UNIVERSITY OF SOUTHAMPTON

Countermeasures for Cache Contention-Based Attacks Based On Randomisation Remapping

by

Xiao Liu
ORCID: 0000-0002-5928-4293

A thesis submitted for the
degree of Doctor of Philosophy

in the
Faculty of Engineering and Physical Sciences
School of Electronics and Computer Science

November 2023

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING AND PHYSICAL SCIENCES
SCHOOL OF ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

by Xiao Liu

Many cache designs have been proposed to guard against contention-based side-channel attacks. Specifically, the last-level cache, which is often a shared cache between different users. One type of well-known cache is the randomisation remapping cache. For example, the CEASER-S cache applies an encryption cypher with a periodically changing key as a cache indexing function. By decreasing the re-keying period, CEASER-S can defeat even a more aggressive contention-based attack. However, this can lead to performance degradation. Balancing the performance and the security against contention-based attacks becomes an essential consideration of the cache design.

In this thesis, we propose a novel cache configuration, logical associativity. By applying this configuration, we propose two secure randomisation remapping cache designs against contention-based attacks. The first cache we propose is the CEASER-SH cache, which is based on the CEASER-S cache. This cache allows the cache line to be placed not only in its mapped cache set but also in the subsequent cache sets. By enlarging the possible placement positions of the cache line, contention-based attacks are mitigated. Hence, the cache does not need to decrease the re-keying period significantly which would cause significant performance degradation. From the simulation results, for example, compared with CEASER-S, CEASER-SH with a logical associativity of 2 can reduce the miss rate by about 26% and the CPI by about 0.8% while maintaining the same security level against an aggressive Prime+Probe attack.

The second secure cache we propose is the Skewed Elastic-Associativity Cache (SEA cache). Unlike from CEASER-SH, this cache allows each user or each process to have different local logical associativity settings. Hence, only some users or processes that request extra protection against contention-based attacks are protected with high logical associativity. Other users can access the cache, or other pages can be accessed in the cache with lower latency and higher performance. The simulation results show that the SEA cache can outperform the CEASER-SH cache in terms of normal user's performance and overall security against contention-based attacks with minor extra power consumption. For example, the SEA cache with logical associativity of 1 for normal protection users and 16 for high protection users achieves better protection against contention-base attacks and about 0.4% CPI degredation in the normal user's core with just 0.01W extra power, compared to the CEASER-SH cache with logical associativity of 8.

Contents

List of Figures

List of Tables

Declaration of Authorship

I declare that this thesis and the work presented in it is my own and has been generated by me as the result of my own original research.

I confirm that:

1. This work was done wholly or mainly while in candidature for a research degree at this University;
2. Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
3. Where I have consulted the published work of others, this is always clearly attributed;
4. Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
5. I have acknowledged all main sources of help;
6. Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
7. None of this work has been published before submission

Signed:.. Date:..................

Abbreviations

AES	Advanced Encryption Standard
BIP	Bimodal Insertion Policy
BOOM	Berkeley Out-of-Order Machine
CoW	Copy-on-Write
CPI	Cycles Per Instruction
CPU	Central processing unit
FCE	Fully Congruent Eviction
I/O	Input/Output
IPC	Instructions Per Cycle
LLBC	Low Latency Block Cipher
LLC	Last-level cache
LRU	Least Recently Used
LFU	Least Frequently Used
L1	Level 1
L2	Level 2
L3	Level 3
MSHR	Miss Status Holding Registers
MPKI	Misses Per Kilo Instructions
OOO	Out-of-order
OPT K	Optimal K Value
PCE	Partially Congruent Eviction
PDK	process design kit
PPP	Prime+Prune+Probe
RKP	Re-keying Period
Reg	Register
RSA	Rivest–Shamir–Adleman
Resp	Response
S	Set
SDID	Security Domain Identifier
Sel	Selection
SEA	Scattered Elastic-Associativity
SHA	Secure Hash Algorithm

SHARP	Secure Hierarchy-Aware Cache Replacement Policy
TDID	Trusted Domain Identifier
VM	Virtual Machine
W	Way

Acknowledgements

This PhD journey has been an immense challenge for me, particularly given the circumstances of the Covid-19 pandemic, during which most of my research was conducted. I am sincerely grateful to all those who assisted and supported me throughout my research project. Without their help, I would have never been able to achieve this milestone.

First and foremost, I extend my deepest appreciation to my supervisors, Professor Mark Zwolinski and Dr Basel Halak, for their selfless support and unwavering encouragement during my research. I am indebted to Professor Mark Zwolinski for his invaluable guidance and inspiration, and to Dr Basel Halak for providing me with valuable feedback and advice. Without their assistance, pursuing a PhD would have remained an elusive dream. I would also like to extend my appreciation to the examiners who have provided their feedback on this thesis: Dr Leonardo Aniello and Professor Konstantinos Markantonakis. Their invaluable suggestions helped me to complete this work.

I am also grateful to Dr Shiyan Hu and Dr Leonardo Aniello for their invaluable suggestions for my nine-month and eighteen-month reports, which laid the foundation for this work.

Special thanks go to my colleagues, Dr Dongyao Zhai and Dr Haibo Su, for their advice not only on research matters but also on my future career prospects. I would also like to express my gratitude to Yiming Liu, Jiahao Pan, and Haotian Duan, whose numerous forms of assistance have been invaluable.

Furthermore, many thanks to Shiroha Takahara and Umi Takahara from SummerPockets, for their encouragement during moments of difficulty that I could not overcome alone.

Lastly, I want to express my deepest appreciation and love to my parents Haiping Liu and Zhanfei Liu, my grandparents, and my wife Hongyu Tu. Your care and love have been instrumental in shaping me into the person I aspire to be.

Chapter 1

Introduction

1.1 Cache

In computer systems, there are generally three basic units, as shown in Figure 1.1. These are the CPU, memory and I/O systems. The CPU does calculations based on the instructions and data. A memory stores these instructions and data. Finally, the I/O moves the information in and out of the system [56].

The operation of the computer can be briefly summarised as follow: From the I/O system, the instructions and data are sent into the computer system. Then the information is stored in the memory. Before the CPU executes the instructions or performs a calculation on any data, the memory must send these instructions or data to the CPU. Otherwise, the CPU may require a stall or move to other tasks that do not depend on the instructions or data currently stored in the CPU registers. However, the main memory is typically large and can take a very long time to access any data. As a result, the speed of transmitting the data from the main memory to the CPU limits the entire computer's performance. This is also known as the Von Neumann bottleneck [8]. To tackle the problem, an effective method is adding a cache that is smaller but much faster than the main memory to temporarily store a copy of some chunks of instructions and data that the CPU may soon require. The existence of caches reduces the overall fetching time and dramatically improves the computer's performance. Normally, a processor has multiple cache levels, especially for a multi-core processor. The last-level cache refers to the cache, which is the farthest to the cores. It is generally shared between all cores. We will explain the architecture of caches further in Section 2.1.

Nevertheless, the cache may expose users' security-sensitive data under cache timing side-channel attacks to an attacker, for example, in contention-based attacks. This is normally caused by the cache sharing between users. Many countermeasure designs proposed to defeat or mitigate these attacks. We will explain this in detail in Sections 2.4 and 2.5.

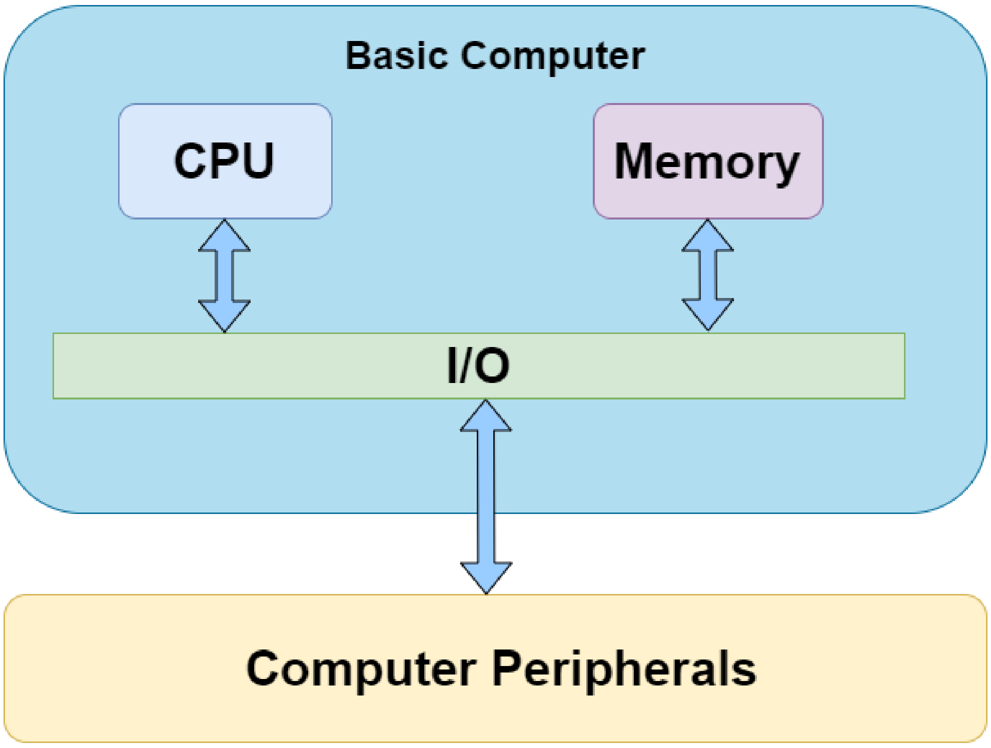


FIGURE 1.1: A basic computer with three essential units. Computer Peripherals are non-essential units.

1.2 Cloud Computing and Virtualisation

Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management effort or service provider interaction [48]. Nowadays, this model has been widely applied to many areas, including industry and personal use. Cloud computing makes working from home possible, guaranteeing normal operations for many industries (especially during the Covid-19 pandemic). For personal use, with the well-developed communication technology, cloud storage and cloud gaming have become the trend. In general, cloud computing has become an essential resource in the world.

However, these real applications do not explain the benefits of cloud computing. There are two core advantages. The first advantage is efficiency. An individual user typically cannot consume all resources on a computer, including computing power, storage, network etc. This causes resource waste and high capital cost. To effectively manage the utilisation of hardware resources, cloud computing allows sharing of hardware resources between different users. This also helps decrease the cost of hardware resources. The other advantage is elasticity. A user may have various tasks during different periods. These specific workloads may require contrasting resources. To achieve this, the user must upgrade the existing physical hardware. On the other hand, scaling the hardware resources in and out is simple on the cloud.

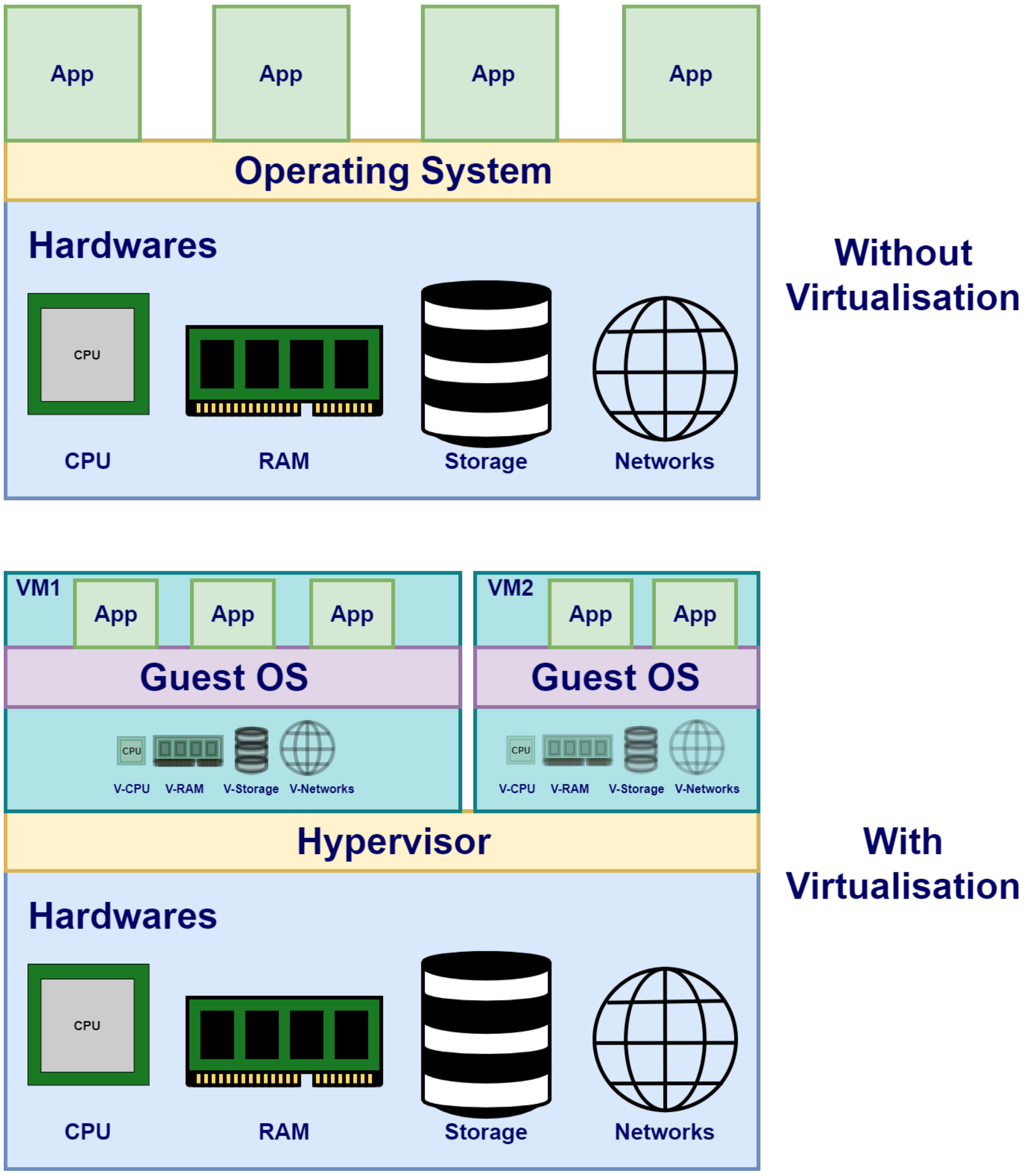


FIGURE 1.2: A comparison of a non-virtualised server (Above) and virtualised server (Below). VM1 in the virtualised server has been allocated more resources than VM2.

For solving the scaling problem, cloud computing relies on a technology or technique called **virtualisation**. Virtualisation is the application of the layering principle through enforced modularity, whereby the exposed virtual resource appears identical to the underlying physical resource being virtualised [21]. By combining the support of both hardware and software, an extra layer is introduced between the actual hardware and the clients' applications, which enables the indirect utilisation of hardware resources.

The enforced modularity in virtualisation guarantees that clients cannot bypass the indirection layer. Hence, different users cannot directly access other users' data. This provides the basic security requirements for sharing the same hardware between different users. There are three types of virtualisation techniques: multiplexing, aggregation, and emulation [21]. Multiplexing indicates that other clients share the same hardware resources. The sharing is achieved either in space or time. For example, the memory usage for an individual client can be limited, which is space multiplexing. For a deeper understanding of virtual memory, see Section 2.2.1. Another example is time multiplexing, where multiple users share the same core. The utilisation of the core needs to be scheduled in time. The second type, aggregation, combines and integrates multiple hardware resources as one and serves the same client. The last type, emulation, differs from the previous two. It usually serves clients with a different architecture than the current physical resources. An example of software that supports such virtualisation is QEMU [11].

Figure 1.2 compares the non-virtualised server and the virtualised server. The virtualised server is split into multiple virtual machines. After applying the virtualisation to the server, the physical machine can be used by different users. Each of the sub-machines is called a virtual machine. From a user's perspective, they cannot distinguish these virtual machines and the physical machines. Because of the abstraction layer of virtualisation, each virtual machine is isolated from others. To manage these virtual machines, software called a hypervisor [21] controls the operation of all virtual machines. Each virtual machine can have its distinct OS. We explain hypervisors and virtual machines in Section 2.2.2.

1.3 Information Security of Cloud

Information security is a major consideration of cloud computing and storage, and includes three distinct functions: access control, secure communications, and protection of private data [65]. Access control defines which user can use or access the computing system or the data. As we have discussed in Section 1.2, virtualisation provides the essential framework for distinguishing between clients. Hence, the access control of users can be implemented. Secure communication indicates the security of data transmission between cloud servers and the user's machine. This is often achieved by performing encryption and decryption on the transmitted data. Finally, protecting private data ensures that even if different users share the same physical resources, the private data should only be accessed by the user themselves. The confidentiality of private data is the basis of the cloud. Without this protection, any users or the service supplier can extract secret contents without permission.

Furthermore, the attacker can violate the user's privacy [89]. Implementing protection for private data always requires extra effort, which may include extra computational power and hardware overheads. Therefore, an important evaluation of protecting private data is the performance degradation and the security enhancement. To reduce performance degradation, private data is typically divided into levels requiring different security protections [65]. Thus, different security levels can have different balances between performance and security.

In this research project, we focused more on protecting private data. As mentioned above, the difficulty of protecting private data is that the attacker and the victim share the same physical resources, such as the cache, which is a part of the memory system. Since the last-level cache is shared between different cores and users, it is usually targeted by the attacker. One typical attack, which mainly targets the last-level cache, is the contention-based attack. We further explain this attack and other relevant attacks and how this type of attack can be leveraged to help the attacker retrieve a user's security-sensitive data in Section 2.3.

1.4 Motivations for research

As we have explained in Section 1.2, cloud computing, which is now widely used, has become essential for organising the modern commercial utilisation of computing system resources. However, sharing resources may cause information leakage [40, 37]. Some attackers may find a vulnerability from the hardware level in the computer architecture. The attacker could retrieve other users' data by observing a side-channel based on a specific hardware system. For example, by using a contention-based attack, which is one type of cache-timing side-channel attack, the attacker could monitor the usage or the trace of cache lines in the cache, and then if specific data was accessed. Furthermore, the attacker could deduce or retrieve some sensitive data which skips the protection of the virtualisation. Such a type of attack can be achieved without physical contact with the victim's machine. Therefore, an attacker can apply contention-based attacks to cloud servers and steal useful information without being noticed by either the cloud vendor or victims.

Many countermeasures [78, 26, 43, 80, 70, 49, 60, 76] have been proposed in the past decades. For example, randomisation remapping cache, which is one type of secure cache against contention-based attacks, has become the design trend. The defence strategy is passive. We will explain this type of secure cache design and provide state-of-the-art design examples in Section 2.4.2. Based on randomisation remapping cache, the cache line is remapped and scattered around in the cache by a dedicated function or a table. Some designs can provide static but overwhelming protection against contention-based attacks. They may reduce the cache performance [49] or require huge power and hardware overheads [63], which is less practical. Some other designs such as CEASER [59], and CEASER-S [60] provide flexible protection against contention-based attacks, using a re-keying function. This type of design avoids huge permanent hardware or power overheads. However, stronger protections can lead to higher performance degradation, which affects all users who share the same server. Although, CEASER-S [60], and another similar design called Scattercache [80], can still defeat an improved attack with the PPP profiling method [57], the performance degradation can become significant. Contention-based attacks have been improved, making them easier to implement and even allowing them to overcome some cache protections in recent years [58]. Hence, such attacks are likely to be a greater threat in the near future.

Therefore, in our research, we want to propose a secure cache design which can provide a better balance between cache security against contention-based attacks, cache performance, and hardware and power overheads. Such a secure cache should achieve better performance than the existing randomisation remapping cache with a re-keying function while providing equivalent or even better protection against aggressive contention-based attacks that might be developed in the future. Furthermore, we also want to mitigate the performance degradation to those users or processes that do not require protection against contention-based attacks. On a cloud server, different users can, therefore, select either high protection or high performance based on their own requirements.

1.5 Objectives

As mentioned in 1.4, many secure cache designs against contention-based attacks have been proposed to provide overwhelming protection. These caches can either cause significant performance degradation or require large hardware overheads. For example, Mirage [63] can cause a 20% increase in the cache storage overhead. Instead, we want to offer a flexible cache design that allows the user to balance between the performance and the security against contention-based attacks. To achieve these, we set two objectives in this research project.

1. Propose a randomisation remapping cache design that allows the cache's associativity to be adjustable. A privileged user can enhance the protection against a contention-based attack while sacrificing some cache performance. This cache should achieve a better performance in terms of cycles per instruction (CPI), and equal security against contention-based attacks than the CEASER-S cache [60]. The hardware overhead and power should be less than other randomisation remapping cache designs that provide fixed high protection but high hardware overheads and power costs. An example of such an existing cache design is Mirage [63], which requires about a 20% increase in the cache storage overhead and 18% increase in the total cache power.

2. Enhance the randomisation remapping cache design. The new cache design should only provide strong protection against contention-based attacks for specific users or processes. Hence, it improves the normal user's cache performance in terms of CPI, while providing the same or better security against contention-based attacks for the protected users. Unprivileged users can decide whether they want to be extra protected or not. Furthermore, the cache access latency should be reduced compared to the design which meets the first objective. The hardware overhead and power consumption should be acceptable for a modern commercial processor, such as an AMD5995wx processor [3].

1.6 Thesis Structure

Chapter 2 provides a literature review of cache architectures and cache-based timing side-channel attacks. Section 2.1 explains the conventional cache architecture and explains the essential terminologies of caches. Section 2.2 briefly discusses virtualisation and the hypervisor, which is associated with our cache design in Chapter 5. In Sections 2.3, 2.4, and 2.5, we then discuss and analyse typical cache timing side-channel attacks and their countermeasures. Section 2.6 explains the state-of-the-art method of implementing a contention-based attack. Section 2.7 introduces the simulators and tools that are used in our research. Finally, we summarise the existing secure cache designs against contention-based attacks and the gap that the existing designs have not covered in Section 2.8.

Chapter 3 introduces our novel cache protection scheme called logical associativity. We first explain the motivation in Section 3.1, and then explain logical associativity and how its three properties can influence the security against contention-based attacks on a randomisation remapping cache in Section 3.2. We introduce our security simulator in Section 3.3, which is used for evaluating the security of randomisation remapping cache against contention-based attacks. Section 3.4 discusses the indexing function selection of the randomisation remapping cache. Finally, the chapter is summerised in Section 3.5.

Chapter 4 introduces our first secure cache design against contention-based attacks. The cache is named CEASER-SH. Section 4.1 explains the motivation for proposing CEASER-SH. Section 4.2 discusses the threat model. Section 4.3 introduces the architecture and implementation of CEASER-SH. In Section 4.4, we show the security evaluation results. Then, in Section 4.5, we evaluated CEASER-SH's performance with different tag latencies using the gem5 simulator [15]. We evaluate the hardware overhead and power consumption in Section 4.6.

