is caused by a cache conflict or a timing noise. By drawing an incorrect inference, the attackers cannot prune the candidate set. This paper only considers the case under the most-unsafe condition of $threshold = S_{min}$.

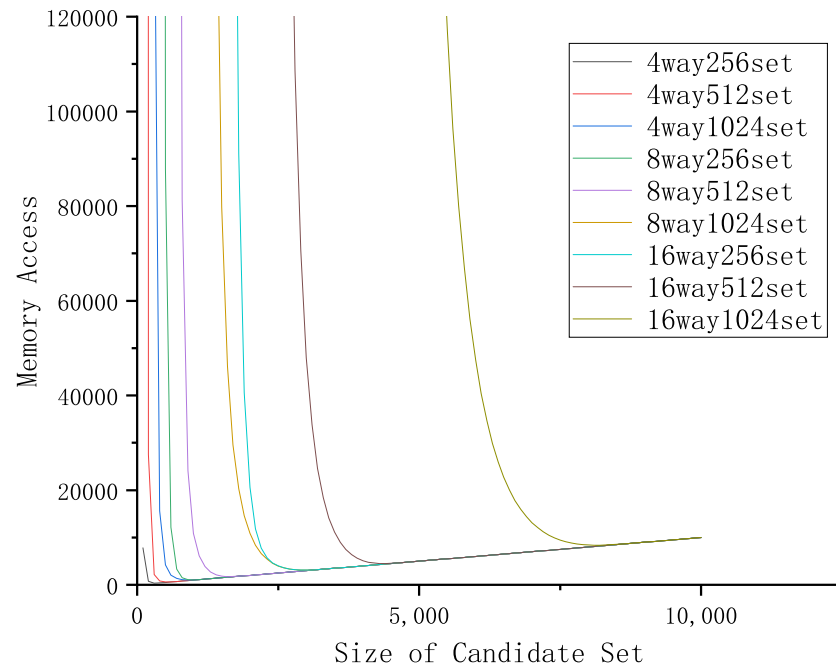


Figure 9. The memory access that the attackers may be expected to consume to initiate at least one cache eviction.

5. Evaluation

In this section, we conducted experiments and analyses about the cache proposed in this paper.

5.1. Security Evaluation

To prune candidate sets, the attackers must test the candidate set by removing a specific cache line. However, since the accuracy of memory access timing measurement is critical to the test algorithm, the attackers must conduct multiple tests to filter out noise.

Let P_a denote the probability that the attackers do not encounter timing noise, N denote the size of the randomly formed candidate set, and S_t denote the number of memory accesses that the attackers perform.

1. When $N \geq threshold$, according to the circuit structure proposed in this article, the probability of the attackers encountering random noise reaches 1. This implies that the attackers will inevitably encounter random noise.

2. When $N < threshold$, the attackers are unlikely to obtain a cache eviction in a single test. This requires the attackers to choose multiple candidate sets, leading to $S_t \gg threshold$. We denote the probability of encountering noise as P_n , so the probability of not encountering noise is $(1 - P_n)$. Thus, the probability of the attackers not encountering random noise is:

$$P_a = (1 - P_n)^{S_t} \quad (8)$$

Our random noise mechanism generates noise once every *threshold* cache accesses, where $threshold = S_{min}$ as described earlier. Consequently, $P_n = \frac{1}{S_{min}}$. Since $(1 - P_n) <$

1, increasing S_t reduces P_a . When $S_t = S_{min}$, P_a has its maximal value. To assess the worst-case scenario for P_a , we set S_t to approximately equal $N_{min,P=1}$, i.e., S_t approaches $N_{min,P=1}$ infinitely.

From the above analysis, Formula (8) can be rewritten as:

$$P_a = \left(1 - \frac{1}{N_{min,P=1}}\right)^{N_{min,P=1}} \quad (9)$$

Because caches with different structural parameters have varying $N_{min,P=1}$ values, we use computer simulations to compute P_a . As shown in Figure 10, P_a cannot exceed 50%, indicating that the attackers are likely to encounter timing noise.