Chapter 5 introduces our second secure cache design, namely, the SEA cache. We first explain the motivation in Section 5.1 and discuss the significant differences and modifications compared to the CEASER-SH cache in Section 5.2. As in Chapter 4, we provide similar evaluations of the SEA cache in Sections 5.3, 5.4, and 5.5. The chapter is summarised in Section 5.6.

Finally, in Chapter 6, we conclude the thesis. We summarise the pros and cons of the CEASER-SH design and the SEA cache design in Section 6.1. We then suggest possible future works in Section 6.2.

1.7 Publications

We have published one paper to date, based on part of Chapter 3 and Chapter 4 of this thesis. We are preparing another journal paper for IEEE Transactions on Computers. This paper uses both Chapters 3 and 5 of this thesis. We are aiming to submit this paper in December 2023. The published work is shown below.

X. Liu and M. Zwolinski, "Mitigating Cache contention-based Attacks by Logical Associativity," 2022 17th Conference on Ph.D Research in Microelectronics and Electronics (PRIME), 2022, pp. 229-232, doi: 10.1109/PRIME55000.2022.9816809.

Chapter 2

Cache Architecture and Cache Timing-side channel attacks

2.1 Cache Architecture

A cache is an intermediate unit between CPU registers and the main memory. It temporarily stores some accessed data or instructions, which helps to reduce the latency caused by directly accessing the same data from the main memory [31]. This is supported by the temporal locality of reference: If a particular memory location is referenced once, then it is likely that the same location will be referenced again shortly [55].

Nowadays, a modern processor usually has different levels of caches. Typically, the cache has three levels. The L1 cache is the closest cache to the core. Usually, it is divided into an instruction cache (I-cache) and a data cache (D-cache). L2 and L3 caches are often unified, which means the data and instructions are hybrid. The L2 cache could be allocated to one or multiple cores. In terms of size, $L1 < L2 < L3$. In some processors, there is no L3 cache. Hence, the L2 is the last level cache. Most processors, including modern commercial desktop or server CPUs, have an L3 cache as the last-level cache (LLC), which connects to the main memory. Figure 2.1 shows the memory hierarchy. The higher level cache normally indicates the cache closer to the cores. For example, L1 is a higher level cache compared to the L2 cache. Each cache is divided into many blocks of the same size. These are called *Cache blocks* or *Cache lines*. Each cache line has its own address, usually segmented into three parts: *Tag*, *Index-bits* and *Offset*. Since each cache line stores many bytes, typically 64, the offset is used to distinguish between the 64 bytes within a cache line. If the entire cache line is accessed, the cache ignores the offset bit. The cache tag is used to identify if the currently stored cache line is the requested one by the processor. The identification is achieved by comparing the tag from the accessed address and the tag stored in the accessed cache lines. The *Index-bit* will be discussed below.

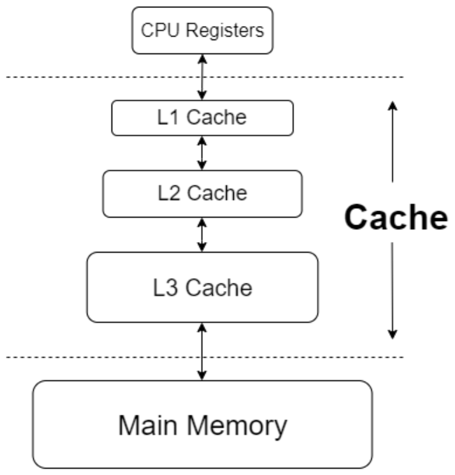


FIGURE 2.1: The memory hierarchy in a modern computer.

The cache line placement in a cache is determined by the cache line's index-bit and the cache's placement policy, such as *direct-mapping*, *set-associative* and *fully-associative* cache. If cache lines have the same index-bit, they are allocated into a specific group of positions in the cache. Such a group is named a **Cache Set**. Within this cache set, one or more positions could have the same index-bits. This is called the **Cache Way**. The number of cache ways within a set is different for different placement policies. For a direct-mapped cache, each cache set only contains one way. In other words, when two cache lines with the same index-bits need to be stored in a direct-mapped cache, the second accessed cache line must evict the first cache line because there are no other positions for it. A fully-associative cache is the opposite of a direct-mapped cache. It has only one set but has the same number of cache ways as the number of blocks in the cache. In other words, when a cache line is accessed, the cache line can be placed anywhere because all cache lines share the same cache set. The set-associative cache is between the fully-associative cache and the direct-mapped cache. In a set-associative cache, each set can have n ways ($1 < n <$ number of cache lines). A 2-way set-associative cache example is given in Figure 2.2. If two cache lines with the same index bits need to be stored in a two-way set-associative cache, both cache lines can reside in the set at the same time without eviction.

Since the cache is much smaller than the main memory, the requested cache line may not be present in the cache. This situation is called a *cache miss*. The opposite is a *cache hit*. After the cache experiences a cache miss, it will send a request to the lower level of memory and wait until the requested block is returned. This procedure can cause some delay which is called the *miss penalty*. When a cache miss occurs, the cache must stall and wait for the incoming cache block. Modern caches use *Miss Status Holding Registers* (MSHR), which temporally hold the missing cache line information to mitigate such a penalty. As a result, the cache can deal with other later cache requests while waiting for the lower-level memory response of the current cache request.

A fully-associative cache could have a lower miss rate than a direct-mapped cache because a particular cache line can be placed anywhere in the cache. However, due to searching, a fully-associative cache normally suffers higher access latency and power consumption than a direct-mapped cache. As a compromise, most modern designs use the set-associative cache.

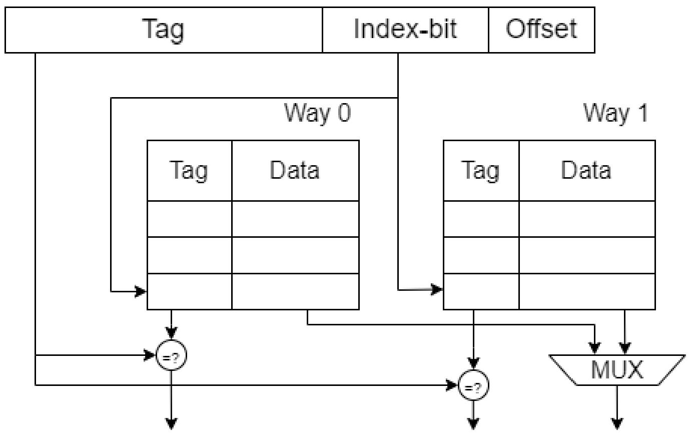


FIGURE 2.2: An example of a two-way associative cache. Both cache ways are searched in parallel.

Based on the placement policy, different cache lines could be mapped to the same cache set. If there is more than one way in the cache set, a *Replacement Policy* must be utilised to determine which way should be replaced by the upcoming cache line. Some common replacement policies are *Least Recently Used* (LRU), or *Random replacement policy*. When the LRU replacement policy is applied, the least recently used cache line within the target cache set will be replaced by the incoming cache line. For the Random replacement policy, one of the cache lines within the targeted cache set is randomly selected and replaced by the incoming cache line. There are also other replacement policies, such as the First in First Out (FIFO) replacement policy, Most Recently Used (MRU) replacement policy and the Bimodal Insertion Policy (BIP) [61].

As we have mentioned, the cache needs to identify if the stored cache line is the requested cache line by comparing the tag bits. The tag storage in the cache is typically separated from the data storage. There are two ways to access the data and tag storage. The first one is sequential access. The cache first searches and compares the tag from the tag storage. If it is a cache hit, the corresponding cache way and the cache set are then sent to the data storage for data access. In a parallel access cache, both the data and the tag are accessed together. Usually, since the tag storage is smaller than the data storage, searching in the tag storage is faster. Hence, when the data from all cache ways are read out, the cache could use the hit results from the tag to decide which cache way should be selected to return the data.

As mentioned at the beginning of this section, there are three cache levels in a modern processor. The synchronisation between different levels of memory is crucial. *Cache*

Inclusion Policy and *Cache Write Policy* were introduced to manage this. The cache Inclusion Policy decides if a cache line existing in the higher-level cache should also appear in a lower cache level. There are three types of inclusion policy: *Inclusive*, *Exclusive* and *Non-Inclusive*. An inclusive cache ensures that cache lines present in the lower-level cache are also present in the higher-level cache, whereas the exclusive cache guarantees that cache lines present in the lower-level cache do not present in the higher-level cache. In contrast, the non-inclusive cache does not impose strict limitations on the inclusion of cache lines across different cache levels. The cache Write Policy decides when other cache levels should update a cache line change if one cache line has been modified in one cache level. There are two types of write policies, one is *Write-Through*, and the other is *Write-Back*. Write-through updates all cache levels simultaneously but write-back only updates when the cache line is needed [9]. Although write-through is much simpler and cleaner than write-back, when a cache line is written multiple times in a higher-level cache, write-back could be more efficient than write-through. This is because the cache does not need to update that cache line in the lower-level caches until the cache line is accessed.

To optimise the power and access latency of the cache, an implementation technique, **multibanked cache** [31], can also be applied. A cache can be segmented into several **Cache Banks**. Each memory (cache) bank can independently handle a request. Hence, if the accessed data are stored in different memory (cache) banks, these data can be accessed simultaneously [45]. Based on the spatial locality of reference, if a memory location is referenced once, its nearby location is likely to be referenced soon after [55]. The cache sets are distributed in low-order interleaving. This means the distribution of sets depends on the last bits of the index bits [32]. For example, cache sets 0, 1, 2 and 3 are stored separately in cache banks 0, 1, 2 and 3, respectively. Cache set 4 is also located in bank 0 since the last 2 bits are 00. An example of the cache bank implementation is the L2 cache of ARM Cortex-A8 processors [5].

2.2 Virtualisation and Hypervisor

In this section, we briefly explain virtualisation and the hypervisor. The **Page** of the virtualisation technology, which is highly related to memory access, is the main focus of the section. This will be an essential factor in our work in Chapter 5.

2.2.1 Virtual Memory

When different processes run on the same machine, they may have conflicts regarding address usage. A programmer cannot know which addresses have been used by other applications on a machine. Therefore address management needs to be achieved automatically. To make this possible, **virtual memory** was applied to the computer architecture.

Before explaining virtual memory, we first define virtualisation. Virtualisation is the application of the layering principle through enforced modularity, whereby the exposed virtual resource appears identical to the underlying physical resources [21].

After applying the virtualisation, the conventional machine addresses (or the physical addresses) are remapped and are called virtual memory. Virtual memory management is achieved by the OS and hardware cooperating. From the process perspective, the OS allocates continuous spaces in the memory. These addresses are no different from the physical memory that the real hardware memory uses. From the OS perspective, these data can be placed anywhere in the memory, even if the addresses can be mapped discontinuously.

Besides the ease of memory management, virtual memory also provides a level of data protection. Since the OS controls the virtual memory mapping, it needs to know which physical address is allocated to which process. When a process tries to access data that does not belong to it, the OS refuses its data access, which is called a *trap*, and to protects the data security of the process to which that data belongs.

An essential technique used to construct the virtual memory mapping is called **Paging**. Paging requires a table to store the mapping between physical and virtual addresses. Such a table is named a page table. Each page mapping that is stored in the page table is called a page table entry. Each process has its own page table. A chunk of the virtual memory is called a page, and a chunk of the physical memory is called a frame [4]. If each page has the same size, the page table only needs to store the start address of the frame. Normally, the physical address is needed in an L3 cache access. Hence, before the cache access, the partial virtual address is used to find the corresponding page table entry and returns the physical address from the frame.

2.2.2 Hypervisor And Virtual Machine

As discussed in the previous chapter, virtualisation supports multiple processes running on the same hardware without conflicts with resources such as memory. By adding another layer of virtualisation to the system, the machine could also be treated as a set of machines, each having a complete environment including processor, memory and I/O [21]. These machines are called virtual machines (VMs). Special system software controls and manages these virtual machines, namely a hypervisor.

For security purposes, the VMs running under the hypervisor should always run with reduced privileges so that the hypervisor can monitor them. The privilege of a process determines if the process is allowed to use some security-sensitive instructions, namely privileged instructions. There are typically two privilege levels: Privileged Mode (Kernel Mode) or Unprivileged Mode (User Mode).

2.3 Cache Timing Side-channel Attack

2.3.1 Timing Side-Channel Attack

A side-channel attack is a non-invasive attack that exploits information leakage from indirect sources or channels, and it normally focuses on the implementation of the cryptographic algorithm instead of the weakness of the algorithm itself [13]. The information leakage could be of many types: Power consumption, time taken for specific operations, or electromagnetic radiation. An attacker can analyse the leaked information from these channels and accelerate the exploitation of the secret key while targeting a cryptographic function. Compared to breaking the cryptographic function by brute force, a side-channel attack provides a practical method for attackers [2, 12, 53, 81, 28, 20]. As a real-life example, if a thief wants to steal something from a house, he might need to do some preparation works before he goes into the house. The most important information he needs is to figure out when nobody is in the house. He can check the water meter frequently and see if somebody in the house is using water. Or, in the evening, he can easily observe if the lights in the house are turned on. These behaviours can also be treated as real-life side-channel attacks.

In this research, we focus on cache timing side-channel attacks. More specifically, our target attacks are cache contention-based attacks. We explain the general timing side-channel attacks first. In this attack, the attacker may measure and analyse a cryptographic system's timing. The timing of the cryptographic system varies with different inputs. By observing the timing difference, the attacker may find useful information that can be used to break the cryptographic system, such as an encryption key. Normally, attackers apply timing side-channel attacks with other types of side-channel attacks. Therefore, they can gain as much leakage information as possible. Cache timing side-channel attacks slightly differ from general timing side-channel attacks. We explain this attack in the next subsection.

2.3.2 Cache Timing Side-channel Attacks

As mentioned in Section 2.1, cache usage mitigates the latency of accessing some recently used data directly from the main memory. Therefore, compared with when a major fraction of the program resides in the cache, there is a noticeable timing difference regarding the total execution time when none or just small portions of the program are stored in the cache. In other words, the timing difference caused by cache hits and cache misses while running a program can be observed and analysed by the attacker as a side channel. Figure 2.3 shows the timing difference between a cache miss and a cache hit. Hence, if the attackers targets the cryptographic process, they can learn useful information and accelerate the encryption key exploration by statistical analysis [1]. This type of attack is called a cache timing side-channel attack.

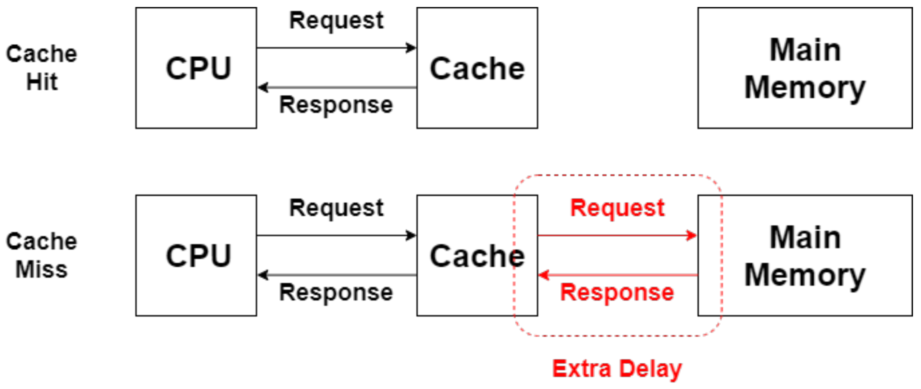


FIGURE 2.3: A timing comparison between a cache hit and a cache miss.

In the rest of this section, some typical cache timing side-channel attacks and their categories are discussed. Then, some practical attack examples are introduced and evaluated. These attack examples leverage the cache timing to learn if a victim cache line is in the cache and to recover the secret key of a cipher [12, 53, 88] or to help the attacker to build a covert channel for transmitting other security-sensitive data [37, 40].

2.3.3 Types of Cache Timing Side-Channel Attacks

The cache timing side-channel attacks were split into two categories by what to measure during the attack. One type is called a **Time-Driven attack**, and the other type is called an **Access-Driven Attack** [54].

For a Time-Driven attack, the attacker compares the victim process's total run time to deduce if the victim has used the targeted cache lines. There are two major types of Time-Driven attacks. They are *Evict+Time* [53] and *Cache Collision Attack* [18].

This type of attack usually requires many samples to determine the average run time of the victim process. The attacker changes the cache state before triggering the victim process, reflecting a cache hit or miss. Due to that cache hit or miss, running the victim process differs from the average time.

Table 2.1 shows the attack procedure of *Evict+Time*. There are two phases in this attack: the preparation phase and the attack phase. The preparation phase runs the victim process twice. The first run ensures all the victim process data are loaded into the cache, and the second run counts the process run time when the process was accessed.

Preparation phase:	
1.	The attacker triggers the victim process.
2.	The attacker triggers and counts the time taken for the victim process.
Attack phase:	
1.	The attacker triggers the victim process.
2.	*Evict*: The attacker uses its data to fill in a specific cache set.
3.	*Time*: Triggering the victim process and time again.

TABLE 2.1: The procedure of Evict+Time attack.

Since the process was accessed, if the self-eviction is ignored, the attacker can guarantee that most cache lines of the process reside in the cache. The attacker can record the timing of the second access for comparison in the later attack phase. If the attacker needs to attack multiple times and the process does not change, the preparation phase does not need to be repeated. In the attack phase, the attacker triggers or waits for the cache access of the victim process, the same as the first step of the preparation phase. Then in the second step, the attacker loads its data into the cache and fills in a cache set or a few specific cache sets. If the victim process used this cache set in step one, the victim's data in such a cache set would be evicted by the attacker's data. Finally, the attacker re-triggers the victim process and counts the run time. If the time taken in the third step of the attack phase is longer than the second step of the preparation phase, the attacker can be sure that the cache misses are caused by the data accesses in the second step of the attack. Hence, the attacker can infer which cache set was accessed by the victim's process data. As an example, the victim runs an AES encryption process with the encryption key K. In the preparation phase, the attacker counts the AES process's total run time as $T1$ clock cycles in the second step when the encryption key is K. During the attack phase, after the attacker ensures the AES process was just executed with the encryption key K, they fill in a cache set with their own data. Then they trigger the victim's AES process and time the total time of execution. For example, if the total run time now becomes $T2$ clock cycles, where $T2 > T1$, the attacker can be sure that the higher latency is due to the extra miss penalties. Such penalties must be because the attacker's data evicted one or more cache lines from that cache set. This timing difference can tell the attacker those cache lines were accessed while the victim was running their AES process with the encryption key K. If those cache lines from the victim process are related to the encryption key, in other words, they are part of the AES S-box, the attacker could leverage this and accelerate the exploration of the victim's AES encryption key. This encryption key exploration process will be discussed further in Section 2.3.4.

Besides *Evict+Time*, there is another Time-Driven attack named *Cache Collision Attack* [18]. The main idea of this attack is to measure the timing difference caused by the reuse of cache lines within the victim process itself. This may be used to refer to security-sensitive information. For example, in an AES attack, an attacker can find the reuse of cache lines and estimate the relations between each portion of the key by applying a cache collision attack [18]. Nevertheless, the cache collision attack is heavily limited by the victim process and less practical for an attack.

Unlike Time-Driven attacks, Access-Driven attacks do not rely on measuring the victim process's total execution time. Instead, the attacker uses their own process to determine if the victim has accessed a cache set. The attacker first occupies some cache lines and then triggers the victim process. After the victim process is executed, the attacker re-accesses those cache lines occupied in the first step. A typical example of this attack is *Prime+Probe* [53], which is introduced below.

Attack phase:
1. *Prime*: The attacker loads cache lines and occupies part of or the entire cache.
2. The victim process is triggered.
3. *Probe*: The attacker re-accesses those cache lines and measures the time set by set.

Table 2.2: The procedure of Prime+Probe attack.

Unlike *Evict+Time* which runs in the order of victim-attacker-victim during the attack phase, *Prime+Probe* runs as attacker-victim-attacker. This gives the attacker more controllability. Also, in an Access-Driven attack, an attacker only needs to compare the time taken to access each set to distinguish which set has been accessed by the victim. The set which takes a longer time must be the set accessed by the victim. The attack precedure is shown in Table 2.2. In the first step of the attack phase, the attacker primes the cache with their own data. Priming can be achieved by accessing an array in the attacker's process. The victim process is then triggered. In the last step, the attacker re-accesses that array. If a cache set were used during the victim encryption process, the cache line primed in the first step would be evicted by the victim, leading to a cache miss. After checking all cache sets, the attacker can retrieve the cache state changes and deduce which cache set has been accessed by the victim. *Prime+Probe* provides a higher fidelity method of exploring if a cache set has been accessed by the victim compared with *Evict+Time*, which observes the total run time changes. Also, in one round, it can observe the state of more than one cache set, which improves the attack efficiency. Moreover, *Prime+Probe* requires one less time for the victim to access each round of the attack compared to *Evict+Time*. From the attacker's perspective, accessing their own data should be much simpler than triggering the victim's process. Hence, the *Prime+Probe* attack is more practical than the *Evict+Time* attack. Like the example of *Evict+Time*, the attacker can utilise *Prime+Probe* to retrieve victim process information too.

Another Access-Driven attack is named *Flush+Reload* [85]. This attack can be treated as a special variant of *Prime+Probe*, which requires a shared memory between the victim and attacker processes. An instruction called *clflush*, which normally is used to invalidate cache lines to maintain cache coherence, is used to evict the specific cache line rather than the entire cache set. Therefore, it is even more precise. Like *Prime+Probe*, after evicting cache lines, it waits until the victim run is finished and then accesses those cache lines and examines if the victim process accessed those shared cache lines. The weakness of this attack is that it relies on the shared memory of security-sensitive data. Also, another variant, the Flush+Flush attack [27], was proposed based on the Flush+Reload attack. The classification of time-driven and access-driven only distinguishes how the attacker measures if a cache line is used by the victim but does not cover the root causes of these attacks. A new classification [42] includes two types of cache timing side-channel attacks: *contention-based attacks* and *reuse-based attacks*. For contention-based (or conflict-based) attacks, some data from the attacker is mapped to the same cache set where the targeted victim cache lines are allocated. It is worth noting that the contention cache lines are different cache lines that have different addresses. When the contention occurs, either the attacker or the victim can evict the other's cache line. The attacker can abuse the deterministic mapping and eviction to prepare the desired cache state and examine the cache residency of the victim's targeted cache lines. Finally, the sensitive data of the victim is retrieved.

	Contention-based Attacks (Find Cache Miss)	Reuse-based Attacks (Find Cache Hit)
Access-Driven (Measure attacker access time)	**Prime+Probe** 1. Attacker Process Evicts 2. Run Victim Process 3. Attacker Process Checks Access	**Flush+Reload** 1. Attacker *clflush* shared memory 2. Run Victim Process 3. Attacker Process Checks Access
Timing-Driven (Measure victim total run time)	**Evict+Time** 1. Run Victim Process 2. Attacker Process Evicts 3. Re-run Victim Process	**Cache Collision Attack** 1. Run Victim Process 2. Re-run Victim Process

Table 2.3: A summary of timing-cache side-channel attacks.

Unlike the contention-based attack, the reuse-based attack does not rely on the conflict mapping of two different processes. Instead, it observes if the same data, shared between the victim and attacker or within the victim itself, is accessed more than once.

A contention-based attack constantly measures the cache miss(es) of addresses and retrieves useful information from it. Conversely, a reuse-based attack expects the same cache line(s) is (are) accessed more than once. In other words, it measures cache hit(s). A summary of typical cache timing side-channel attacks is shown in Table 2.3.

2.3.4 Examples of Cache Timing Side-Channel Attacks

In Section 2.3.3, 4 typical cache timing side-channel attacks and their classifications have been introduced. In this subsection, some real attack examples are explained in detail, which also explains why cache timing side-channel attacks are dangerous.

As discussed in 2.3.1, most side-channel attacks focus on attacking cryptographic systems such as an encryption cipher. In the first example, we provide an example of how cache timing side-channel attacks can be applied to break AES [12, 53]. The second example is attacking RSA [44], which is an encryption function not based on lookup tables. The last example is using cache side-channel attacks to build a covert channel. An attacker can use this covert channel to transmit data between different processes. This is a crucial step in the well-known attacks: Meltdown [40] and Spectre [37].

2.3.4.1 Cache Timing Side-Channel Attacks on AES

A typical example of a cache timing side-channel attack is using either *Evict+Time* or *Prime+Probe* to attack AES [53]. Here, we explain how an AES First-Round Attack can be achieved.

Four pre-calculated tables are needed during the AES encryption process, namely T_0, T_1, T_2, and T_3. Each table has 256 table entries, and each table entry has 4 bytes. Since the widely used cache line size is 64 bytes, each cache line will be filled with $64 \div 4 = 16$ table entries [25]. In other words, T_0, T_1, T_2, and T_3 are each segmented into $256 \div 16 = 16$ cache lines, a total of 64 cache lines. It is worth noting that only the cache lines that store the required table entries will be loaded into the cache during the encryption process. In the entire AES encryption procedure, the calculation needs to be done in 10 rounds. In the first 9 rounds, each T_0, T_1, T_2, and T_3 will be utilised 4 times per round, 36 times in total. Which table entry is needed during the calculation depends on the table indices values. These table indices in the first round of encryption are calculated as $Y_i = K_i \oplus P_i$. K_i and P_i are parts of truncated key and plaintext, respectively. For AES-128, the length of the key K and the plaintext P is 128. Since each K_i and P_i has an 8-bit length, there are 16 truncated parts of the key and the plaintext.

Since there are 16 table entries stored in the same cache line, the attacker can only know the first 4 bits of Y_i, which is shown as $\langle Y_i \rangle$, by observing the cache state changes and learning which cache lines that store the pre-calculated tables are used. Then, if the plaintext is known, the first 4 bits of the K_i can be interpreted. After repeating this for all 16 truncated parts of the key, half of the key can be recovered.

The major step in this attack is observing and analysing the cache state changes. Two cache timing side-channel attacks were used: Evict+Time, and Prime+Probe. Using one of these attacks, the attacker can examine if any table entries were accessed during the encryption. Although the attacker cannot distinguish in which round the table entry was accessed, applying many trials with different plaintexts allows the attacker to find the probability differences and recognise if table entries in a cache line were accessed in the first round of encryption.

Also, it is worth noting that the attack that applies Prime+Probe is much more efficient than applying Evict+Time. This is because Prime+Probe can target multiple cache sets in one round, and can provide high-fidelity results since it relies more on the attacker's cache lines.

Overall, during the first round of the AES encryption process, the pre-calculated tables are loaded based on the XOR of the plaintext and key. Although the table might be accessed in later rounds, fixed XOR results of the partial plaintext and the partial key can always lead to accessing the same cache line in the first round. Since this cache line is always accessed (with 100% probability of access) instead of probabilistically like other cache lines, an attacker can distinguish the targeted cache line from many trials. Finally, the partial table indices of the pre-calculated table stored in that cache line and known plaintext can help the attacker recover the partial key.

2.3.4.2 Cache Timing Side-Channel Attacks on RSA

Another cache timing side-attack example is applying such an attack to RSA [44], which is another widely used encryption algorithm. RSA has two different keys, private and public. The private key is used to decrypt the encrypted message. The public key is used for encryption. Since the generation of the keys is not related to this section, we do not discuss this in detail.

For the RSA decryption process, the calculation is done by applying the equation $m = c^d$ $mod\ n$. m is the plaintext message, and c is the ciphertext. Users must have two numbers, d and n, as their private keys during the calculation. By calculating the equation, the user can recover the plaintext. However, the key size is normally very large. Some common examples of key lengths are 2048 and 4096 bits. The power calculation, c^d, will require huge computational power. To optimise this, some implementations of RSA apply a square-and-multiply algorithm. In this algorithm, some intermediate numbers of squaring are calculated to reduce the rounds of power calculation instead of computing m directly based on c.

There are two major calculations: Squaring and Mod, Multiplying and Mod. The RSA implementation takes $c_{current} = 1$ as its start. It first picks the first bit of d. If d is 1, both calculations should be performed, namely, $c_{next} = c_{current}^2\ mod\ n$ followed by $c_{next} = (c_{current} \times d)\ mod\ n$. c_{next} in the current operation will become $c_{current}$ in the next operation. If d is 0, only Squaring-and-Mod is needed. After finishing the operations based on the current bit of d, the next bit of d is used to do the same calculations. By doing such a calculation until the last bit of d is calculated, the user can retrieve the plaintext.

Although such an implementation dramatically improves the performance and efficiency of using computational resources, it can also be leveraged with the cache timing side-channel attack by an attacker to retrieve the victim's private key. Unlike AES, RSA does not require a pre-calculated lookup table. However, the intermediate values must be stored temporarily for the next round of calculations. Although the attacker does not know where the data could be stored in the cache, an attack on RSA [44] was proposed to figure out the mapping and read out most of the bits of the victim's RSA private key.

In this attack, the attacker uses the Prime+Probe attack to occupy a candidate cache set. Hence, by repeatedly applying the Prime+Probe attack, the attacker tracks the candidate cache set when the RSA encryption process is running. As we have explained, RSA only has two different cases: Either it needs to do Squaring and Mod when d is 0, or both Squaring and Mod, and, Multiplying and Mod when d is 1. These two cases have different timing delays. After accessing the intermediate data from the cache, $d = 0$ requires 2-3 times more delay than $d = 1$. Since the same cache set needs to be accessed many times, when the attacker finds the trace of a cache set which is always accessed with a regular pattern in terms of timing, they can deduce that this candidate cache set stores the intermediate values of RSA. If not, the attacker can then discover the trace of another cache set. Furthermore, by observing the trace of this cache set, they can read out most bits of d from the timing delays between two adjacent cache accesses. From their real attack on servers and desktops, the attacker recovered a large portion of the RSA private key (which weakens the security of the RSA encryption implementation.) .

Overall, an attacker can apply cache timing side-channel attacks to prepare the expected cache status before each round of the RSA calculation starts. Due to the specific access pattern of the square-and-multiply algorithm, the attacker can simply recognise that the cache set stores the intermediate data of RSA. This makes such an attack more practical than the AES attack, had mentioned in Section 2.3.4.1. Moreover, from the trace of the cache set, the attacker can read most bits of the private key. By knowing most bits of the private key, the attacker can use a brute-force attack to find out the rest of the private key.

2.3.4.3 Covert Channel Attack

Another example of using cache timing side-channel attacks to retrieve sensitive data is building a covert channel, such as the C5 covert channel [46] . There are two well-known attacks: *Spectre* [37] and *Meltdown* [40]. These attacks leverage the design flaws of other parts of the processor's hardware to retrieve sensitive security data that the attacker ideally should never access. We start the explanation with the Meltdown attack [40].

The main idea of the Meltdown attack [40] is leveraging the out-of-order (OOO) execution and the speculation execution of the modern processors. In more detail, as we have mentioned in Secion 2.2.2, a non-privileged user cannot directly access the privileged user's data. When this happens, the processor can raise an exception to the OS and may terminate the program causing such an exception. Hence, the rest of the program is not executed. However, this may not be true in an OOO processor. In such a processor, a sequence of code within the same program is split into a few micro-operations. These micro-operations are executed in parallel if they are independent of each other. When these micro-operations are finished in execution, their results are gathered in order. If an exception is raised, all the results are invalidated by flushing the pipeline. Nevertheless, when two micro-operations are executed simultaneously and the second micro-operation is later than the first micro-operation in terms of the code order, the second micro-operation may be executed before the first micro-operation causes an exception. Even if the second micro-operation is accessing privileged data, the processor does not recognise it until either of the micro-operations throws an exception. Unfortunately, although the processor has flushed the pipeline and does not leave any visible architectural effect for the attacker to reveal security-sensitive data, there could be microarchitectural effects. For example, if the second micro-operation accesses a cache line, this cache line can be stored in the cache. More importantly, this is not affected by the pipeline flushing. Using the method, an attacker can access an element of an array indexed by privileged data that he wants to discover. Later, by using any timing cache side-channel attacks we have introduced in Section 2.3.3, such as Flush+Reload, the attacker can read out which cache line that stores the array in the cache [40]. Hence, they can deduce the privileged secret data.

The other attack example is Spectre [37]. Similar to Meltdown , this attack also leverages speculative execution. Here, an attacker misleads the branch predictor in the processor and utilises speculative execution to access the victim's memory space that the attacker should not be able to touch. Specifically, when the data that is used for determining the conditional branch is not stored in the cache, to avoid a long time waiting for loading data from the main memory, the processor uses the previous branching results to execute the rest of the code. If the attacker trains the branch predictor to take branches many times, it then executes the code within the branch even if the processor cannot ensure if this needs to be executed. Like the meltdown attack, attackers can access an array indexed by the secret data from the victim process. Later, when the processor loads the data from the main memory and notices it should not take the branch, the corresponding cache line indexed by the secret data has been loaded into the cache. In other words, the cache state has been modified. Using timing side-channel attacks, the attacker can finally deduce the secret data by examining which element within the array is in the cache. Unlike Meltdown, Spectre does not rely on any exceptions.

It is worth noting that the root of these attacks' vulnerability is branch prediction, speculative execution and out-of-order execution, not the cache. Other microarchitectural states can also construct a covert channel, such as the ALU [40]. In other words, cache timing side-channel attacks in these attacks are not the principal offender but an accessory.

2.3.4.4 Summary

In this subsection, we have introduced a few examples of the cache timing side-channel attacks were used to retrieve other users' partial keys or data, including privileged users. In the AES and the RSA examples, the attacker leverages the implementation of the encryption process and the cache timing to examine the use of pre-calculated lookup tables that are highly related to the encryption key or the cache set stores the intermediate number. The attacker can deduce the other users' encryption keys based on the usage of these cache lines.

In the Meltdown and Spectre attacks, the attacker leverages the vulnerabilities from other modules of the processors, such as speculative execution, OOO execution and branch prediction. Although directly accessing the victim's data can cause the exception, the attacker could indirectly access the victim's data and observe the microarchitectural state changes. The attacker can read the trace of the cache lines indexed by the victim's data. Hence, by checking the residency of those cache lines using cache timing side-channel attacks, the attacker can finally "access" the victim's data.

We found that no matter what the attack target is in these cache timing side-channel attacks, the cache's vulnerability is never changed. The security-sensitive data can be reflected in the cache state changes, whether or not the specific cache lines are stored in the cache. Hence, the attacker has two different methods to observe the cache state changes in the relative level of the cache. The first way is to examine if the shared cache lines between the victim's process and the attacker's process are used. These are known as Re-use Based attacks. The second way is to learn the access trace of specific cache sets. The essential condition for the attacker to achieve this is to find a group of cache lines within their process, and these cache lines should contend with the victim target cache line. In other words, the target cache line can evict this group of cache lines and vice versa. As we explained, this attack is named a contention-based attack.

There is no known real-world attack based on the contention-based attack. This may be due to the effort of a lot of research to discover and fix the hardware vulnerabilities caused by the contention-based attack before the real attempts have been made. The contention-based attack could still be dangerous if it is improved.

As a result, if we can eliminate or mitigate the root of the attack method, the corresponding attack could be eliminated or become impractical, no matter what the attack target is.

2.4 Countermeasures of Contention-Based Attacks

In the early years, the cache timing side-channel attacks were only used to attack encryption ciphers such as AES. Some simple countermeasures were proposed, such as disabling the cache [53]. However, these methods either dramatically sacrifice the performance of using caches or increase the implementation complexity of the encryption ciphers, making the solutions impractical.

As discussed in Section 2.3.4, many other attacks based on cache timing side-channel attacks were proposed during the last two decades. Finding solutions for the implementation of each specific software does not solve the problem. The only feasible solution is fixing or mitigating the vulnerability caused by the cache itself. Since contention-based attacks and re-use based attacks have distinct root causes, the countermeasures that deal with different attacks are designed separately.

Many mitigations and countermeasures were proposed at both hardware and software levels. Different techniques were applied, such as cache partitioning [78, 26], cache remapping [78, 43, 80, 70, 49, 59, 60, 76], and other techniques [67, 82, 36, 68, 84]. They all try to balance the cache's security, performance, and other overheads. The countermeasures can be divided into two categories: passive strategy and active strategy. The active strategy detects malicious behaviour and then activates its defence method. In detail, an active strategy requires the detection of cache side-channel attacks. For example, the cache can be protected with a dedicated detector against contention-based attacks. This detector tracks all accesses of the cache. When the detector realises that there are many cache accesses to a specific cache set, it can deduce that these could be malicious accesses. Hence, the cache defence can fight back in many ways, such as limiting cache access from suspicious processes or terminating the process.

On the other hand, a passive strategy always provides the defence no matter if malicious behaviours or users are found. For example, the cache is frequently remapped after a period of time. An active strategy always requires accurate detection of contention-based attacks. If the attacker finds a way to trick and skip the detector, the countermeasure can be treated as disabled. The detection also needs dedicated hardware or computational power when it is achieved in software. This may lead to a waste of resources. On the other hand, a passive strategy treats all accesses equally. However, this may lead to performance loss since not all users or processes require protection against contention-based attacks.

As mentioned in Section 2.3.3, contention-based attacks and reuse-based attacks have different root causes. Therefore, there are different countermeasures to these two types of attacks. In this section, we specifically explain some countermeasures with different techniques for contention-based attacks. These countermeasures may have different focuses in terms of balancing different factors. We mainly focus on hardware solutions for mitigating Contention-Based attacks.

2.4.1 Cache Partitioning

Cache partitioning [78] is a strategy that physically separates the victim's process data and the attacker's data. The physical isolation guarantees that the victim's data will not contend with other unsecured processes. In other words, only data from selected processes or only the victim process itself can share the cache ways in the specific cache set. As a result, the security-sensitive victim cache line cannot be evicted by the attacker's process at all, which leads to the failure of a contention-based attack. The major disadvantage of this strategy is that it usually hurts performance since other processes must be run with lower cache associativity. This type of countermeasure can be seen as a passive defence strategy.

2.4.1.1 PL Cache

The partition-Locked Cache (PLcache) [78] implemented dynamic cache partitioning, which uses one extra locked bit, L, to indicate such a cache line is locked. An extra ID is attached to each cache line to indicate which process this cache line belongs to. The L bit and ID are also added for the page entry and segmentation table. Hence when loading the cache line into the cache, the PL cache knows the security information of the upcoming cache line. Since the OS controls the paging and segmentation, it also controls the dynamic partitioning of the PL cache. Unlike static cache partitioning, which statically allocates cache ways to each process, the PL cache provides a finer granularity. When a new cache line **N** needs to replace an old cache line **O**, the PL cache must check the L bit. If both cache lines **N** and **O** are not locked, or both cache lines are locked and belong to the same process, they can be treated as normal cache access. However, there are different situations when **N** has a different L bit and ID. When the cache line **N** needs to be locked, it can replace any unlocked or locked cache lines that belong to the same process. This guarantees that no locked cache lines from the different processes can interfere with each other. When the cache line **N** does not need to be locked and **O** is a locked line which has been chosen as the replacement victim, the storage of **N** bypasses this level of the cache. This design improves cache usage. Security-sensitive processes can still use most of the cache ways in a cache set. They can only use the unlocked cache ways for those processes that are not security sensitive. Compared to static partitioning, the performance degradation is less since more cache ways can still be shared between different processes. Although this method distinguishes between secured and non-secured processes, it does not detect if the malicious behaviour is performed in the cache. Hence, this design is still a passive defence. However, the PL cache only lockes the security-sensitive cache lines after all these lines are loaded into the cache, the pre-calculated lookup tables are loaded gradually during the AES encryption process. Therefore, after applying Prime+Probe to the PL cache, the cache lines accessed by the victim (locked cache lines) can still be exposed [38].

2.4.1.2 NoMo Cache

The NoMo cache [26] is another partition-based cache design. It focuses on attacks when both victim and attacker processes share the L1 cache on a simultaneous multithreading (SMT) processor. The NoMo cache design's main idea is to allocate a few cache ways as unique ways to each process running on the same core. The rest of the designated ways will be shared between processes. This design prevents a process from monopolising the entire cache set, which means one process can never use its cache lines to prime a whole cache set. As we have explained in Section 2.3.3, the attacker can only figure out the usage of the victim cache line when they can use a group of cache lines to evict or occupy the potential positions where the victim cache line might be placed. During a Contention Based attack on the NoMo cache, the attacker can only track the victim cache line usage when those cache lines are placed in the shared cache ways. Hence, increasing the minimal guaranteed cache ways can enhance the security against contention-based attacks on the NoMo cache. As a result, only partial information about the victim's cache line access is leaked. From the experimental results of the NoMo cache, the leakage is dramatically reduced compared to the conventional cache designs. Nevertheless, the cache partitioning strategy in NoMo still limits the maximum cache way of each process, which still sacrifices the performance. Especially, the L1 cache normally has relatively low associativity. When more processes run on the same processor, the available cache associativity on the L1 cache to each process becomes lower, leading to a high miss rate.

2.4.2 Cache Remapping

Another type of countermeasure is randomised cache remapping [78, 43, 59, 33, 87]. This is also the main focus of the research in this thesis. The cache remapping strategy is also a passive cache defence. The main principle of this defence is obfuscating the cache line mappings in a cache, either using a table to store the cache remapping or a dedicated function to recalculate the index bits of each cache line. Since the remapping only occurs in the cache, the mapping difference is not visible outside the cache. Hence, the distribution of the cache lines becomes much harder to predict based on the cache line address. The benefit of applying cache remapping is that the remapping depends on the hardware of the cache itself since the attacker can never manipulate the cache mapping. Although some enhancements of randomisation remapping cache need the hypervisor or kernel to be involved, the essential remapping is achieved at the hardware level.

Based on the attack examples we have explained in 2.3.4, we notice that finding a group of cache lines that can evict the target cache line, namely the eviction set, is crucial to contention-based attacks. By applying the cache remapping, the attacker cannot find the eviction set easily, which increases the attack difficulty. However, while the cache defences against contention-based attacks are improved, many advanced methods of finding eviction sets were also proposed [44, 58, 57]. This leads to a race between countermeasures and attacks. In this subsection, we focus on explaining how the remapping caches were proposed and enhanced. Section 2.6 will explain how the contention-based attacks (or the method of finding the eviction set) were developed based on the new countermeasures.

2.4.2.1 Newcache

Newcache [43] is a countermeasure based on the RPcache design [78]. Unlike the RPcache, which uses a set-associative cache as the base design, Newcache applies a direct-mapped cache. The address is randomly remapped by adding a table with the address to cache set mapping. This table is fused with the address decoder within Newcache. Hence, an incoming cache line can be placed anywhere in the cache. Since Newcache uses a direct-mapped cache, there is only one cache way in each cache set. The remapping of the cache can be seen as a cache-line-level remapping. The protection bit (P bit) and the Trust Domain Identifier (TDID) are added as part of the index bits. Hence, these bits are checked during cache access. The P bit indicates if the cache line is protected, and the TDID is used to distinguish the cache lines between different trust domains. If the index bits are not matched in the comparison during cache access, this indicates there is no valid mapping in the cache. The cache then randomly selects a cache line as the replacement victim. Hence a new mapping needs to be created and stored in the table. Otherwise, if only the tag comparison mismatches, the Newcache ensures the mapping is valid, but there is a tag miss. In such cases, the cache miss can be handled as normal. Newcache can fit with the L1 cache, which defeats the contention-based attacks and does not drop the cache performance. However, such a design may not be suitable for L2 and L3 caches since both the control logic and the storage overhead are proportionally increased with the cache size.

2.4.2.2 CEASER Cache

CEASE cache and CEASER cache [59] are the prototypes of all randomisation remapping caches proposed later, including our cache designs. Both caches utilise an encryption cipher to encrypt the tag and the index bits of the original address of the accessed cache lines. The ciphertext, which is the encrypted address of the cache line, is then split into two parts: encrypted tag and encrypted index bits. The encrypted tag is stored in the cache the same as in the conventional cache. During cache access, the encrypted tag is compared with the tag stored in the cache. Hence, the cache returns either a cache hit or a cache miss. The encrypted index bits determine which cache set should be selected to store the upcoming cache line. The reason for not using the original tag for storage and comparison is straightforward. The plaintext and the encrypted ciphertext with a fixed key are always one-to-one mappings. Hence, using the original or encrypted address, the cache can find only one specific cache line. This may not be true if the index bits and original tag bits are from different types of addresses. Since the index bits in CEASE and CEASER cache must be the encrypted index bits, the easiest design is storing and comparing the encrypted tag. When a cache line needs to be evicted from the CEASE or CEASER cache, both caches need to perform the decryption based on the cache set, encrypted tag and the corresponding encryption key. The major impact of these cache designs on the cache performance is that the encryption process requires extra clock cycles. Therefore, finding a secure encryption cipher which needs less computation time is fundamental to such cache designs. To achieve this, an encryption cipher called Low-Latency Block-Cipher (LLBC) was proposed for the CEASER cache. LLBC requires one or two clock cycles to encrypt or decrypt. Therefore the latency due to the cipher is very low and almost negligible for the L3 cache. In terms of security, both CEASE and CEASER remap the cache based on the LLBC. The key of the LLBC can be generated from a hardware-based pseudo-random number generator and stored temporarily in the dedicated storage. Since the encryption and decryption are entirely achieved at the hardware level, none of the users know how the cache lines are distributed in the cache. Nevertheless, for the CEASE cache, the attacker can still construct an eviction set of the target cache line within a few seconds [44]. (This proves that the CEASE cache, which has static randomisation remapping, can only protect the cache from contention-based attacks for a short period.)