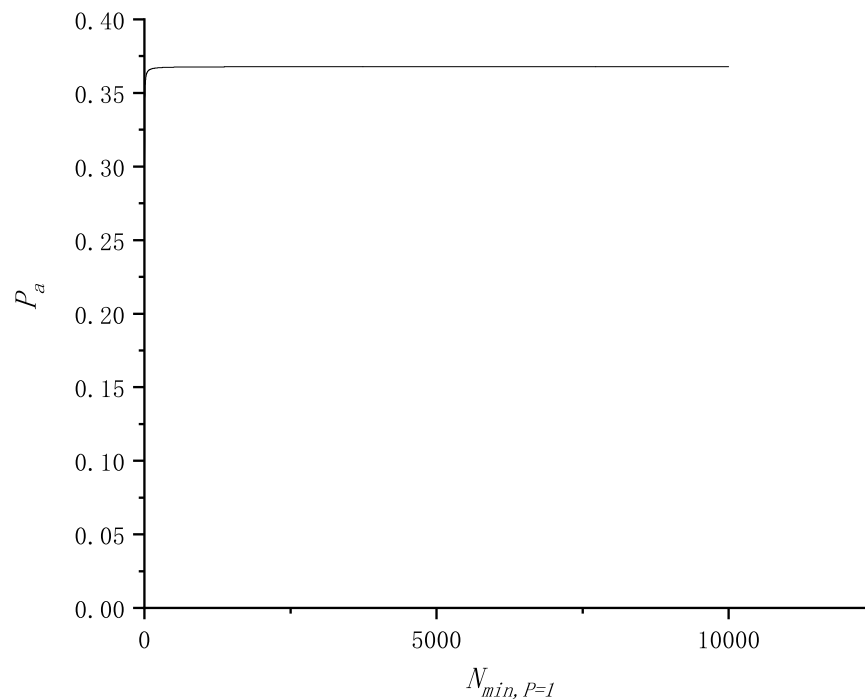


Figure 10. The probability of the attackers not encountering a timing noise.

When pruning the candidate set, irregular and high-frequency timing noise can disrupt the attackers' calculations [10]; while certain statistical methods can mitigate these noise effects [8], the probability of the attackers making incorrect judgments remains high when dealing with long-duration and low-regularity noise. Consequently, the attackers may be unable to identify the minimal eviction set.

When $N < threshold$, the probability that the attackers do not encounter random noise cannot be reduced to zero. In practice, the attackers may encounter more complex forms of noise than just the timing noise generation module. Other sources of noise include the translation lookaside buffer, instruction cache, and cache prefetching [7,8]. These additional sources of noise can significantly increase the difficulty of initiating successful attacks.

5.2. Performance Evaluation

To model the proposed cache, we made modifications to the gem5 simulator. The parameters of the gem5 simulation that we used are listed in Table 1. For the processor model, we opted for the O3CPU model.

The Standard Performance Evaluation Corporation (SPEC) CPU suite is a well-established set of compute-intensive benchmarks used for testing processor performance. SPEC CPU2017, publicly released in 2017, includes a range of state-of-the-art applications such as alpha-beta tree search and pattern recognition in *deepsjeng*, Monte Carlo tree search, game tree search, and pattern recognition in *leela*, as well as a recursive solution generator in

exchange2. To evaluate the proposed cache, we ran the SPEC CPU2017 rate applications, using SimPoint in gem5 to accelerate the simulation. We tested each workload with at least 20 million instructions, using the first 2 million instructions to warm up the cache and the remaining 18 million to collect performance statistics.

To measure the overall performance, we report the normalized performance of the proposed design to *base* that is a conventional and insecure processor.

Table 1. System settings for the simulated architecture.

Parameter	Value
Architecture	1 Core, O3 CPU model
Core	Out-of-order, no SMT, 100 load queue entries, 100 store queue entries, 300 ROB entries
Data Cache	512KB, 16-way, cache line size is 64-bit
Instruction Set	RISC-V ISA

The address encryption mechanism can optimize the placement of cached data, resulting in improved cache hit rates. The placement of cached data is critical for achieving high performance in modern processors, as it involves maximizing the association between cache lines to exploit spatial locality. There are various optimization strategies for cache placement, such as page coloring [26] and other methods. One such strategy is the skewed associative cache [27,28], which skews the cache to improve cache hit rates.