An enhanced protection defence is added to the CEASE cache to improve its security against contention-based attacks. Since LLBC achieves the indexing in the cache, when the LLBC applies the distinct key, the same cache line can be mapped to a completely different cache set. In other words, the cache mapping becomes different. Hence, they improve the CEASE cache, which allows the encryption cipher's encryption key to be changed periodically. The new design is named the CEASER cache [59]. To achieve the remapping, the CEASER cache gradually applies re-keying, which means the encryption key is updated. The CEASER cache requires more hardware to support this function.

Two registers are used to store the encryption keys. One is the current key, and the other is the next key. A counter is utilised to indicate when this re-keying should happen. Rather than counting the clock cycles, it counts the number of cache accesses. When the counter reaches a threshold value, the counter is reset, and one cache set is selected to perform the remapping based on the next key. We call such a threshold value the re-keying period. After the remapping of the set is finished, a register called the Set-Relocation Pointer (SPtr) stores the next following cache set number, indicating that the cache set is the next remapping target. After another re-keying period, the next cache set is remapped in the same way. The entire cache keeps being remapped until the last cache set is reached. Hence, the next key becomes the current key and is overwritten in the current key registers. The next key register updates its value with a new key generated from the random number generator. When the CEASER cache is accessed, the address of the cache line is encrypted by both the current and the next keys. For performance, CEASER applies two LLBCs to calculate the encrypted address in parallel. After the encrypted address is obtained, the encrypted index bits based on the current key are selected and compared with the values in SPtr. If the encrypted index bits are larger or equal to SPtr, this means that the cache set is still using the current key. Then, the encrypted address based on the current key is used, and vice versa. Such an implementation ensures the entire CEASER cache is remapped after a fixed number of cache access. Since their designs do not use a table to store the mapping but calculate on the fly, the hardware overhead does not increase linearly with the size of the cache. Hence, applying CEASE or CEASER as the L2 or L3 cache is more practical than the Newcache [43].

However, under an improved contention-based attack [60], which significantly reduces the time of constructing an eviction set, implementing a contention-based attack on either CEASE or CEASER cache becomes practical.

Also, the LLBC in the CEASER and CEASE cache was found to be vulnerable due to a serious flaw in its encryption algorithm [16]. Based on the vulnerability of the LLBC itself, an attacker might be able to ignore the re-keying function. The encryption cipher should be replaced to ensure the CEASER cache can still defeat Contention-Based attacks. PRINCE [19], another low-latency block cipher, was suggested as the encryption function in CEASER. PRINCE was verified that it does not suffer the same vulnerability as LLBC. In terms of the performance and overhead, a 7-round PRINCE has 980ps latency and requires 3737 cells. Under a 3GHz frequency, PRINCE requires 3 clock cycles to compute the ciphertext. For an L3 cache which may need 40 clock cycles for a cache access, this increase is acceptable.

2.4.2.3 CEASER-S Cache

Although CEASER provides decent protection, a few methods to accelerate the eviction set's construction were proposed [60], which will be discussed further in Section 2.6.1. These methods dramatically reduce the time taken to find an eviction set. Hence implementing a Contention-Based attack on CEASER becomes practical. As a result, the CEASER cache is pushed to decrease its re-keying period to maintain security against such attacks. Nevertheless, as we mentioned above, this can lead to performance degradation of the cache. To further enhance the CEASER cache, it is combined with a skewed associative cache [66] to prevent contention-based attacks [60]. An example of a skewed cache is given in Figure 2.4. In this skewed cache, there are two skewed cache ways. Unlike the conventional cache, each cache way is driven by different index bits. As a result, a cache line can be mapped to different cache sets when placed in different cache ways. Furthermore, if two cache lines are mapped to the same cache set in one of the cache ways, it is unlikely they can be mapped to the same cache set in another cache way. To achieve the skewed cache, the cache separates cache ways into different partitions. The cache deploys an identical function and distinct encryption keys for each cache partition. This cache design is called CEASER-S cache [60]. The encryption key in each partition is unique so that only the cache ways in the same partition share the same cache mapping. A 4-partition CEASER-S cache is shown in Figure 2.5.

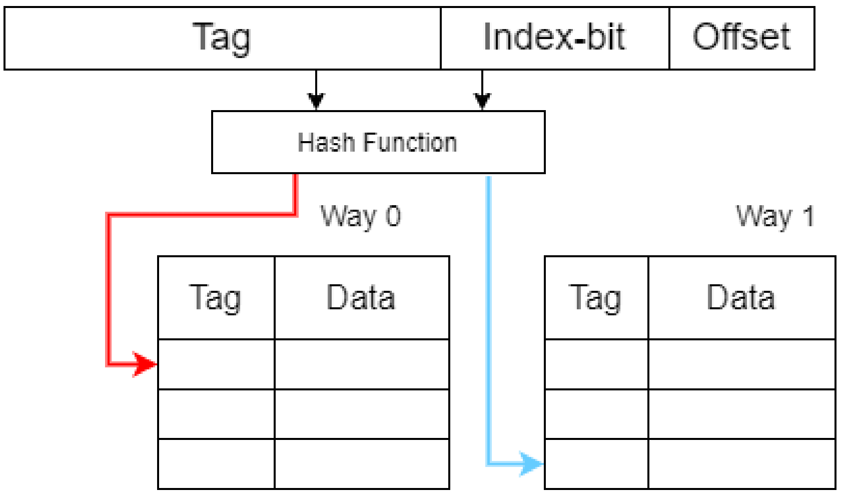


FIGURE 2.4: A two-way skewed associative cache example.

It is worth noting that the term "Partition" in the randomisation remapping cache is entirely distinct from cache partitioning mentioned in 2.4.1. Different from cache partitioning, the partitions in the CEASER-S cache only indicate they have different cache mappings based on distinct encryption keys, but any of the cache lines can still be placed in any of the partitions. Hence, the partition in the CEASER-S cache does not decrease the efficiency of using the cache. This cache design dramatically increases the complexity of the remapping. If the attacker is unaware of the CEASER-S cache architecture, they may build an eviction set as before and use it in a Contention-Based

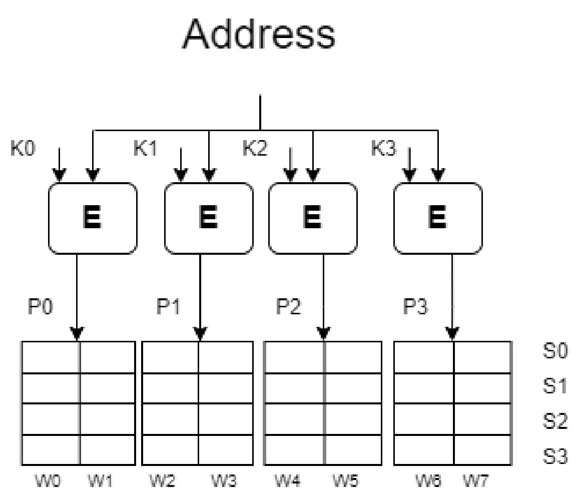


FIGURE 2.5: A 4-partition CEASER-S cache. E indicates the block cipher, and each partition has its partition.

attack. However, the member of the eviction set is highly unlikely to conflict with the target cache line. Thus, the attacker may not be able to evict the target cache line from all possible placements. This can finally lead to a failure of the attack. As a result, this cache design enhances the remapping to achieve a finer granularity. CEASER-S dramatically increases the complexity of building an eviction set for Contention-Based attacks.

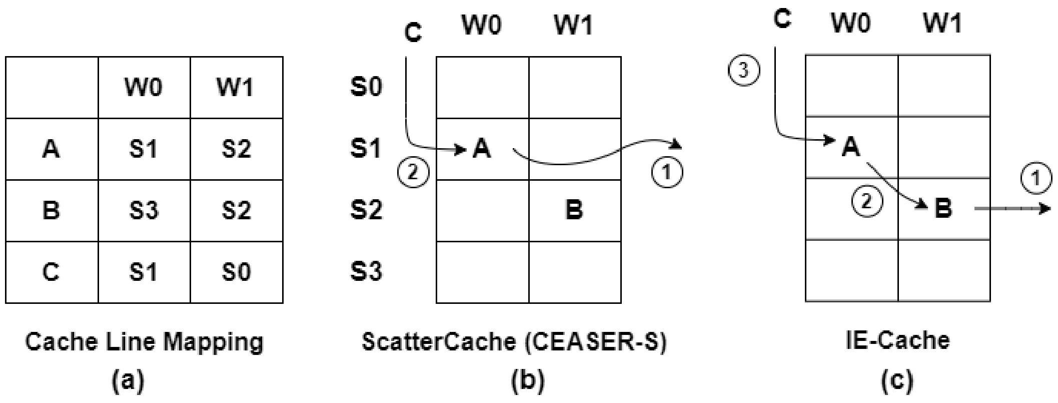
2.4.2.4 ScatterCache Cache

ScatterCache [80] is a special case of the CEASER-S cache. ScatterCache has the same number of partitions and number of cache ways. In other words, each cache way has its own unique mapping. However, instead of having multiple functions or keys for multiple cache ways, ScatterCache also suggested another solution: using either a hash function or an encryption cipher to compute the ciphertext and then truncating the ciphertext into multiple bits as the index bits of each cache way. This method avoids the overhead expansion due to the increasing number of cache partitions. With many partitions, finding the eviction set whose member has the same mapping becomes almost impossible for an attacker. To further improve the security against contention-based attacks, a Security Domain Identifier (SDID) was proposed in the ScatterCache as an add-on. The hypervisor or OS has control of the SDID assignment. Based on the security requirements of each process, the hypervisor or OS can assign different SDIDs to each process. When performing the encryption, the SDID is also treated as part of the input and consolidated with the address of the cache line as the plaintext of the encryption cipher or the input of a hash function. Hence, processes with the same security requirement can share the same cache mapping. Such an idea is similar to the TDID in Newcache [43]. The enhancement of ScatterCache further improves the protection against Contention-Based attacks on a skewed randomisation remapping cache with a re-keying function.

The ScatterCache and CEASER-S cache mentioned in Section 2.4.2.3 provide better security against contention-based attacks than the CEASER cache. However, both caches require more frequent re-keying under the state-of-the-art contention-based attack [57]. This can reduce the cache performance.

2.4.2.5 IE-Cache

Two other secure cache designs [49, 70] were proposed based on ZCache [64], which was not a secure cache design but only targeted improving cache performance. Although a skewed associative cache [66] was proposed to reduce contentions and improve cache performance, one paper [70] claims using a skewed cache on L2 or L3 caches that normally have a large size has a negligible effect on cache performance.



	W0	W1
A	S1	S2
B	S3	S2
C	S1	S0

Figure 2.6: An example of cache replacements within ScatterCache and IE-cache. (a) shows the mappings of cache line A, B, and C. (b) and (c) show the replacement procedures in ScatterCache and IE-cache, respectively.

ZCache is based on skewed associative cache [66]. A dedicated hash function drives each cache way. After the address is hashed, the cache lines are all re-distributed. The unique feature of ZCache is that the replacement in the cache is a multiple-level replacement. When a cache line is accessed and needs to replace one of the cache lines that already resides in the cache, the upcoming cache line randomly selects a cache way and finds the mapped cache set based on the index bits. After selecting the replacement victim, the victim is not evicted directly by the new cache line. Instead, this victim then becomes the first-level replacement victim. The first-level replacement victim can select the second-level replacement victim, which the first-level replacement victim cache line can replace. The number of indirection replacement levels is determined when the ZCache is designed. Nevertheless, ZCache was proposed for improving cache performance by reducing conflicts between cache lines. Hence, it does not prevent contention-based attacks.

IE-cache [49] and another design [70] add multiple cryptographic functions to encrypt the address of cache lines in Zcache for mitigating contention-based attacks, which is similar to the changes of the CEASER-S cache [60] on a skewed associative cache [66]. Processes in the cache are divided into two security domains. Each domain has its own cryptographic function's key. We provide an example to compare ScatterCache's and IE-cache's replacement differences in Figure 2.6. The mappings of cache lines A, B, and C are shown in Figure 2.6(a). We assume cache lines, A and B, have been placed in the cache. C is the incoming cache line. The cache randomly selects way 0 for cache line C when C is accessed. Based on the address mapping shown in Figure 2.6(a), cache line A currently occupies cache line C's position, which means cache line A contends with C. These are shown in both Figure 2.6(b) and (c). As shown in Figure 2.6(b), in the ScatterCache, cache line A will be evicted to make space for cache line C. Then, cache line C is loaded to where cache line A was. However, in the IE-cache with two indirection replacement levels, cache line A will be placed in another way instead of being evicted from the cache. For example, cache line A is moved to way 1 and replaces cache line B. As a result, cache line B is evicted from the cache, and the way 0 set 1 is empty for cache line C. This indirection in IE-cache hides the contentions between cache lines A and C. The attacker can observe that cache line B is evicted by cache line C rather than cache line A. This effect confuses the attacker and increases the difficulty of achieving a contention-based attack. IE-cache's drawback is that moving one cache line to another cache way needs more address calculations. In the previous example, if a cipher were used to calculate the new addresses, IE-cache requires two decryptions and two encryptions, which cause extra delays.

2.4.2.6 Mirage

Mirage [63] is another randomisation remapping cache design. This cache applies a V-way cache [62] as its substrate. Similar to the skewed associative cache and ZCache, the V-way cache was proposed to improve the cache performance by reducing cache line conflicts. Unlike the traditional cache, the V-way cache has no static one-to-one mapping between the data and tag storage. The association between tag storage and data storage are achieved by storing extra pointers, which is similar to Newcache [41]. The address is first parsed when a cache line is accessed, and the cache set is found based on the index bits. Hence, the tag is compared with all tag storage lines in the set. If it is a tag hit, the pointer in that tag storage line is used to index the data storage. Then, the data can be read out as a direct-mapped cache. When a cache line needs to be evicted from the cache, the pointers stored in the data storage can be utilised to find the corresponding tag storage and then recover the full address of the cache line. Since tag and data storage are associated with pointers, a cache line could be stored in any data storage line.

In the tag storage, the V-way cache has more tag storage lines than the data storage lines in each set. For example, an 8-way cache can be built as 12 tag storage lines and 8 data storage lines in a V-way cache. Nevertheless, the number of valid tag storage lines remains the same as that of data storage lines. The extra tag storage lines in each set ensure the cache eviction is unlikely to happen even if multiple cache lines mapped to the same cache set are accessed simultaneously. The V-way cache can then store 12 cache lines mapped in the same set by validating 4 extra tag storage lines and invalidating 4 tag storage lines from other sets.

Nevertheless, the V-way cache was not designed to defeat Contention-Based attacks. Based on its design, Mirage was proposed and added some protection schemes on the V-way cache. The address of the cache line in Mirage is skewed, which performs the randomisation remapping. Mirage used a 12-round PRINCE encryption cipher as the function. The special technique is that Mirage applied two encryption functions with a load-aware selection module. During a cache line access, Mirage does not randomly select one of the encryption functions or the keys but selects the skewed mapping whose target set has more invalid tag storage lines. The method differs from the partition selection in CEASER-S and ScatterCache. By combining a dedicated replacement policy, Mirage achieves a fully associative like cache.

The major drawback of the Mirage cache is that the overhead is relatively high compared to other randomisation remapping caches mentioned in this section. The performance of Mirage was evaluated with Firesim [35], a Rocket-core [6] based simulator. The power was evaluated by using CACTI6.0 [50]. The baseline system is configured as a processor with 8 RISC-V cores running at 3GHz and three levels of caches. The hardware overhead mainly comes from the extra storage required. Since Mirage needs more spare tags and pointer storage for tag and data storage, a 16 MB LLC implemented as Mirage needs about 20% extra storage. And the power consumption is increased by about 4W, compared to a conventional cache. Also, since the association between tag and data storage is achieved by pointers, Mirage has to search the tag and data in serial, which increases the access latency.

2.4.2.7 Chameleon Cache

Chameleon Cache [74] is an extension of the randomisation remapping cache designs. A victim cache [34], which is a small fully associative cache between different levels of memory to reduce the cache miss, is applied to a randomisation remapping cache with cache reinsertion. When a new cache line is accessed, the encrypted address is first computed with the remapping function. Based on the encrypted address, the randomisation remapping cache and the victim cache perform parallel searching of the targeted cache line. If the cache line is hit in the randomisation remapping cache, the cache hit can be handled normally. Otherwise, the cache line could be found in the victim cache. Then, the cache line is reinserted into the randomisation remapping cache. The cache way is randomly selected. If the selected place has stored another valid cache line, such a cache line is swapped with the accessed cache line and placed into the victim cache. In the last case, the accessed cache line was not found in the randomisation remapping cache or the victim cache, the cache line is then loaded from the main memory (if this is the LLC.). This loading may result in a cache line eviction from the randomisation remapping cache. However, the selected victim is moved from the randomisation remapping cache to the victim cache. When the victim cache is full, a replacement victim cache line is selected from the victim cache and evicted. As a result, when the Chameleon Cache performs an access, the evicted cache line may not conflict with the accessed cache line. This is similar to an indirect eviction in IE-Cache. Hence, it becomes difficult for an attacker to find a solid eviction set for a contention-based attack. Moreover, Chameleon applied a reinsertion mechanism to refill the cache lines pushed into the victim cache back to the randomisation remapping cache. Such a design prevents the attacker from flushing the victim cache by accessing many random uncached cache lines and exposing the contentions between the accessed cache line and the target cache line. Periodically, the cache line in the victim cache is selected and re-stored in the randomisation remapping cache. Since there are more partitions, the reinserted cache line might be reinserted with a different mapping compared to its previous mapping.

Overall, this design uses a victim cache to perform like a fully-associative cache. Although cache line storage is the same as other randomisation remapping cache, the reinsertion mechanism and victim cache reduce the contentions between the accessed cache line and the evicted cache line. Hence, the success rate with an eviction set of the Contention-Based attack is reduced. This also helps the randomisation remapping cache keep a relatively high re-keying period even if a stronger profiling method is developed. However, since the size of the victim cache must be fixed when the cache is designed, this design does not provide extra flexibility over security or performance.

2.4.2.8 PhantomCache

PhantomCache [73] is a randomisation remapping cache similar to the CEASER-SH cache design. In this cache design, the address of the accessed cache line will be re-computed based on salts, random numbers generated from a random number generator (RNG), and a dedicated hash function. When a cache line is accessed, the tag of the cache line performs XOR with the MSB part of the salt that has the same length as the tag. After performing a hash to the XORed results, the hashed value is XORed with the initial index bits of the cache line address and the rest of the bits from the salt. The cache lines are remapped in the entire cache by hashing and XORing with a random number. The cache line is searched in parallel in r different cache sets during each cache access. The cache line could be placed in any of the sets. These r cache sets are computed by using the address and r different random numbers based on the remapping method above. To perform a low latency data lookup, PhantomCache searches different cache banks in parallel. Hence, in the best case, all r cache sets are in different cache banks. The cache only needs to wait for one lookup period in the cache bank and then determines if it is a cache hit or cache miss. The default setting of PhantomCache sets r as the number of cache banks applied. If an access results in a cache miss, the cache randomly selects one of the r cache sets and applies a normal replacement policy to determine the eviction victim if all cache ways in the set are valid. Overall, PhantomCache remaps the cache using its own hash function calculated with random numbers. Since the cache largely increases the number of cache sets where a cache line can be placed, the eviction probability decreases when applying an eviction set.

However, there are several limitations to the PhantomCache design. First, the hash function used in PhantomCache is not cryptographically secure. Such a hash function was proposed for the LLBC in the CEASER cache [59], which was intended to build a function with low latency. However, using the vulnerability of LLBC [16] significantly weakens the security of the CEASER cache. A non-cryptographically secure hash function may become vulnerable. Second, since the salts are completely random, the probability that the r number of calculated cache sets are all mapped into the different cache banks is very low. Also, there could be a low probability that all cache sets are mapped to the same cache bank. Therefore, the cache may require more time for data lookup in the cache bank than in the ideal case. Third, the number r is fixed in PhantomCache. Since each cache line requires an extra log_2^r bits to store the random number that was used for the remapping, the hardware design means that the r must be a constant. If an advanced profiling method is proposed, the PhantomCache cannot enhance its security against Contention-Based attacks. Fourth, because the PhantomCache relies on multi-set searching, power consumption is a major design concern. Except for the power consumption due to the additional logic, the lookup in banks requires at least r times the dynamic power, which is dramatically increased compared to the conventional cache design.

2.4.3 Cache Monitoring

There are countermeasures based on monitoring suspicious behaviour in the cache [82, 77, 67], such as over-threshold access to a cache set. After detecting suspicious behaviour, a monitor enables some defence mechanisms against contention-based attacks. However, this may also affect the normal process operation.

2.4.3.1 SHARP Replacement Policy

Secure Hierarchy-Aware Cache Replacement Policy (SHARP) [82] is a novel replacement policy that defends inclusive caches from cross-core contention-based attacks. When a cache line is replaced in LLC, it checks this cache line's status in all private caches. The status of a cache line in LLC can fall into 3 categories: The first type is a cache line that does not exist in any of the private caches. The second type is a cache line that only exists in the private cache, which contains the process causing replacement. The third type is a cache line that exists in other private caches, which excludes the process causing the replacement. SHARP treats the first type of cache line as a priority replacement victim. If there is no first type cache line, it then searches the second type, then the third type. By replacing the first type, no contention will occur. By replacing the second type, the contention only exists in one private cache, which the other private caches from LLC cannot observe. The last type is the most dangerous replacement target since this replacement can affect other private caches. A monitor is applied in SHARP to count if the number of replaced type 3 cache lines exceeds a threshold value. If it does, the process which causes the replacement will be terminated by the OS. However, this defence strategy could be accidentality triggered by an innocent process.

2.4.3.2 SCAAT

SCAAT [67] combines the idea of cache status monitoring and cache randomisation remapping in Section 2.4.2. Unlike other randomisation remapping caches, SCAAT only remaps when the detector detects suspicious behaviour. Combining monitoring and remapping might be a good idea, but that paper did not evaluate the cache's security. Moreover, that paper did not mention which cache level can be implemented like this. From the performance evaluation and the system design in the paper, such a design could be more suitable for the L1 cache since their remapping is also achieved using a table lookup.

2.5 Countermeasures of Reuse-Based Attacks

Many countermeasures mentioned in Section 2.4 cannot defeat the reuse-based attacks since the fundamental cause of attacks is different, as discussed in section 2.3.3. For Contention-Based attacks, an attacker uses their cache line, which contains contentions with the victim's security-sensitive data, to detect if the target cache lines are accessed. However, in a Reuse-Based attack, the attack only relies on the same cache line shared between the attacker and the victim. As a result, even if the entire cache is remapped or partitioned, the cache access to the same cache line is not affected.

Sometimes, shared data are writable data. The cache requires two copies of the same data to allow writing between different users. A Reuse-Based attack can be easily defeated in such a case since the attacker and the victim do not share the same cache line. Nevertheless, in most cases, shared data are read-only. Therefore, the cache only needs to require one copy of such data, but this leaves the vulnerability to the Reuse-Based attack. The simplest solution is forcing the cache to request multiple copies of the shared data, even if the data is read-only. The drawback of the method is obvious: Since one copy of the same data is needed when a user requests the data, the cache storage may be filled with the same data and not be efficiently used. Hence, the performance of the cache is degraded.

Some coutermeasures against reuse-based attacks were proposed [82, 42, 52]. We will briefly discuss some of these countermeasures in this section.

2.5.1 SHARP

As we have mentioned in section 2.4.3.1, SHARP [82] was proposed against both Contention-Based attacks. Nevertheless, SHARP also proposed the idea to counter a Reuse-Based attacks. SHARP limits the *clflush* instruction, a fundamental instruction used in the Flush+Reload attack [85]. Since the *clflush* instruction was used to maintain cache coherency, it cannot be entirely disabled. Nevertheless, SHARP modified the permission of the *clflush* instruction. In the normal user mode, the user cannot use such instructions to evict any read-only pages. As most of the shared libraries are read-only files, this prevents the attacker flushing the target cache lines easily. Since writing on the shared data can allow the cache to request another copy from the main memory, which is known as Copy-on-Write (CoW), this does not expose any cache line status to the attacker.

2.5.2 Random Fill Cache

Random fill Cache Architecture [42] was proposed against reuse-based attacks. Compared to SHARP, this cache design is a passive countermeasure. This work determines that the root of reuse-based attacks is the correlation between cache fill and demand memory access. In other words, the cache lines existing in the cache must be accessed before a process. For example, in the *Flush+Reload* attack, after the attacker evicts the shared data, if the victim user accesses the target cache line, the cache line must be brought back to the cache. This deterministic causation is the core factor of the attack.

Based on this point, the *Random fill Cache Architecture* modified the conventional cache filling strategy and applied it to the L1 cache. When a cache miss occurs in the L1 cache, the demanded cache line will not be cached in L1 but directly forwarded to the processor. Then, one cache line is randomly selected and cached from a pool formed by the neighbours of the demanded cache line. The Random fill cache only performs the storage of this selected cache line but does not send it to the processor during this access. As a result, the cache state change is only caused by the randomly selected cache line. Hence, the attacker cannot directly learn the cache line usage from the victim process and the cache state. Although the accessed data is not stored in the cache, filling a neighbouring cache line in the cache still guarantees the performance of a cache. This is due to the spatial locality of the memory; if a particular memory location is referenced once, then the nearby locations are likely to be referenced shortly. Nevertheless, the Random fill Cache Architecture was proposed for the L1 cache and may not be suitable for L2 or L3 caches.

2.6 Advanced Contention-based Attacks on Randomisation Remapping Cache

This section focuses on how contention-based attacks can be achieved on a randomisation remapping cache since this is essential for evaluating the security of such a cache design. As explained in Section 2.3.3, a contention-based attack's core procedure is using a few cache lines from the attacker process to evict the security-sensitive data. This eviction could be done easily on a conventional cache but not on a randomisation remapping cache. Therefore, a critical phase, finding those cache lines which can evict security-sensitive data, namely the *Profiling Phase*, must be done at the beginning of the attack. Many advanced profiling methods have been proposed in the past 5 years [71, 75, 57]. These methods weaken the randomisation remapping cache's protection against contention-based attacks.

After the profiling phase, the attacker uses those identified cache lines to implement many rounds of contention-based attacks on any of the targets mentioned in Section 2.3.4. The attack processes have been discussed in Section 2.3.3. Such a phase is called the *Exploitation Phase* [58]. These two phases will be discussed within the following two subsections. We will also explain the state-of-the-art profiling method [57].

2.6.1 Profiling Phase

The profiling phase is unnecessary when attacking a conventional cache. The index bit of the address determines the cache set. Therefore, the attacker only needs to allocate themselves $S \times W$ continuous cache lines in the attacker program, and then they have sufficient cache lines to occupy a whole level of cache. S and W indicate the number of sets and ways in a level of cache. This method becomes impossible in a randomisation remapping cache since all addresses are remapped using a cryptographic primitive or a table. After the remapping, those $S \times W$ cache lines will be randomly distributed and not continuous anymore. Therefore the attacker cannot ensure these cache lines are sufficient to occupy a cache or even a specific cache set. To successfully achieve a contention-based attack, the attacker must find a set of cache lines that guarantees the eviction of the victim cache line or has a higher probability than randomly picking addresses. This set of cache lines is called the *Eviction Set* [42]. As mentioned at the beginning of Section 2.6, finding and constructing enough such eviction sets is called the *Profiling Phase.*

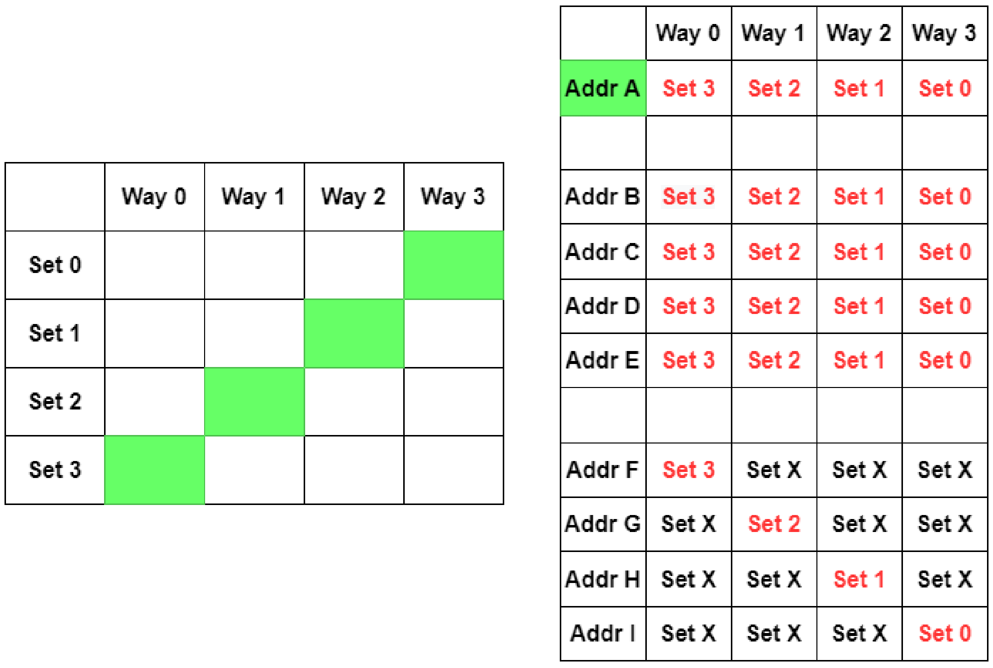
2.6.1.1 Profiling For Fully Congruent Eviction Sets

Fully Congruent Eviction (FCE) Sets are the opposite of *Partially Congruent Eviction (PCE) Sets*, which indicates that all members in this eviction set have the exact mapping as the targeted cache line. There are no PCE Sets in conventional caches and caches that do not have partitions, for example, a CEASER cache [59]. The definitions and the differences will be further discussed in Section 2.6.1.2.

Some profiling methods for finding the FCE sets were proposed [42, 60]. These methods use an algorithm to filter out the cache lines that can cause contentions in a cache set. More specifically, when attacking a cache with static mappings, such as a conventional cache or CEASE cache [59], repeatedly loading the cache with cache lines will eventually cause an eviction. This indicates that one cache set is full, and the last cache line belongs to the cache set. The attacker can finally find the cache lines that contend in a cache set by accessing part of the accessed cache lines and examining if the contention still exists. These cache lines then form a FCE set.

2.6.1.2 Profiling For Partially Congruent Eviction Sets

ScatterCache [80] was claimed to defeat a contention-based attack that applied an FCE set. As discussed in Section 2.4.2, ScatterCache and CEASER-S (we call them CEASER-S-like Caches below) combine the CEASER cache [59] and Skewed cache [66], significantly reducing the possibility of finding a cache line with the exact mapping as a target cache line. For a CEASER-like cache, if a cache line F can evict the target cache line A, the attacker can guarantee that before the next re-keying, cache line F is mapped to the same cache set as cache line A in all cache ways. For a CEASER-S-like Cache, this will not be true. Since each way has its own mapping, cache lines F and A may be allocated into the same cache set of way 0, but with an extremely high probability that they do not share the same set in other cache ways. As a result, cache line F may or may not evict cache line A when cache line F is accessed next time. Such a cache line is called as *Partially Congruent Cache Line.* This cache line F can be used to form a *Partially Congruent Eviction (PCE) Set.* Relatively, the attacker still has an very small chance of finding a cache line B with the exact mapping as cache line A in all cache ways. Such a cache line B is called a *Fully Congruent Cache Line*, which can be used to form a FCE set as mentioned in Section 2.6.1.1.



(a)

	Way 0	Way 1	Way 2	Way 3
Set 0				🟩
Set 1			🟩	
Set 2		🟩		
Set 3	🟩			

(b)

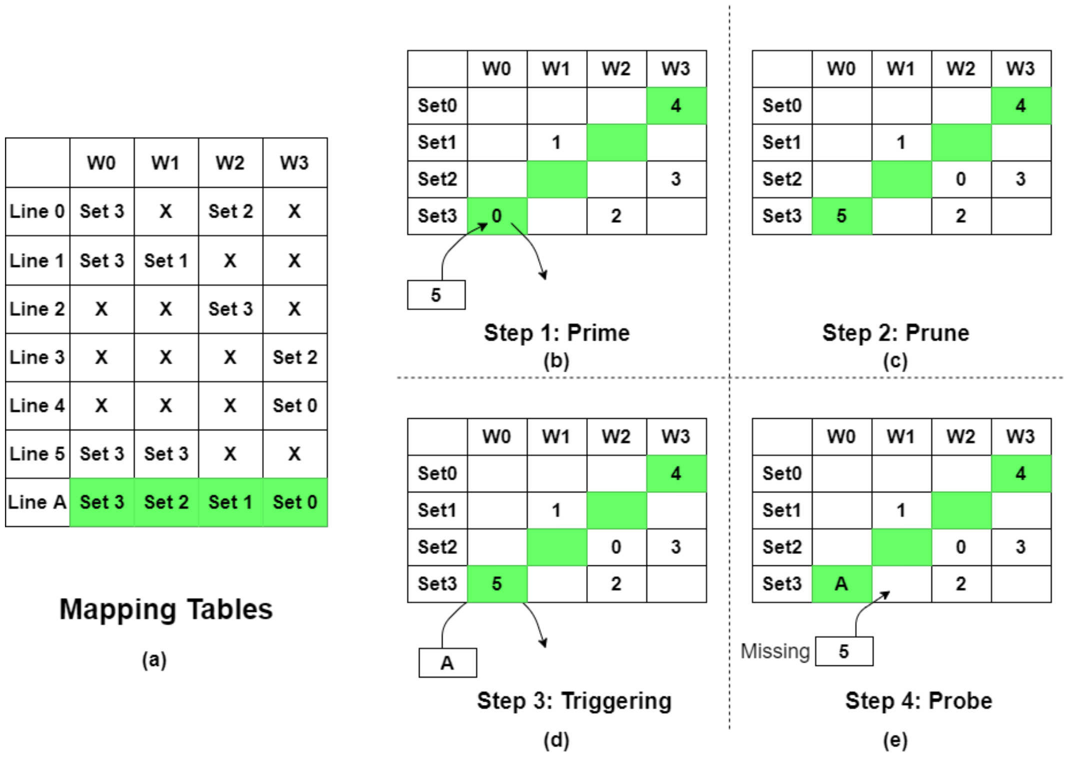
	Way 0	Way 1	Way 2	Way 3
Addr A	Set 3	Set 2	Set 1	Set 0
Addr B	Set 3	Set 2	Set 1	Set 0
Addr C	Set 3	Set 2	Set 1	Set 0
Addr D	Set 3	Set 2	Set 1	Set 0
Addr E	Set 3	Set 2	Set 1	Set 0
Addr F	Set 3	Set X	Set X	Set X
Addr G	Set X	Set 2	Set X	Set X
Addr H	Set X	Set X	Set 1	Set X
Addr I	Set X	Set X	Set X	Set 0

FIGURE 2.7: An example of a FCE set and PCE set. Address A is the victim cache line. Address B, C, D, and E can form a FCE set and addresses F, G, H, and I can form a PCE set.

An example of a PCE set and an FCE set are shown in Figure 2.7. In Figure 2.7 (a), there is a 4-way CEASER-S cache with 4 sets and 4 partitions. In Figure 2.7 (b), address A is the victim cache line which could be placed into any labelled position in the cache. Addresses B, C, D and E have the exact mapping as address A, so they can form a FCE set with 4 eviction cache lines. Addresses F, G, H, and I only have the same mapping with address A in at least one way, and the mappings in other ways can be ignored. They can form a PCE set with 4 eviction cache lines. The probability of finding a fully congruent cache line can be calculated as $P = \frac{1}{S^W}$, where S and W indicate the number of cache sets and cache ways, respectively. As fully congruent cache lines are extremely difficult to find in a CEASER-S-like cache, building an eviction set with fully congruent cache lines also becomes impractical. However, building a PCE set is the only possible method of attacking a CEASER-S-like cache. Therefore, Scattercache was also evaluated by applying a PCE set [80]. However, the evaluation was still not accurate because the paper made a non-precise assumption: The eviction probability must be $\geq 99\%$. Due to this assumption, Scattercache's security against contention-based attacks was overestimated.

Later, **Prime+Prune+Probe** (PPP) was proposed as a new profiling method [57]. This method enhances and generalises the profiling of PCE sets. PPP decreases the number of accesses needed to a practical number. We explain the profiling procedure with an example in Figure 2.8. To avoid confusion with eviction cache lines in Figure 2.7, we use numbers to represent candidate cache lines.

The first step of profiling is called Prime. The attacker randomly selects K cache lines and loads them into the cache. These K cache lines are initial candidates for profiling. When loading these cache lines into the cache, some cache lines could be evicted by other cache lines within these K cache lines. Hence, the number of cache lines that are left in the cache is K'. In the example of Figure 2.8, the attacker loads 6 cache lines (cache line 0 to 5) into the cache. As mentioned before, the cache way is selected randomly. Based on the cache mapping table in Figure 2.8(a), the cache lines are placed as shown in Figure 2.8(b). All potential placements of the victim cache line, cache line A, are highlighted in green. Both cache lines 0 and 5 are placed in cache way 0. Since both cache lines are mapped to set 3 way 0, cache line 5, which is accessed later than cache line 0, evicts cache line 0. Finally, 5 out of 6 cache lines reside in the cache. In this example, K is 6, and K' in this step is 5.



	W0	W1	W2	W3
Line 0	Set 3	X	Set 2	X
Line 1	Set 3	Set 1	X	X
Line 2	X	X	Set 3	X
Line 3	X	X	X	Set 2
Line 4	X	X	X	Set 0
Line 5	Set 3	Set 3	X	X
Line A	Set 3 🟩	Set 2 🟩	Set 1 🟩	Set 0 🟩

	W0	W1	W2	W3
Set0				4 🟩
Set1		1	🟩	
Set2		🟩		3
Set3	0 🟩		2	

	W0	W1	W2	W3
Set0				4 🟩
Set1		1	🟩	
Set2		🟩	0	3
Set3	5 🟩		2	

	W0	W1	W2	W3
Set0				4 🟩
Set1		1	🟩	
Set2		🟩	0	3
Set3	5 🟩		2	

	W0	W1	W2	W3
Set0				4 🟩
Set1		1	🟩	
Set2		🟩	0	3
Set3	A 🟩		2	

FIGURE 2.8: An example of the PPP profiling process. (a) shows the table of cache line mappings, X indicates do not care. (b,c,d,e) shows the four steps of PPP profiling.

In the second step, the attacker re-accesses those K addresses a few times, called pruning. By doing this, there could be more addresses being cached. In other words, K' could increase while the number of repetitions of the second step increases. When those evicted cache lines are re-accessed during the pruning, they have a significant chance of being placed in another way. Therefore, they may finally get cached. In this case, K' is incremented. In the worst case, this cache line could conflict with any of the K addresses again, and finally, one cache line is evicted, and another is cached. In this case, the value of K' maintains the value. Overall, the number of cached cache lines K' could get very close to or even equal K by repeated pruning. If there are still cache lines conflicting after a few iterations of pruning, the attacker can also withdraw those addresses and use K' addresses for the later steps, which is also called aggressive pruning. In our example, the attacker re-accesses 6 cache lines, which is shown in Figure 2.8(c). Since cache line 0 was evicted in the previous step, it is loaded back again. This time, cache line 0 may be placed into another cache way, for example, way 2. Based on its mapping in Figure 2.8(a), cache line 0 is placed in set 2 way 2, which currently is empty. Since other cache lines are still in the cache, they will stay where they were.

In the third step, the attacker triggers or waits for the victim process. If the attacker is lucky, the victim's cache line can replace an attacker's cache line. This depends on the capturing probability of profiling, which will be discussed later. In our example in Figure 2.8(d), cache line A is the victim cache line. After the victim process is triggered, cache line A is loaded into the cache. Cache way 0 is randomly selected by the cache. Based on the mapping, cache line A should be placed in set 3 way 0. Cache line A evicts cache line 5, which currently occupies this position.

Furthermore, in the last step, the attacker re-accesses K' cache lines and examines which one was missing. The missed cache line is then added to a PCE set. In our example in Figure 2.8(e), after re-accessing those six cache lines, the attacker can notice cache line 5 is missing. Therefore, cache line 5 is added into a PCE set, similar to the address F in Figure 2.7. Other members of a PCE set can also be found like this. It is worth noting that cache line 4 in Figure 2.8 may become another PCE cache line since it has the exact mapping as cache line A in way 3.

As mentioned in step three, the probability of a victim cache line evicting one of the K' cache lines is called the capturing probability [1]. This probability depends on the cache mapping, replacement policy and if the victim cache line has already been cached before the attacker's profiling starts. When the victim cache line is not cached, and the replacement policy is the Random replacement policy, each member of K' has an equal probability of being evicted by the victim cache line. Therefore the capturing will only depend on the coverage of K' of total $S \times W$ cache lines or as a formula:

$$P_C = \frac{K'}{S \times W} \tag{2.1}$$

[1]In the paper of PPP[57], this is named the catching probability. We rename it as the capturing probability to avoid confusion.

It is also worth noting that W cache lines are not enough for a PCE set. When using the FCE sets, the attacker ensures W cache lines occupy all places where the victim could be placed. When using the PCE set, this will become probabilistic rather than deterministic. If the number of cache lines within a PCE set increases, the attacker can have a better chance to reserve the places where the victim cache line could be placed. Such probability is named the *Eviction Probability*, P_E. As a trade-off, if the attacker wants an eviction set with a higher eviction probability, the profiling time required will be increased. The assumption of ScatterCache [80], which is that the eviction probability must be 99%, hugely increases the profiling time. The attacker uses an eviction set with a lower probability which takes less time to build, and can still achieve a contention-based attack [76]. The eviction probability depends on the size of the eviction set and the replacement policy. For attacking when the Random replacement policy is applied, the probability can be calculated as equation 2.2. G indicates the size of the eviction set.

$$P_E = 1 - (1 - \frac{1}{W})^{\frac{G}{P}} \tag{2.2}$$

There is another issue of profiling that could reduce the probability of building the eviction set. After the attacker profiles one round, the victim cache line may still reside in the cache. This may not be a problem if the second round of the profiling successfully primed the victim cache line. However, if the attacker did not successfully evict the victim cache line, both the victim cache line and the eviction set's candidates will remain in the cache, and no eviction cache line can be found in this round. A solution to this issue is either accessing many addresses to push out the eviction set or bearing a lower success rate of the attack [58].

2.6.2 Exploitation Phase

During the exploitation phase, the attacker uses the eviction sets obtained from the profiling phase to attack the target using one of the contention-based attacks. The primary process of implementing an attack is similar to what has been mentioned in Section 2.3.3, but since the eviction set does not guarantee the eviction of the victim cache line, the attacker may need to take some extra effort. After the attacker primes, the victim cache line is accessed and may evict one of the cache line from the eviction set. During the probe step, the eviction set is re-accessed. If the victim cache line was not evicted by any cache lines from the eviction set, the victim cache line could still remain in the cache after the probe step. This can disrupt the second round's exploitation and further reduce the attack success rate, unless the attacker somehow pushes the victim cache line out, such as evicting the entire cache [80].

Another major distinction comes out when the second round of attacks starts. For a conventional attack using the FCE set, the attack does not need to be repeated on the same victim cache line, but due to the P_E of a PCE set, the attack on the same victim cache line might need to be repeated a few times. After the first round of exploitation on an address, the PCE set still resides in the cache. This may not become an issue if at least one eviction cache line occupies a place that could conflict with the victim cache line, but it could reduce the eviction probability if fewer places with conflicts are occupied. Otherwise, the eviction set becomes useless in the worst case when none of it occupies any conflict places and is stuck in the cache. The attacker may use the *clflush* instruction to flush out the eviction set before the next round of exploitation so that the eviction set can still be reused [76]. Overall, this increases the difficulty of implementing a real contention-based attack.

2.7 Simulation Tools For Cache Evaluations

To evaluate a secure cache design, designers normally need to consider several aspects of the cache, such as cache performance, cache security against specific attacks, hardware overhead and power consumption [59, 80, 63, 74, 73]. To perform these evaluations, designers require some dedicated tools to simulate the proposed cache design and compare it with the existing cache designs based on the results. We introduce the simulators and tools that are used in the later chapters.

The gem5 simulator [15] is a well-known open-source simulator, which can be used to perform computer architecture simulations. The simulator supports multiple instruction set architectures, including X86, SPARC, MIPS, ARM, etc. By modifying the cache models in the gem5 simulator, the designer can evaluate the cache performance such as cache miss rate, Instructions Per Cycle and Miss Per Kilo Instructions, under specific workloads.

CACTI6.0 [50] is a widely used cache access modelling tool to estimate the delay, power and area. Since building a cache on a physical chip normally requires a dedicated memory compiler and lots of effort in building and testing the cache system, most secure cache designs [59, 80, 63, 74, 73] evaluate their designs in the above aspects by estimating the cache modifications with CACTI6.0. It allows users to modify cache configurations, such as cache size, associativity, access mode, etc. It also supports four different process technologies: 90nm, 65nm, 45nm, and 32nm. Nevertheless, CACTI6.0 cannot estimate the overheads caused by the additional modifications of a dedicated secure cache controller. To overcome this, the designer must evaluate the cache control logic modifications separately.

To estimate the cache control logic, at least a gate-level modified cache controller for the proposed cache design is needed. To do this, for evaluating our caches, we modify the control logic of the SiFive inclusive cache [23] and synthesised the logic by using an open-source Nangate 45nm technology library [69]. We will further discuss these in Section 4.6.