In Figure 11, we observe that some benchmarks, such as *mcf*, *xalancbmk*, and *xz*, exhibit a small decrease in cache miss-rates. This is due to the address encryption mechanism employed, which optimizes the placement of cached data. Unlike traditional cache structures, this mechanism aligns with the skewed associated cache [29], placing adjacent data to improve spatial locality. Overall, this approach reduces the cache miss rate for most benchmarks. However, the integrated mechanism incurs a 2.9% overhead in CPI, as seen in Figure 11.

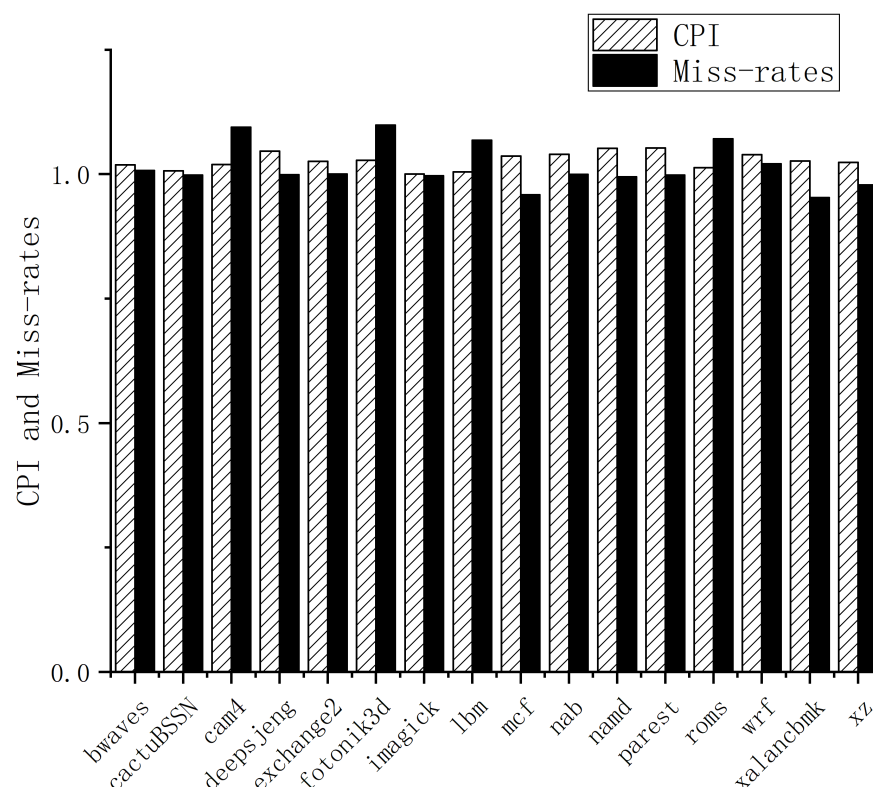


Figure 11. The miss-rates and CPI of address encryption and timing noise extension mechanism.

5.3. Delay of Encryption Circuit

The cache performance is degraded by the access delay caused by the address encryption mechanism. As shown in Figure 11, while the miss-rates of some benchmarks decrease, their performance does not improve, such as *xz*, *xalancbmk*, and *mcf*. This is because the address encryption mechanism introduces a delay in the cache access. Compared to traditional caches, the address encryption mechanism requires two more clock cycles to access the cache due to the additional address encryption requirements. Therefore, this performance overhead is inevitable. From Figure 11, we can see that the integrated mechanism has a 2.9% overhead in CPI, even though it has a better cache hit rate.

5.4. Threshold

The *threshold* represents not only the interval between cipher updates, but also the interval between random noise generation in our design. Specifically, in our design, we set the *threshold* to be equal to S_{min} . Formula (6) shows that the attackers can choose a smaller candidate set if the probability of address conflict is bigger, while the address encryption mechanism can decrease the probability. Incorporating the address encryption mechanism requires the attackers to form a larger candidate set, resulting in an increase in S_{min} . This increase means that a cache with the encryption mechanism may be able to satisfy security requirements using a bigger threshold compared to a cache without it. Therefore, reduced performance overhead may be caused.

6. Conclusions

While traditional address encryption methods can increase the difficulty for the attackers to find the minimal eviction set, they do not completely prevent the attackers from building side-channels. To address this issue, this paper proposes a timing noise extension mechanism that introduces random timing noise to prevent the attackers from measuring the timing of cache access.