Based on the contention-based attack with PPP profiling that was discussed in 2.6, we develope our own security simulator to evaluate the cache security against the contention-based attack. This will be explained in Section 3.3.

As a short summary, Table 2.4 lists all the tools and evaluation factors.

Evaluation Factors	Tools
Cache Performance	gem5 simulator [15]
Cache Delay, Power, Area	CACTI6.0 [50] , SiFive inclusive cache [23], Nangate 45nm technology library [69]
Cache Security	Self-developped Security Simulator

TABLE 2.4: A summary of cache evaluation factors and tools.

2.8 Summary

The cache is used to store data temporarily, and has been used in CPU design for decades. Due to the memory temporal locality, the accessed data will likely be accessed again. The cache helps the cores hold more recently used data and avoid a longer stall in the cores for directly accessing the data from the main memory, dramatically improving the processor's performance. However, the normal behaviour of the cache can also be vulnerable. Compared to a cache miss, a cache hit can show different results regarding power, timing, etc. For example, an attacker could deduce if a specific cache line was accessed based on timing differences. This is named as a timing side-channel attack. By obtaining knowledge of the cache line residency, the attacker could retrieve some useful information without privilege, such as an other user's partial encryption key.

Many countermeasures have been proposed against cache timing side-channel attacks [80, 60, 63, 43]. Most secure cache designs focus on cache contention-based attacks. This is because a contention-based attack does not require shared data between the attacker and victim processes. Therefore, a contention-based attack is more dangerous than a reuse-based attack. The countermeasures can be split into two types: Active defence, such as a cache monitor, and Passive defence, such as randomisation remapping using a cryptographic function to obfuscate the entire cache mapping. Passive defence protects all users so that it cannot be bypassed. We summarised major cache timing side-channel attacks and their countermeasures on Table 2.5.

Although some existing randomisation remapping cache designs have dramatically increased the difficulty of implementing a contention-based attack, a newer profiling method

Table 2.5: The summary of major cache timing side-channel attacks and their countermeasures.

Type of attacks	**Defence Strategy**	**Countermeasures**
Contention-Based (Prime+Probe, Evict+Time [53])	Cache Partitioning	PL Cache [79]
		NoMo Cache [26]
	Cache Remapping	Newcache [43]
		CEASER Cache [59]
		CEASER-S Cache [60]
		ScatterCache [80]
		IE-Cache [49]
		Mirage [63]
		Chameleon Cache [74]
		PhantomCache [73]
	Cache Monitoring	SHARP[83]
		SCAAT [67]
Reuse-Based (Flush+Reload [85], Cache Collision[18])	Disabling CLFLUSH	SHARP[83]
	Bypass Cache Filling	Random Fill Cache[42]

could still weaken such protection against an attack. Increasing the re-keying frequency could be a solution, but a higher re-keying frequency can lead to cache performance degradation. Other randomisation remapping cache designs provide more protection against contention-based attacks but have huge area or power overheads.

Based on the state-of-the-art profiling method mentioned in Section 2.6.1.2, we observed that cache associativity can significantly impact the implementation of contention-based attacks. Hence, we propose a novel idea called logical associativity, which will be introduced in chapter 3. The logical associativity allows the user to modify the associativity dynamically while the processor is powered on. When a contention-based attack is upgraded by the attacker, the user can adjust the logical associativity, preventing huge performance degradation due to the high re-keying frequency. We propose two different cache designs based on logical associativity in chapters 4 and 5. Using the tools introduced in 2.7, we also evaluate these cache designs.

Chapter 3

Logical Associativity

As we discussed in Section 2.4, many cache designs against contention-based attacks have been proposed in the past 15 years. Last-level cache protection has been considered since 2015. Randomisation remapping cache, which we discussed in Section 2.4.2, has become a practical solution and focus for the secure last-level cache. However, since contention-based attacks have become stronger, some randomisation cache designs that provide fixed protection have been cracked and are vulnerable [59].

Some other randomisation cache designs with frequent remapping (re-keying) or permanently strong protections have been proposed to fix the security gap caused by new contention-based attacks. Nevertheless, older designs could be forced to have a high re-keying frequency, which causes significant performance loss, if a new contention-based attack is developed in the future. The later designs may require huge overheads and have performance degradation while providing greater protections.

In this chapter, we explain a novel idea: **logical associativity**. This idea can be added to existing cache designs like the CEASER-S cache [60], to enhance the security against contention-based attacks or to balance the performance of the cache.

We first explain the motivation for logical associativity in Section 3.1. Then, in section 3.2, we discuss the three properties of logical associativity, reconfigurability, overlapping and alienation, and how these properties enhance the security against contention-based attacks. Section 3.3 discusses the simulator we implemented for security evaluation. Section 3.4 explains how the indexing function was selected based on the performance evaluation. Finally, section 3.5 provides a summary of logical associativity.

3.1 Motivation

While the randomisation remapping cache has been improved, the implementation of contention-based attacks has also been refined. However, nobody can predict how aggressive contention-based attacks could be in the future. As the hardware designer, we must not build up a protection scheme for the cache which is only valid against contention-based attacks with the current implementation complexity level, since a more complex attack with a stronger profiling method could be proposed later and becomes a more dangerous threat. However, providing very strong protection against contention-based attacks may cause unnecessary performance degradation. As a result, secure cache designs against contention-based attacks should provide flexibility to allow users to adjust between performance and security, based on their requirements.

Based on other research [57, 86], we found that the cache parameters such as cache size, replacement policy and cache associativity could affect the difficulty of implementing a cache contention-based attack. Cache associativity especially plays a significant role. Also, the high re-keying frequency of the randomisation remapping cache can significantly reduce the cache performance. Therefore, we propose a novel idea which gives further flexibility, the flexible associativity of randomisation remapping caches. By increasing the associativity and keeping the re-keying frequency relatively low, the cache could enhance its security against more aggressive contention-based attacks while the performance degradation is smaller. We name this method **Logical Associativity**.

3.2 Introduction of Logical Associativity

In this section, we explain the idea of logical associativity. We first introduce what logical associativity is, and then discuss three properties of logical associativity: Reconfigurability, Overlapping and Alienation. We will also explain how these properties can help the randomisation remapping cache enhance its security against contention-based attacks.

3.2.1 Logical Associativity

In most existing randomisation remapping last-level cache designs [59, 60, 80], as mentioned in 2.4.2, when a cache line needs to be accessed, the address of the cache line is encrypted by a dedicated function. When the encrypted address is ready, it can be segmented into two parts, encrypted tag bits and encrypted index bits. The bit width of the encrypted tag and encrypted index bits are the same as the tag and index bits in the original address, respectively. The encrypted index bits are used to index the search in the cache. Unlike existing designs, after the encrypted index-bits are calculated, the cache can access not only the set pointed to by the encrypted index-bits, namely the **Home Set**, but the following $H-1$ cache sets as potential placement positions; we name this **logical associativity**.

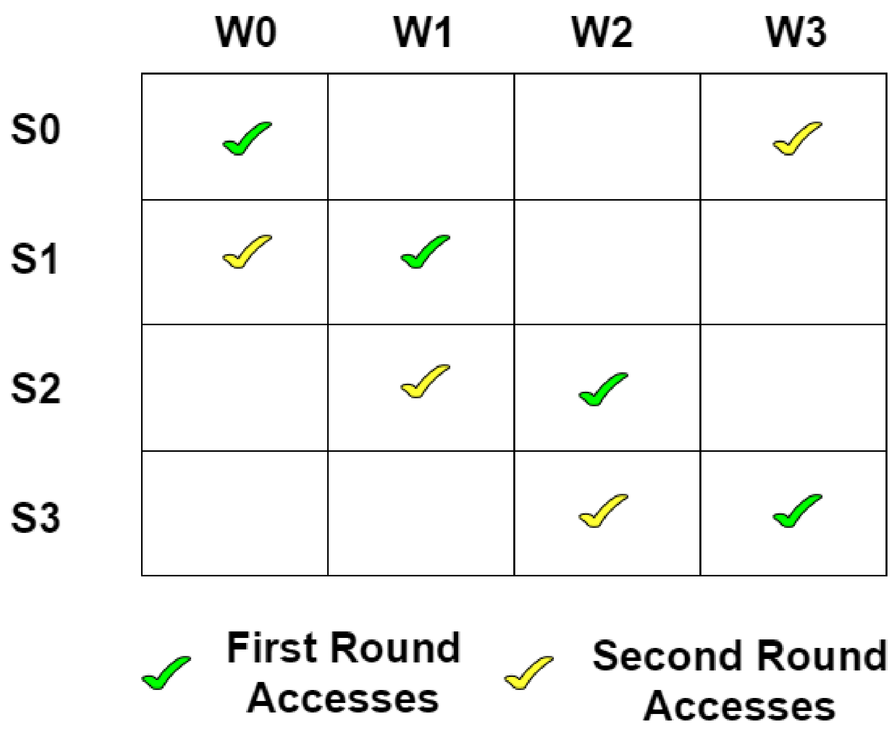


FIGURE 3.1: An example of cache access order of the CEASER-S cache with logical associativity. The cache has 4 cache ways, 4 cache sets and logical associativity of 2. Green and yellow check marks indicate first round and second accesses, respectively.

These H cache lines form a logical cache set, and the value H is the logical associativity. Within this logical set, each cache line is a logical cache way. For example, we add the logical associativity to the existing cache, CEASER-S cache. In Figure 3.1, we show the access orders in this cache. This is a cache example with 4 cache ways, 4 cache sets and $H = 2$. After the encrypted index bits are calculated based on the appropriate key, each cache way receives the corresponding cache set number. The set numbers are 0,1,2,3 in cache ways 0,1,2,3, respectively. In CEASER-S or ScatterCache, the cache will only search the cache lines in $S0W0$, $S1W1$, $S2W2$, and $S3W3$. S indicates the set number, and W indicates the way number. These blocks are highlighted in green. For a cache with logical associativity, after the cache searches these 4 cache lines, namely the home set, the cache then performs burst-lookup-like accesses and accesses the following cache ways in the corresponding cache set. The cache lines $S1W0$, $S2W1$, $S3W2$, and $S0W3$ are accessed. Since S3 is the last physical set in the cache, the following cache set of $S3W3$ is $S0W3$.

There are three properties of logical associativity. When it is applied to a randomisation remapping cache, these properties can help the original cache enhance security against contention-based attacks. The first property is that the associativity becomes adjustable, which is **Reconfigurable**. Secondly, due to the increase in associativity, the placement of some cache lines can cause **Overlapping**, which can further increase the attack complexity. Lastly, **Alienation** allows different users to have different associativity. We explain these three properties further in the later subsections.

3.2.2 Reconfigurability

We know that increasing the cache associativity in a randomisation remapping cache could dramatically increase the contention-based attack complexity. The reason that a high associativity could affect the attack complexity is different from re-keying. As discussed in Section 2.4.2.2, re-keying limits the maximum time for an attack. Therefore, the attacker can only construct an eviction set with limited sizes. A high associativity leads to more potential placement positions for the cache accesses. In other words, by applying an eviction set with the same size, the attacker can only occupy a smaller portion of the victim's potential placement positions when the associativity is larger. However, a higher associativity always means higher cache access latency. An extreme example would be a fully associative cache, which may not be practical for a large L3 cache implementation.

The reconfigurability of the cache allows the privileged user, for example, a cloud vendor, to dynamically increase the associativity. Caches with self-reconfigurability have been proposed for many years, for example the SeReMo cache [30]. Although the original goal of the self-reconfigurable cache is to reduce the performance impact when high contention misses are detected, this strategy is also useful to counter a contention-based attack. In the SeReMo cache, each of the four cache lines is segmented as a module, the reconfigurability is achieved by re-setting the connection netlists between the modules, which is similar to the reconfiguration of an FPGA. When the associativity of the cache is doubled, the set number is halved. In other words, SeReMo temporarily changes the cache's physical arrangement.

However, logical associativity achieves reconfigurability in another way. We do not need to modify the original mapping of the randomisation remapping cache but we can add another module, namely a logical associativity unit, to store the logical associativity settings and send extra requests to the tag and data storage of the cache. We will explain this module in Section 4.3.1. The logical associativity can only be set by the privileged user. After the machine is booted, the privileged user can still modify the logical associativity. However, the modification is limited to increasing the logical associativity. To decrease the logical associativity, flushing the entire cache or over-ranged cache lines is needed. This is because the cache lines placed with the old logical associativity may not be found with the new logical associativity. An example is shown in Figure 3.2. The Figure 3.2(a) shows a CEASER-S cache with logical associativity, and the initial logical associativity is set to 1 (equivalent to no logical associativity). A cache line **A**

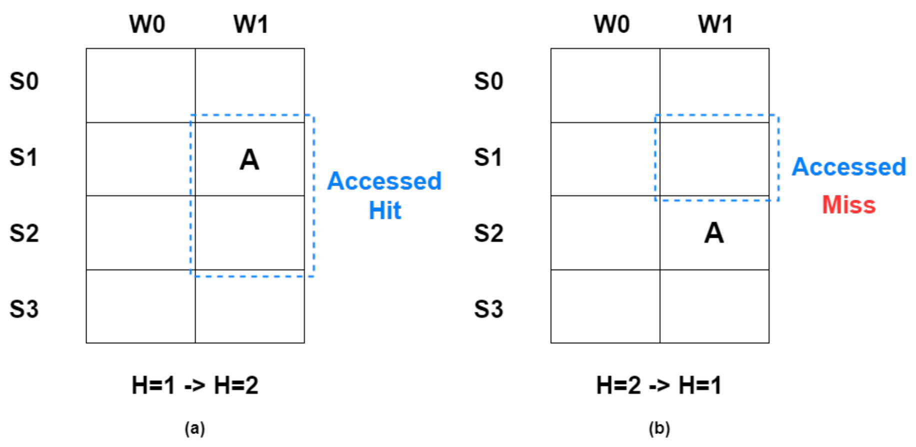


FIGURE 3.2: Two examples of the impact of the logical associativity changes on the existing cache lines. (a) logical associativity is increased from 1 to 2. (b) the converse case. Now, the cache line **A** is not accessed. This can cause an error.

is placed in $S1W1$ under $H = 1$. If H is now increased to 2, the cache will search set 1 and set 2 in way 1. Therefore, the cache line **A** could still be found with the new logical associativity. However, Figure 3.2(b) shows the opposite case. The cache line **A** was placed with $H = 2$. Although the cache line has the encrypted index bits as 1, the cache line **A** is placed in $S2W1$. When the logical associativity decreases to 1, the cache will only search the home set of way 1, $S1W1$. Hence, accessing cache line **A** leads to a cache miss. Another copy of the cache line **A** will then be sent to the cache, which causes a fatal error in the cache operation due to a memory coherence error.

It is worth noting that increasing the logical associativity is not equivalent to increasing the physical (or conventional) associativity or the number of partitions which is the number of different mappings shared by cache ways. Partitions are explained in Section 2.4.2.3. In a skewed randomised remapping cache, such as CEASER-S [60], there are no correlations between the mappings of each physical cache way. Although the attacker cannot observe the H set numbers when the logical associativity is set to H, the mappings in the later $H - 1$ sets all depend on the Home set number. From such a perspective, logical associativity is more similar to the partitions in the CEASER-S cache. The major difference is due to the second property of logical associativity, which is overlapping as explained next.

3.2.3 Overlapping

As discussed in Section 3.2.2, not only logical associativity could cause the associativity reconfigurations but also other cache designs [30]. Ideally, doubling the physical associativity should have equivalent effectiveness against contention-based attacks as logical associativity. However, this is not true, due to the overlapping of logical associativity. Overlapping indicates the overlapping potential placements between different logical cache sets. Such overlapping could reduce the eviction probability of the attacker's eviction set. Furthermore, this changes the success rate of contention-based attacks. To explain the overlapping, we first provide three examples in Section 3.2.3.1 to show the implementation of a contention-based attack on the original CEASER-S cache with different configurations and without logical associativity. We then compare these with Contention-Based attacks on the CEASER-S cache with logical associativity in Section 3.2.3.2.

3.2.3.1 Without Logical Associativity

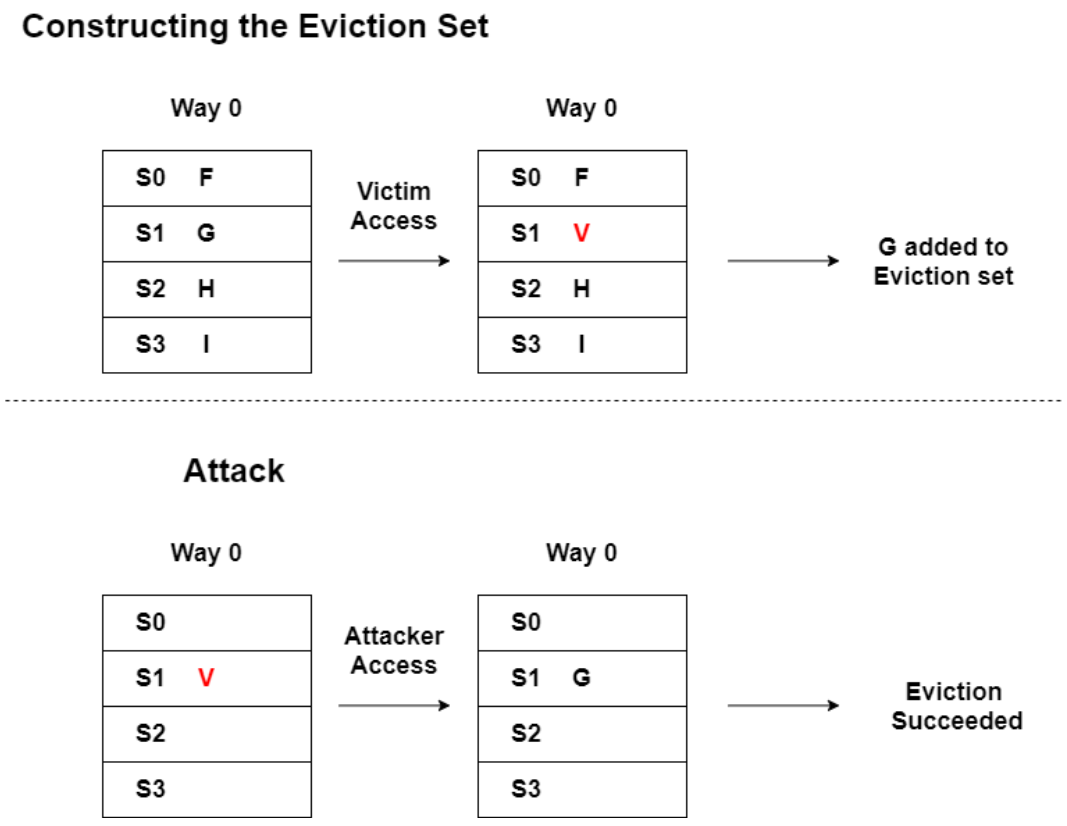


FIGURE 3.3: An example of the Prime+Probe attack on a CEASER-S cache with one way and one partition.

A simplified example of a contention-based attack on an example CEASER-S cache with one cache way is shown in Figure 3.3. We assume the victim cache line V has the encrypted index bits set to 1. By following the PPP profiling method [58], the attacker first randomly accesses some cache lines and tries to occupy $S1$. After a few rounds of pruning, cache lines F, G, H, and I finally reside in $S0$, $S1$, $S2$, and $S3$. After the victim access the cache, cache line G is evicted by cache line V. Hence, by re-accessing those cache lines, the attacker could access cache line G, which results in a cache miss. Then, the attacker adds cache line G into their eviction set. During the attack phase, the attacker accesses their eviction set. Since G has the same encrypted index bits as V, which is 1, G can only be placed in $S1$ again. Finally, the attacker deduces the victim cache line V was accessed due to another cache miss of cache line G.

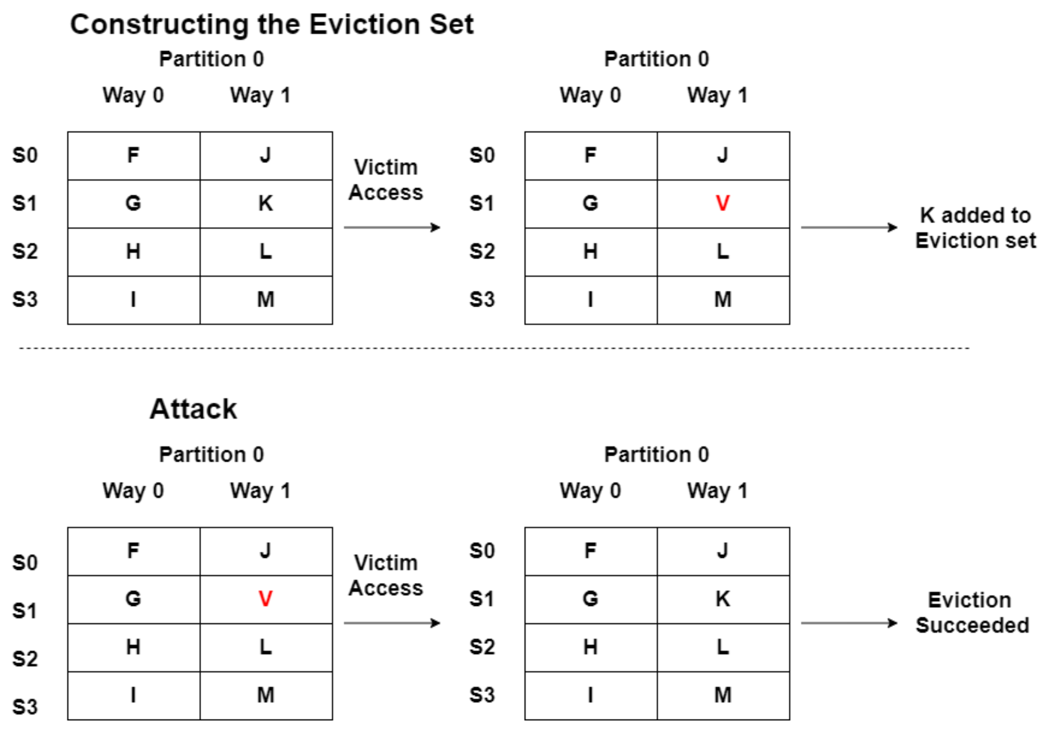


FIGURE 3.4: An example of the Prime+Probe attack on a CEASER-S cache with 2 ways and 1 partition.

As a comparison, we show a similar example on a CEASER-S cache with two ways but one partition, in Figure 3.4. During the construction of the eviction set, the victim cache line V evicts cache line K, and K becomes a member of the eviction set. During the attack, the cache line V is reloaded into the cache. However, since the cache way is selected randomly, cache line V could be placed in either cache way 0 or 1. Since both ways share the same partition, K can also be placed in either way 0 or way 1. Hence, the probability that K can evict V is 50%, which is less than CEASER-S with one cache way and one partition in Figure 3.3. In Figure 3.4, the victim cache line V is successfully evicted by cache line K.