Our study compared two methods: one that only encrypts the mapping relationship between physical addresses and cache addresses, and another that integrates the address encryption mechanism and the timing noise extension mechanism. We found that the latter method is more secure. Specifically, when the timing noise generating interval matches the minimal size of the candidate set, the resulting timing noise introduces a level of randomness that can cause fatal errors for the attackers attempting to find the minimal eviction set. In fact, existing algorithms are unable to obtain the minimal eviction set under these conditions.

This paper also takes into account the impact of delay caused by the encryption module in order to simulate the circuit's behavior in practical settings. Despite the added delay, the integrated mechanism, which includes the address encryption mechanism that improves cache hit rate, only incurs a 2.9% overhead on CPI.

Author Contributions: Conceptualization, D.W., S.T. and W.G.; Methodology, D.W.; Software, D.W.; Validation, D.W.; Investigation, D.W.; Data curation, D.W. and S.T.; Writing—original draft preparation, D.W., S.T. and W.G.; Writing—review and editing, W.G. and S.T.; Visualization, D.W., S.T. and W.G.; Supervision, W.G.; Funding acquisition, W.G. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported in part by the China Ministry of Science and Technology under Grant 2015GA600002.

Acknowledgments: The authors would like to thank Wanlin Gao for funding authorization. Additionally, we are fortunate and thankful for all the advice and guidance we have received during this work.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Mahmood, S.; Herman, G.L. A modular assessment for cache memories. In Proceedings of the 52nd ACM Technical Symposium on Computer Science Education, Virtual, 13–20 March 2021; pp. 1089–1095.
2. Kocher, P.; Jaffe, J.; Jun, B. Differential power analysis. In *Advances in Cryptology—CRYPTO’99, Proceedings of the 19th Annual International Cryptology Conference, Santa Barbara, CA, USA, 15–19 August 1999*; Proceedings 19; Springer: Berlin/Heidelberg, Germany, 1999; pp. 388–397.
3. Tsunoo, Y.; Saito, T.; Suzaki, T.; Shigeri, M.; Miyauchi, H. Cryptanalysis of DES Implemented on Computers with Cache. In *Cryptographic Hardware and Embedded Systems—CHES 2003*; Walter, C.D., Koç, Ç.K., Paar, C., Eds.; Springer: Berlin/Heidelberg, Germany, 2003; pp. 62–76.
4. Brumley, B.B.; Hakala, R.M. Cache-Timing Template Attacks. In *Advances in Cryptology—ASIACRYPT 2009*; Matsui, M., Ed.; Springer: Berlin/Heidelberg, Germany, 2009; pp. 667–684.
5. Yarom, Y.; Genkin, D.; Heninger, N. CacheBleed: A Timing Attack on OpenSSL Constant Time RSA. In *Cryptographic Hardware and Embedded Systems—CHES 2016*; Gierlichs, B., Poschmann, A.Y., Eds.; Springer: Berlin/Heidelberg, Germany, 2016; pp. 346–367.
6. Yan, M.; Sprabery, R.; Gopireddy, B.; Fletcher, C.; Campbell, R.; Torrellas, J. Attack directories, not caches: Side channel attacks in a non-inclusive world. In Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 19–23 May 2019; pp. 888–904.
7. Qureshi, M.K. CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping. In Proceedings of the 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Fukuoka, Japan, 20–24 October 2018.
8. Werner, M.; Unterluggauer, T.; Giner, L.; Schwarz, M.; Gruss, D.; Mangard, S. ScatterCache: Thwarting Cache Attacks via Cache Set Randomization. In Proceedings of the 28th USENIX Security Symposium (USENIX Security 19), Santa Clara, CA, USA, 14–16 August 2019; pp. 675–692.
9. Wang, Z.; Lee, R. New cache designs for thwarting software cache-based side channel attacks. In Proceedings of the 34th Annual International Symposium on Computer Architecture, Orlando, FL, USA, 17–21 June 2007; Volume 35, pp. 494–505. [\[CrossRef\]](#)
10. Vila, P.; Köpf, B.; Morales, J.F. Theory and Practice of Finding Eviction Sets. In Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 19–23 May 2019; pp. 39–54.
11. Song, W.; Li, B.; Xue, Z.; Li, Z.; Wang, W.; Liu, P. Randomized last-level caches are still vulnerable to cache side-channel attacks! However, we can fix it. In Proceedings of the 2021 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 24–27 May 2021; pp. 955–969.
12. Liu, F.; Wu, H.; Lee, R.B. Can Randomized Mapping Secure Instruction Caches from Side-Channel Attacks? In Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy, Portland, OR, USA, 14 June 2015; Association for Computing Machinery: New York, NY, USA, 2015. [\[CrossRef\]](#)
13. Purnal, A.; Giner, L.; Gruss, D.; Verbaudhede, I. Systematic analysis of randomization-based protected cache architectures. In Proceedings of the 2021 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 24–27 May 2021; pp. 987–1002.
14. Solihin, Y. *Fundamentals of Parallel Multicore Architecture*; CRC Press: Boca Raton, FL, USA, 2015.
15. Kocher, P.; Genkin, D.; Gruss, D.; Haas, W.; Yarom, Y. Spectre Attacks: Exploiting Speculative Execution. *Commun. ACM* **2018**, *63*, 93–101. [\[CrossRef\]](#)
16. Thirumalai, C.S. Review on non-linear set associative cache design. *Int. J. Pharm. Technol.* **2016**, *8*, 5320–5330.
17. McIlroy, R.; Sevcík, J.; Tebbi, T.; Titzer, B.L.; Verwaest, T. Spectre is here to stay: An analysis of side-channels and speculative execution. *arXiv* **2019**, arXiv:1902.05178.
18. Yarom, Y.; Falkner, K.E. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, 20–24 August 2014.
19. Canella, C.; Bulck, J.V.; Schwarz, M.; Lipp, M.; von Berg, B.; Ortner, P.; Piessens, F.; Evtushkin, D.; Gruss, D. A Systematic Evaluation of Transient Execution Attacks and Defenses. In Proceedings of the 28th USENIX Security Symposium, Santa Clara, CA, USA, 14–16 August 2019.
20. Lipp, M.; Schwarz, M.; Gruss, D.; Prescher, T.; Haas, W.; Fogh, A.; Horn, J.; Mangard, S.; Kocher, P.; Genkin, D.; et al. Meltdown: Reading Kernel Memory from User Space. In Proceedings of the 27th USENIX Security Symposium (USENIX Security 18), Baltimore, MD, USA, 15–17 August 2018.
21. Liu, F.; Lee, R.B. Random Fill Cache Architecture. In Proceedings of the IEEE/ACM International Symposium on Microarchitecture, Cambridge, UK, 13–17 December 2014.
22. Liu, F.; Hao, W.; Mai, K.; Lee, R.B. Newcache: Secure Cache Architecture Thwarting Cache Side-Channel Attacks. *IEEE Micro* **2016**, *36*, 8–16. [\[CrossRef\]](#)
23. Wang, Z.; Lee, R.B. A novel cache architecture with enhanced performance and security. In Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-41 2008), Lake Como, Italy, 8–12 November 2008.
24. Talluri, M.; Hill, M.; Khalidi, Y.A. A New Page Table for 64-bit Address Spaces. *Oper. Syst. Rev.* **1995**, *29*, 184–200. [\[CrossRef\]](#)
25. Lesage, B.; Griffin, D.; Altmeyer, S.; Cucu-Grosjean, L.; Davis, R.I. On the analysis of random replacement caches using static probabilistic timing methods for multi-path programs. *Real-Time Syst.* **2018**, *54*, 307–388. [\[CrossRef\]](#)
26. Xiao, Z.; Dwarkadas, S.; Kai, S. Towards practical page coloring-based multicore cache management. In Proceedings of the 2009 EuroSys Conference, Nuremberg, Germany, 1–3 April 2009.

27. Seznec, A.; Bodin, F. Skewed-associative Caches. In Proceedings of the PARLE '93, Parallel Architectures and Languages Europe, 5th International PARLE Conference, Munich, Germany, 14–17 June 1993.
28. Seznec, A. A case for two-way skewed-associative caches. *ACM SIGARCH Comput. Archit. News* **1993**, *21*, 169–178. [[CrossRef](#)]
29. Tan, Q.; Zeng, Z.; Bu, K.; Ren, K. PhantomCache: Obfuscating Cache Conflicts with Localized Randomization. In Proceedings of the 2020 Network and Distributed System Security Symposium, San Diego, CA, USA, 23–26 February 2020.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.