Another example is shown in Figure 3.5. We apply similar conditions as the previous two examples, except we change the cache to 2 cache ways and 2 partitions. During the construction of the eviction set, the victim's cache line V evicts the cache line G in $S1W0$. Then, cache line G becomes a member of the eviction cache set. During the attack, cache line V could be placed in either way 0 or way 1. In this example, cache line V has been previously stored in way 1. Since way 0 and way 1 are two different partitions, in way 1, cache line V may to be mapped to a different cache set than $S1$. In this example, it is placed in $S2W1$. Another cache line Z was placed in $S1W0$ due to previous accesses. Because cache line V is only congruent cache line G in way 0, the attacker cannot predict where cache line G could be mapped to in cache way 1. It is very unlikely to be mapped to the same set as cache line V, in a real-sized cache. In other words, the eviction probability of cache line G being in way 1 is equivalent to that of a random cache line.

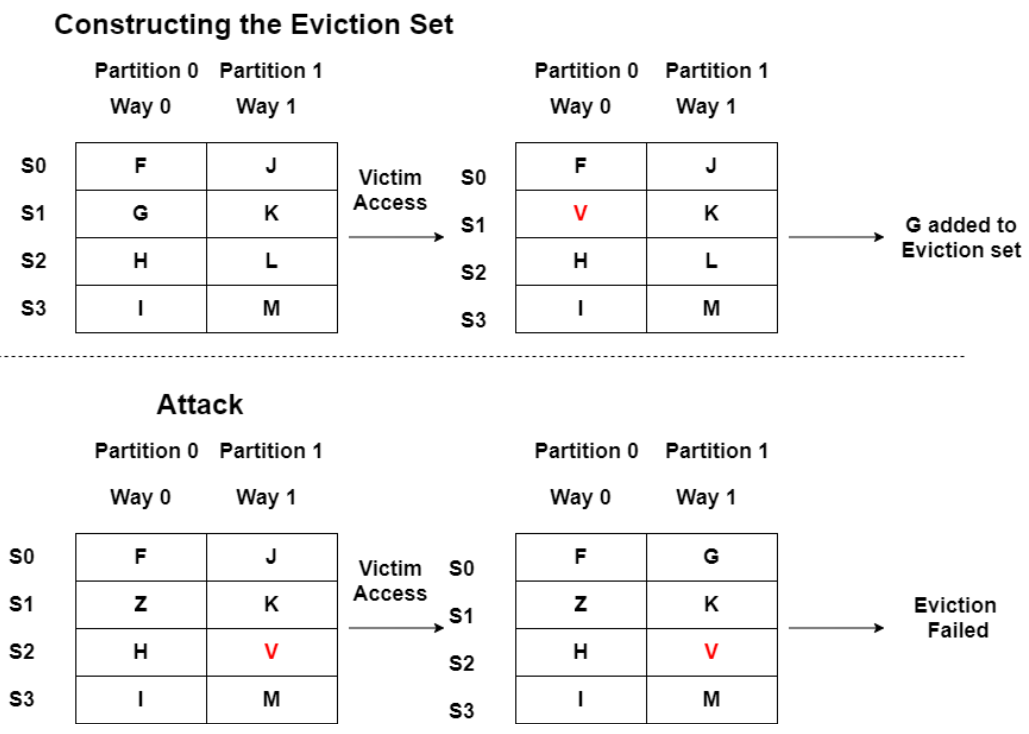


FIGURE 3.5: An example of the Prime+Probe attack on a CEASER-S cache with 2 ways and 2 partitions.

Based on these three examples, we can see that achieving a contention-based attack on a randomisation remapping cache like the CEASER-S cache becomes more complicated when the associativity or the number of partitions is increased. When the cache has one way and one partition, the eviction is always successful since the eviction set can always occupy the position where the victim cache line will reside. When the cache has two ways and one partition, the attack success depends on whether the eviction cache lines and the victim cache line are mapped to the specific partition and the same cache way. This can be any of the cache ways in that partition. When the cache has two ways and two partitions, the attack almost can only succeed if the eviction cache lines and the victim cache line are mapped to the specific cache way.

3.2.3.2 With Logical Associativity

We now provide an example of the logical associativity on a CEASER-S cache with one cache way and $H = 2$ is shown in Figure 3.6. As with CEASER-S, F, G, H, and I still reside in $S0$, $S1$, $S2$, and $S3$. During the victim access, since $H = 2$, the cache line V belongs to logical set 1. This means that the cache line V could be placed in either S1, which is its home set, or S2. In the example, we assume cache line V is placed in $S1$

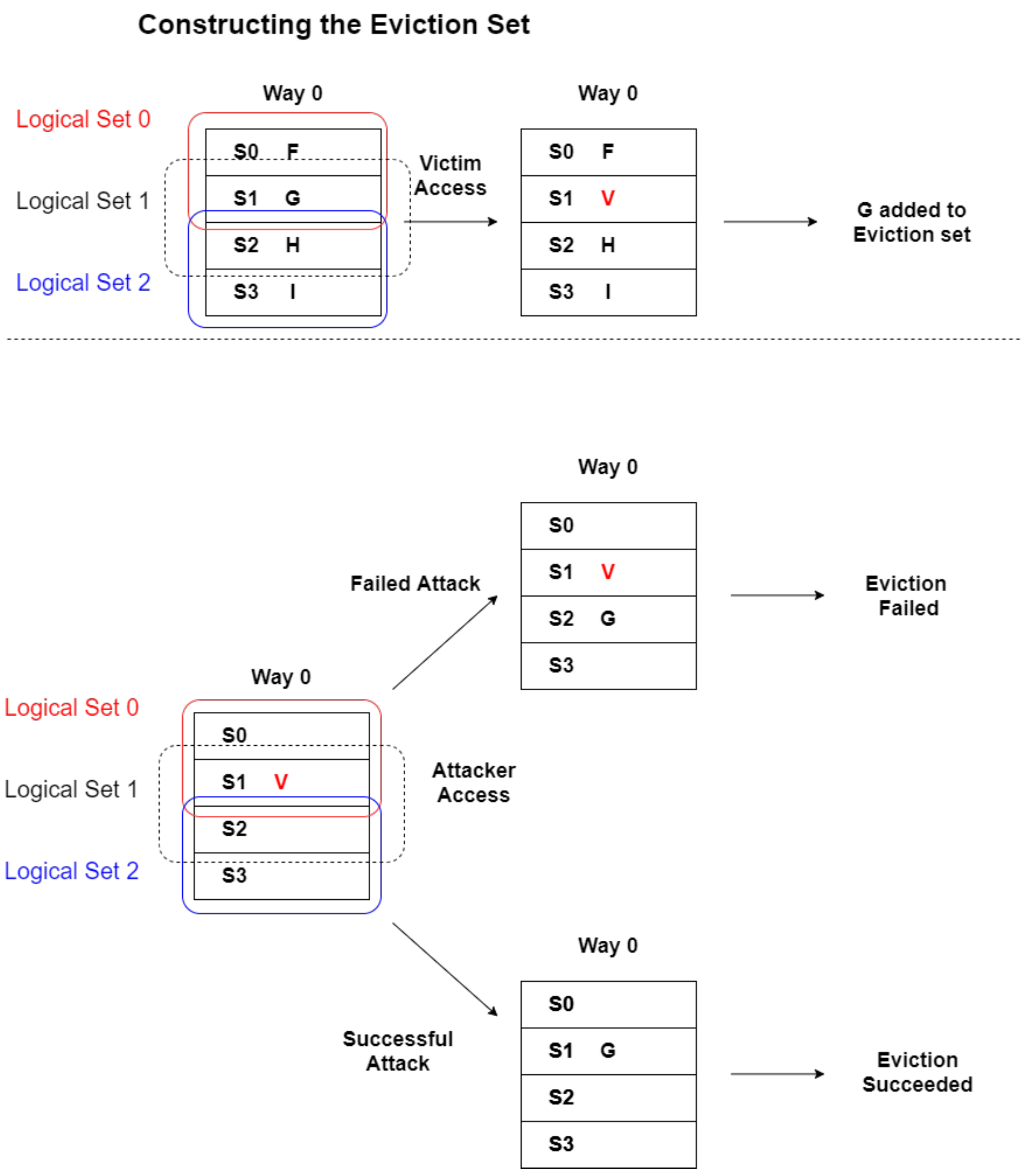


FIGURE 3.6: The first case of the Prime+Probe attack on a CEASER-S cache with logical associativity (Both the victim cache line and eviction cache line share the same logical set.) .

during the eviction set construction. Cache line G, which was placed in $S1$, is evicted by the victim. Hence, similar to other examples in Section 3.2.3.1, the cache line G is added to the eviction set. However, here comes the uncertainty for the attacker. Such a cache line G may belong to logical set 0, or logical set 1. In other words, the eviction cache line may be in the same logical set as the victim-targeted cache line, or there is a partial set-overlapping in the logical set of both the eviction cache line and the victim-targeted cache line. Also, when the logical sets of the victim cache line and the eviction cache line partially overlap, the placement of the victim cache line during the attack can make the attack completely different. We will discuss these three cases separately.

In the first case, both G and V are from logical set 1. Hence, in the attack phase, the attack succeeds if G is placed into the same cache set as V. This is similar to the example of CEASER-S with one partition and two cache ways in Figure 3.4. In Figure 3.6, we show examples of a successful attack and a failed attack. When the attack succeeds, both cache lines must be placed in the same set. Otherwise, for example, G is placed in S2 but V is in $S1$, so the eviction of this round failed.

In the second and third cases, which are distinct from CEASER-S without logical associativity, the eviction cache line belongs to a different logical cache set. For example, cache line G now belongs to logical set 0. Compared to the first case, these cases have no obvious difference during the construction of the eviction set for the attacker. However, a huge impact in terms of the attack complexity can be seen in the attack phase, as explained in the following paragraphs. In the second case, the victim cache line is placed in the overlapping range of both cache lines' logical sets. Whereas in the third case, the victim cache line is placed outside the overlapping range.

An example of the second case is shown in Figure 3.7. The victim cache line V is still placed in S1, which is within the overlapping range between logical set 0 and logical set 1. Hence, the attack succeeds when G is also placed in $S1$, which is within the overlapping range. Otherwise, the attack fails if G is placed in the $S0$.

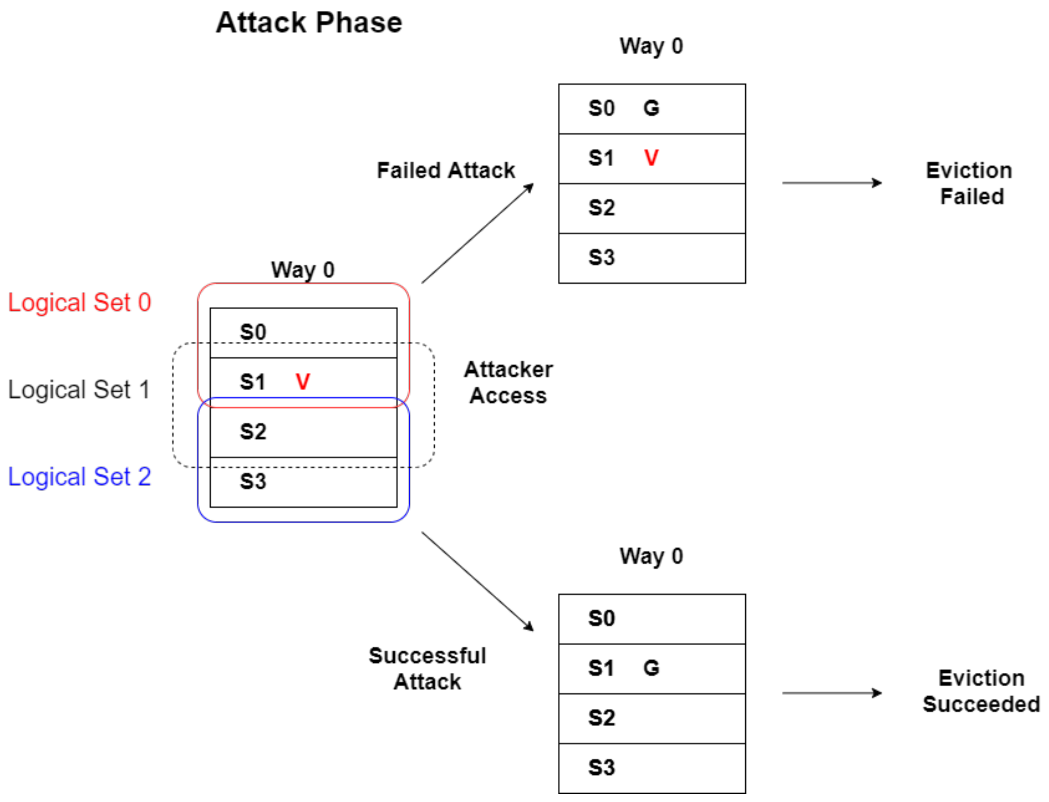


FIGURE 3.7: The attack phase under the second case of the Prime+Probe attack on a CEASER-S cache with logical associativity (The victim cache line and eviction cache line are from different logical sets, and the victim is placed in the overlapping range.) .

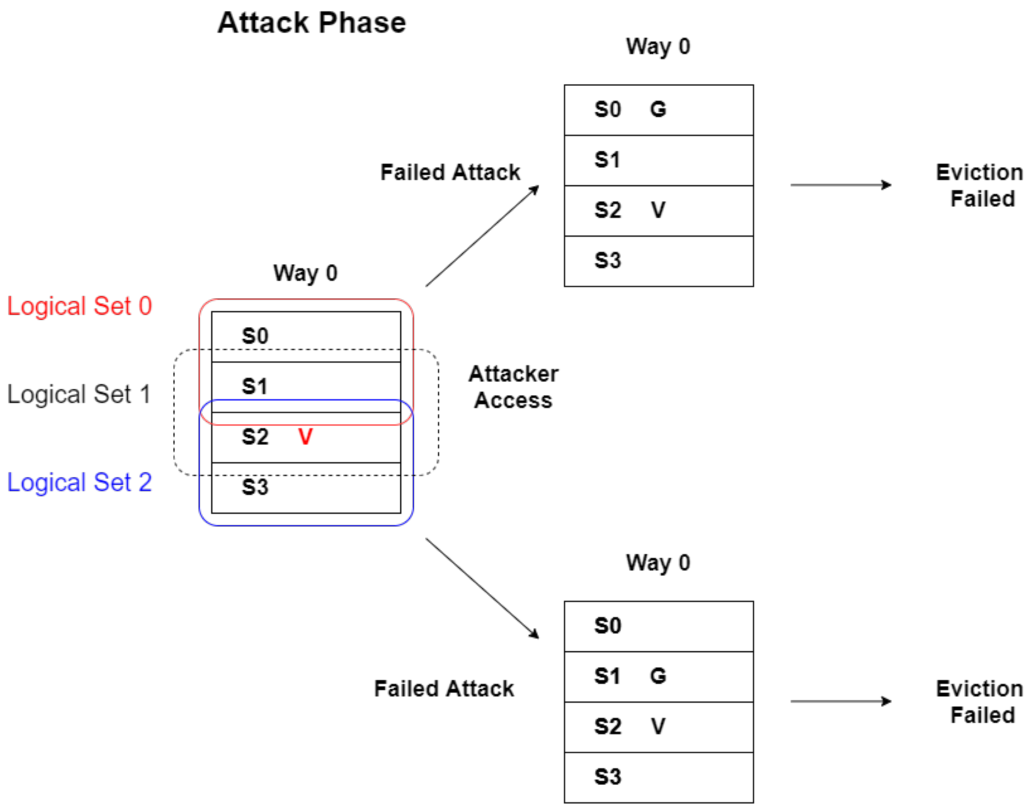


FIGURE 3.8: The attack phase under the third case of the Prime+Probe attack on a CEASER-S cache with logical associativity (The victim cache line and eviction cache line are from different logical sets, and the victim is placed outside the overlapping range.)

The third case is shown in Figure 3.8. The victim cache line V is now placed in S2, which is different from the placement during the construction of the eviction set. S2 is outside the range of logical set 0, therefore, the attacker's cache line G cannot evict V at all. In other words, the eviction cache line becomes useless.

Compared to the two examples without logical associativity shown in Figure 3.3 and Figure 3.4, adding logical associativity reduces the effectiveness of the constructed eviction cache lines. Although such a method is less effective than increasing both the physical associativity and partitions, logical associativity still provides stronger protections against contention-based attacks than increasing physical associativity only, which can be achieved in the SeReMo cache [30].

To prove this, we set up an experiment. We use our own simulator (the simulator is further discussed in Section 3.3) to compare the effectiveness against a contention-based attack of increasing logical and physical associativity. In the experiment, three different CEASER-S cache configurations are compared under Prime+Probe attack, including 16 ways and 16 partitions, 32 ways and 16 partitions, and 32 ways and 32 partitions. These configurations are applied to an L3 cache with an 8MB size, which is the same cache size as in other research [60, 57]. Hence, it allows us to provide a fair comparison and verify our simulation results based on their previous theoretical results.

Table 3.1: The success rate of **Prime+Probe** attacks with different associativity. (H indicates the logical associativity, and P indicates the number of partitions. H=1 indicates no logical associativity.)

Eviction Set Set Size (G)	**Success Rate (%)**			
	16 ways		**32 ways**	
	P=16			**P=32**
	H=1	***H=2***	***H=1***	***H=1***
176	50	23	29	16
576	90	58	68	44
752	95	67	78	53

We use three cache configurations to compare with the logical associativity of 2, 16 physical cache ways and 16 partitions on a CEASER-S cache. Since the eviction set is only valid before the entire cache is re-keyed, we ignore the re-keying function for this experiment. Here, we assume the attacker has successfully built eviction sets with three sizes (176, 576, 752), which is in line with other work [57]. The success rates of Prime+Probe on CEASER-S with 16 ways, shown in Table 3.1, align with other theoretical results [57]. Compared with CEASER-S with 16 cache ways and 16 partitions, all other caches dramatically reduce the success rates of the Prime+Probe attack. For example, for the eviction set with 752 members, the success rate on a CEASER-S with 16 cache ways and 16 partitions is about 95%. However, this value is reduced to 67%, 78% and 53% in other cache configurations, respectively. The logical associativity reduces the CEASER-S cache attack success rates compared with CEASER-S with 32 ways and 16 partitions, but is still higher than CEASER-S with 32 ways and 32 partitions. This also applies to attacks using different eviction sets of different sizes.

In conclusion, logical associativity is not just adding a potential placement position for the cache lines, like increasing the physical cache ways but maintaining the partitions. With overlapping, it further reduces the effectiveness of the eviction set in an attack so that the success rate of implementing a contention-based attack will be lower. Although increasing both the number of partitions and physical cache ways can be more effective against a contention-based attack, this can only be achieved by physically redesigning the cache architecture, which contradicts the reconfigurability of the randomisation remapping cache. Overall, logical associativity improves on the SeReMo cache [30] in mitigating contention-based attacks.

3.2.4 Alienation

Alienation is another property of logical associativity, which allows cache accesses with different IDs to have different logical associativity settings. This ID can be a user ID or a process ID. By providing finer-level protection against contention-based attacks, the performance degradation can be limited only to the users or processes that require extra protection. Hence, this can improve the overall cache performance. In this subsection, we only introduce alienation but do not discuss the implementation. The implementation will be discussed in Chapter 5.

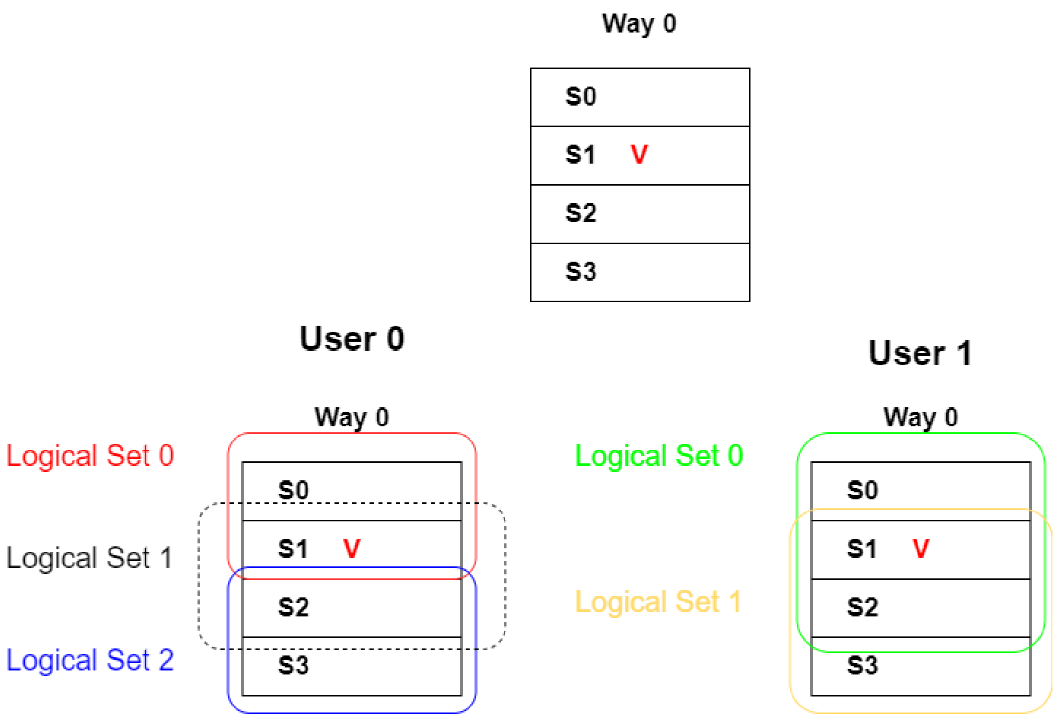


FIGURE 3.9: The alienation example of the logical associativity. User 0 has a logical associativity of 2, and user 1 has a logical associativity of 3.

As we mentioned in 3.2.2, logical associativity does not change the physical connection netlists in the tag and data storage. Therefore, if the cache knows which process or user requires what logical associativity, those accesses could be made with distinct logical associativity. In Figure 3.9, as an example, we show the different logical associativity views from different users when alienation is applied. When the cache receives a request, the cache distinguishes the user by their ID. For example, if a processor core is limited to a specific user each time, the cache only needs to know which core requests the cache line. After parsing the ID, the cache can access the cache bank with the corresponding logical associativity. Due to the reconfigurability, the logical associativity setting can even be increased. User 0's logical associativity setting is 2 and user 1's logical associativity is 3. Since higher logical associativity may lead to higher cache access latency, the user who has the lower logical associativity can have a better performance. The lowest logical associativity can be limited to a privileged user.

3.3 Security Simulator

3.3.1 Brief Introduction to Security Simulator

We developed a security simulator to evaluate the Prime+Probe attack with PPP profiling [57] on randomisation remapping caches. This simulator evaluates the security of randomisation remapping caches with different configurations under contention-based attacks. Unlike in the performance evaluations in the later content, we did not use the gem5 simulator to evaluate security. The reason is that we are not evaluating the current Prime+Probe attack but the Prime+Probe attack with an advanced profiling method. Hence, we need to evaluate the security under the worst case. We made some attacker-friendly assumptions. It is worth noting that in a real attack, these assumptions are hard to achieve. Therefore, the success rate in a real attack should be less than or equal to the results in our simulation. The assumptions are: there is only one targeted victim cache line. An attacker can use all of the time for profiling before re-keying. The attacker could trigger the access of the targeted cache line itself. Also, an attacker can use *clflush* or other equivalent methods to aviod the issues of that the victim cache line and the PCE sets may stuck in the cache during the profiling phase and the explotation phase. These issues can reduce the attack success rate, which has been discussed in 2.6.1.2 and 2.6.2. Later, we analyse the obtained eviction set. Under a real attack or on the gem5 simulator, this is hard to achieve.

Furthermore, in a real attack, the attacking code quality may also affect the attacking quality. Since the randomised remapping cache with re-keying counts the number of cache accesses, a bad attacking implementation could waste some cache accesses and reduce the size of constructed eviction set.

3.3.2 Security Simulator Implementation

As mentioned in 3.3.1, the simulator implements the Prime+Probe attack with PPP profiling [57]. The structure of the simulator is shown in Figure 3.10.

We separate the simulator into two parts. The first part profile the constructing of the eviction set. The addresses of the victim-targeted cache line and all cache lines within the eviction set are stored in a file. The second part of the simulator implements the Prime+Probe attack using the stored eviction set. By setting the iteration rounds, each eviction set is used many times. By calculating the average success rate of the attack when applying different eviction sets, the Prime+Probe attack success rate on a specific cache is evaluated.

In the profiling phase, the simulator allows the user to define basic cache parameters such as cache size, cache physical associativity, and replacement policy. Since the simulator is used for randomisation remapping cache with or without logical associativity, the simulator also allows changes to partitions, indexing function, re-keying period, the maximum size of eviction set, and logical associativity. These parameters can be set by a user script. In terms of the indexing function, our simulator supports SHA-3 and PRINCE, we will discuss this in Section 3.4. In the simulator, we use a two-dimensional array as Tag storage. Since the data storage itself is unnecessary, we ignore it in the simulator. Each cache line has a dedicated variable to store its access status. This status can be used for replacement policies like LRU.

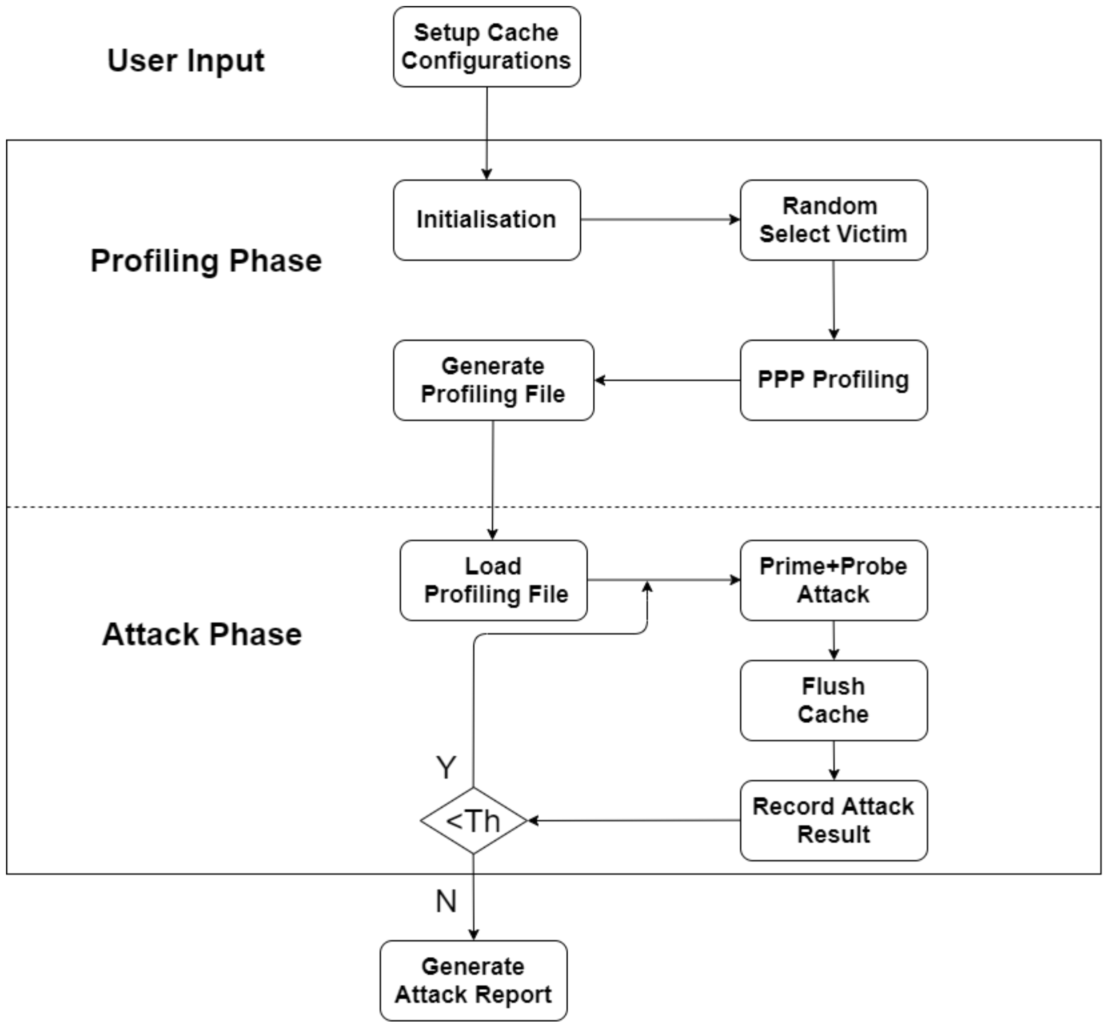


FIGURE 3.10: The structure of our security simulator.

After the simulator is initialised, it randomly selects a value as the address of the victim-targeted cache line. This value is also saved in the external file for the attacking phase. All cache lines and their status in the cache are set to 0. Then, the actual profiling starts. The simulator performs PPP profiling to the cache. The pruning step, which is the second step of PPP profiling, starts after the fifth round. In each round, the number of cache accesses is recorded. Also, the eviction cache line is recorded. The PPP profiling runs until the cache access number reaches the re-keying period, or the eviction set has reached the expected size. It is worth noting that these two cases are for two different circumstances. The former is for constructing the eviction set with the maximum size within the fixed number of accesses. The latter one is for constructing a fixed-size eviction set.

In the attacking phase, the user can set the rounds of the Prime+Probe attack for the eviction set and its targeting victim cache line. The simulator reloads the addresses of the entire eviction set and the victim target cache line. When using the eviction set for the attack, we also prune the eviction set after the prime stage. This is to avoid self-eviction and make use of most cache lines in the eviction set. After each round of the Prime+Probe attack, the simulator records if the round of the attack succeeded, or in other words if the victim cache line evicts any of the eviction set. The total success rate is calculated after all rounds are finished.

3.3.3 Security Simulator Validation

As we discussed in section 3.3.1, the simulator applies the PPP profiling [57] on randomisation remapping caches to construct PCE sets. Then, the simulator can measure the success rate of evicting a targeted victim cache line when using a constructed PCE set. Therefore, to validate the simulator, we need to construct and evaluate PCE sets of some random addresses and compare them to the theoretical results obtained from the equation 2.2 under different cache configurations.

In detail, we applied 10 different random addresses as the victim addresses in the simulator and constructed the PCE sets with sizes (G) of 176, 576 and 752 under three different cache configurations. These configurations include a CEASER-S with 16 ways and partitions, a CEASER-S with 32 ways and 16 partitions, and a CEASER-S with 32 ways and partitions. The random replacement policy is utilised. Each victim address and the corresponding PCE set are tested $10k$ times in Prime+Probe attacks. The success rate of the attack represents the eviction probability (P_E) in the equation 2.2. The success rates under different cache configurations are calculated as the average success rates when applying different victim addresses. The simulated results and the theoretical results are shown in Table 3.2. By comparing the simulated results and the theoretical results, we found that the results are very close under different cache configurations. In details, for example, under a CEASER-S with 16 ways and partitions, the success rates of attacking 10 different addresses when applying an eviction set with the size of 176 are 50.46%, 50.74%, 49.63%, 51.09%, 49.16%, 49.89%, 50.15%, 49.78%, 49.60% and 50.10%. The average success rate is 50.1%, which is very close to the theoretical result of 50.8%. Hence, the simulator is verified.

Table 3.2: The **Prime+Probe** success rate comparisons of the simulated results and the theoretical results under different cache configurations. The values in the brackets indicate the theoretical results. (P indicates the number of partitions.)

Eviction Set Set Size (G)	**Success Rate (%)**		
	16 ways	**32 ways**	
	P=16		**P=32**
176	50.1 (50.8)	28.9 (29.5)	15.8 (16.0)
576	89.6 (90.2)	68.0 (68.1)	43.8 (43.5)
752	94.8 (95.2)	77.8 (77.5)	52.5 (52.6)

3.4 Cryptographic Function Selection

As mentioned in ScatterCache [80], two types of cryptographic functions, SCv1 and SCv2, could be applied to the randomisation remapping cache. The randomisation remapping cache with a SCv2 type function, which uses the full range of the encryption ciphers' ciphertext as the encrypted address, requires one cryptographic function for each partition, which could be costly in terms of the hardware overhead in a multi-partition randomisation remapping design. The SCv1 type function, which only requires partial ciphertext for indexing each cache way, is more flexible because the output size does not need to be identical to the width of cache index-bits. From other research [57], we know that more partitions provide much better protection against contention-based attacks. Therefore, we set the remapping cache as the full partition. In other words, each cache way is indexed by different encrypted index bits. Because other research [16] has exposed the vulnerability of the LLBC cipher originally proposed in the CEASER cache [59], another suitable cryptographic function needs to be selected.

Although ScatterCache [80] suggested hash functions or many low latency block ciphers can be used as the SCv1 function, we first need to compare the difference between using a hash function and an encryption cipher. We simulated and compared two different CEASER-S implementations with a hash function SHA-3 [24] and an encryption cipher PRINCE [19]. The reason for selecting these functions are: 1. Both functions are secure enough for our designs. SHA-3 is the latest member of the Secure Hash Algorithm family. PRINCE is the popular candidate to replace the original LLBC in the CEASER cache. This has been proposed elsewhere [80, 63, 16]. 2. There are open-source implementations of both SHA-3 and PRINCE available. Therefore, we do not need to build one from scratch.

We think an encryption cipher, such as PRINCE, is more suitable for being the cryptographic function of the randomisation remapping cache. The main reason is that hardware-implemented hash functions may require many clock cycles to compute the output, even for some lightweight hash functions or some other ciphers, e.g. Quark [7], Hash-One [47], Present [17] [39]. However, the cache performance is sensitive to latency. Therefore, an encryption cipher, such as PRINCE which only needs 3 clock cycles, is preferred.

Using encryption ciphers as an SCv2 type function, could provide a better mapping. This is because encryption ciphers are invertible so the inputs and outputs must be paired one-to-one. Conversely, hash functions can cause collisions. Due to these collisions, the cache lines may not be normal distributed in all cache sets. Hence the cache performance is degraded. Nevertheless, since we are using both functions as SCv1-type functions, this advantage does not apply to the design.

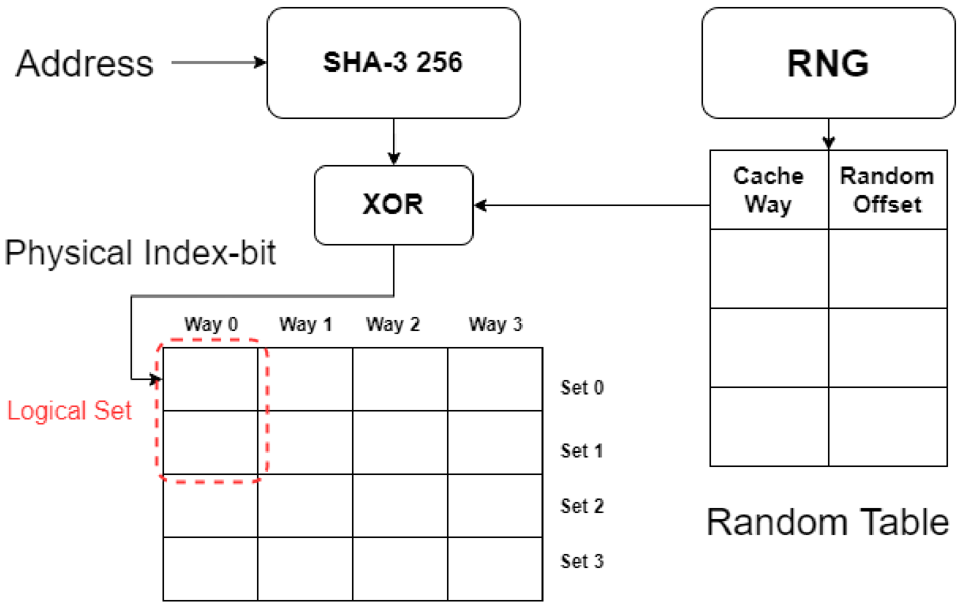


FIGURE 3.11: The structure of a SHA3-256 indexed CEASER-S cache.

To validate this hypothesis, we implemented these two functions on the existing CEASER-S design and simulated them on the gem5 simulator [15]. The design with SHA-3 is shown in Figure 3.11. When a cache line is accessed in the cache, the original address of this cache line will be passed to the hardware-implemented SHA-3 256. The hashing output can be segmented into a few parts. Each part has the same length as the index bits needed for the CEASER-S and can have bits overlapping. Since SHA-3 does not rely on keys, the re-keying must be performed in a different way. Therefore, after the output of the SHA-3 is available, each part of the output will be used to derive the physical index-bit by calculating an XOR with a relative entry from a table called the Random Table. This table stores random numbers generated from a random number generator, which is the same as the generator used in the CEASER cache [59], with the same length as the index bits. Each cache partition has a random number stored in this table. This random number should be updated frequently.

For testing, we used the Mibench suite [29] as the benchmark. The latency of both PRINCE and SHA-3 was considered in the simulation. The latency of PRINCE is 3 clock cycles [19]. A basic implementation of the SHA-3 latency is 24 cycles [72]. In terms of the cache configurations, we utilised an ARM O3 CPU model at 2GHz. Both L1 and L2 caches are set as private caches. Since the performance difference between different cache models is not obvious when using large caches, we set the cache size relatively small. The size of each level of cache is 16K, 32K and 256K. The physical associativity is 4, 8 and 16, respectively. We did not apply re-keying in this experiment since we mainly compared the differences caused by the cryptographic functions themselves. The simulations in the later chapters do consider the re-keying. Also, it is worth noting that we only used these small configurations to compare these two functions. We applied a different cache configuration to all later simulations. The small configuration is used because the performance differences are more obvious when using MiBench. However, these simulations are sufficent for determining the suitable indexing functions.

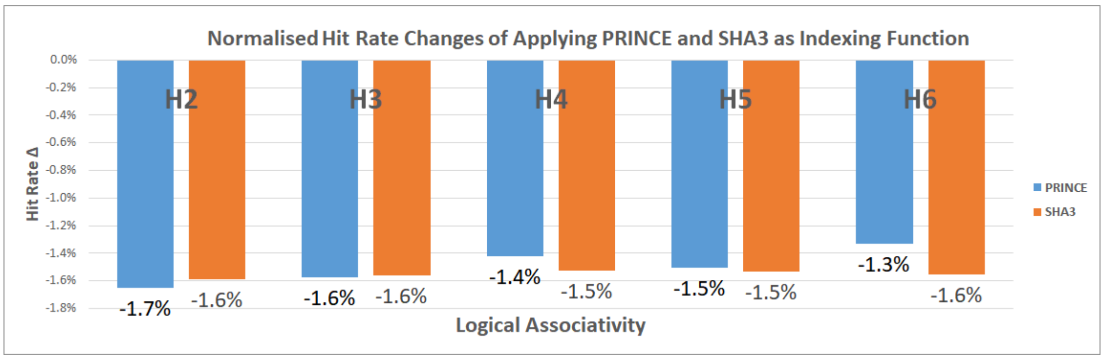


FIGURE 3.12: The comparison of normalised hit rate changes between applying PRINCE and SHA3 as indexing function.

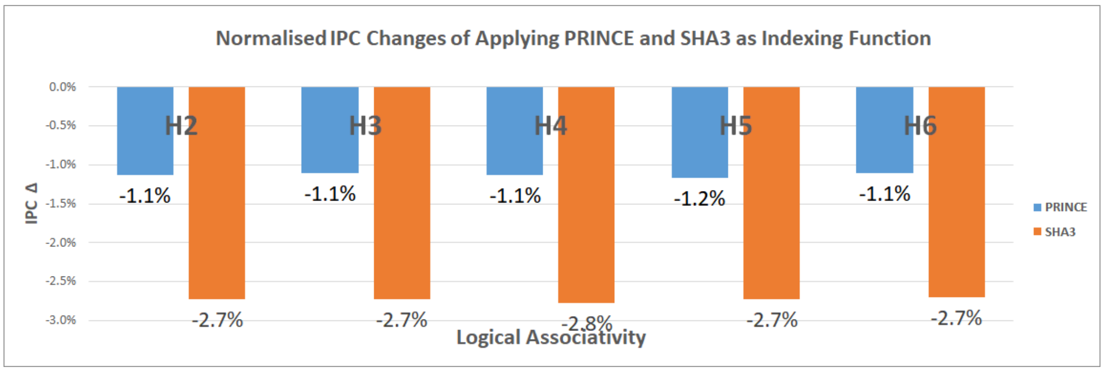


FIGURE 3.13: The comparison of normalised Instruction Per Cycle changes between applying PRINCE and SHA3 as indexing function.

The hit rate and the Instructions Per Cycle (IPC) of the CEASER-S cache with two different functions have been evaluated. Since we only focus on the indexing function itself, we disabled the rekeying of the CEASER-S in this experiment. Therefore, the results can only reflect the performance differences between the two functions with or without logical associativity.

The hit rates are shown in Figure 3.12. The baseline is the hit rate of the conventional cache with all three levels using the set-associative cache and LRU replacement policy.

Both caches have achieved lower hit rates than the baseline. When logical associativity is applied, the hit rates of both caches are lower than the baseline and are approximately the same. The reason both indexing functions provide caches with similar hit rates is that the ciphertext of PRINCE is divided into multiple parts as the index bits in multiple cache ways. Hence, in terms of reducing collisions, PRINCE loses the benefit of a one-to-one mapping. Furthermore, the hit rate degradation is also caused by the replacement policy in the last-level cache since the CEASER-S with full partitions is forced to have the Random Replacement policy. When increasing the logical associativity, both caches show lower hit rate degradations. Regarding the IPC shown in Figure 3.13, we found that the CEASER-S with PRINCE achieves a much lower IPC than with SHA-3. For example, when the logical associativity is 2, CEASER-S with PRINCE drops about 1.1% in IPC and CEASER-S with SHA3 drops about 2.7% in IPC. Combined with the hit rates results, we can see that although both caches have similar hit rates, the one with SHA3 still has worse performance than the one with PRINCE. This is due to the high latency of SHA3. Hence, the cache with SHA3 requires higher access latency. **As a result, a low-latency cryptographic function which is not vulnerable is a suitable choice for the indexing function.**

Since we did not find any other secure ciphers with lower latency than PRINCE, and other secure cache designs against contention-based attacks also utilise PRINCE as the cryptographic function, we decided to apply PRINCE as the SCv1 type indexing function for our later cache designs.

3.5 Summary

In this chapter, we introduced our new defence scheme for mitigating contention-based attacks on randomisation remapping cache. We name this Logical Associativity. Unlike some secure cache designs which provide strong protection but permanent performance degredation, this scheme allows the user to dynamically increase the logical associativity of the cache when the extra protection against contention-based attack is needed. Such configurations can significantly affect the attack complexity, which can help the cache achieve better performance before a new profiling method is found, and still be protected even under more advanced profiling methods of contention-based attack. We explained three properties of logical associativity: Reconfigurability, Overlapping, and Alienation. These properties are the key factors that logical associativity can mitigate contention-based attacks.

We introduced our security simulator, which is used to evaluate the success rate of a Prime+Probe attack on a randomisation remapping cache. We verified the simulator using theoretical results [57]. We showed logical associativity can reduce the effectiveness of the eviction set in an attack.

Finally, we evaluated two types of cryptographic functions for use with logical associativity. We compared a CEASER-S cache with SHA3 [24], which is a hash function, and a CEASER-S cache with PRINCE [19], which is an encryption cipher example, on the gem5 simulator. Both caches' re-keying functions have been disabled. For achieving better performance, we decided to choose PRINCE as the indexing function. Since it has a low latency, which is key to the cache. Another reason is that PRINCE has been used in other randomisation remapping cache designs [63, 16].

Chapter 4

CEASER-SH Cache

In this chapter, we introduce and evaluate the first cache design with logical associativity, namely the CEASER-SH cache. This cache is based on the CEASER-S [60] design. We have introduced logical associativity in chapter 3. In this chapter, we first explain the motivation of the CEASER-SH cache and then the threat model. After these, we discuss the implementation of the CEASER-SH cache. We especially focus on how logical associativity is achieved in the CEASER-SH cache and on the replacement victim selections. Later, we compare the simulation results of both the CEASER-S cache and CEASER-SH cache in terms of performance and security. Also, we will evaluate the hardware overhead and the power consumption of the CEASER-SH cache.

4.1 Motivation

As mentioned in Section 2.3, a cache timing side-channel attack is a powerful attack that could learn useful information from the cache line residency to accelerate the exploration of the victim's key or to build a covert channel and transmit the victim's sensitive data. Contention-based attacks, which do not require any shared data between different users, become the focus of secure cache designs. Although many countermeasure designs have been proposed in the past years, for example, the cache designs we explained in Section 2.4, the attack has also been developed and upgraded. The advanced attacks push cache protection to a higher level.

In Section 3.1, to provide flexibility in randomisation remapping caches for future advanced profiling methods of contention-based attacks, we proposed logical associativity. We then add this protection scheme to an existing randomisation remapping cache, CEASER-S [60]. The new cache design is named CEASER-SH.

By performing the simulation, we want to show that CEASER-SH, which has logical associativity, can achieve better performance than CEASER-S cache while providing equivalent or better security against contention-based attacks under a much stronger profiling method.

4.2 Threat Model

As in other works [60, 80], our cache defences are designed only to defeat contention-based attacks on the LLC. We do not consider other attacks. Both the victim and the attacker are non-privileged users. The cloud vendor (or the privileged user) can be trusted.

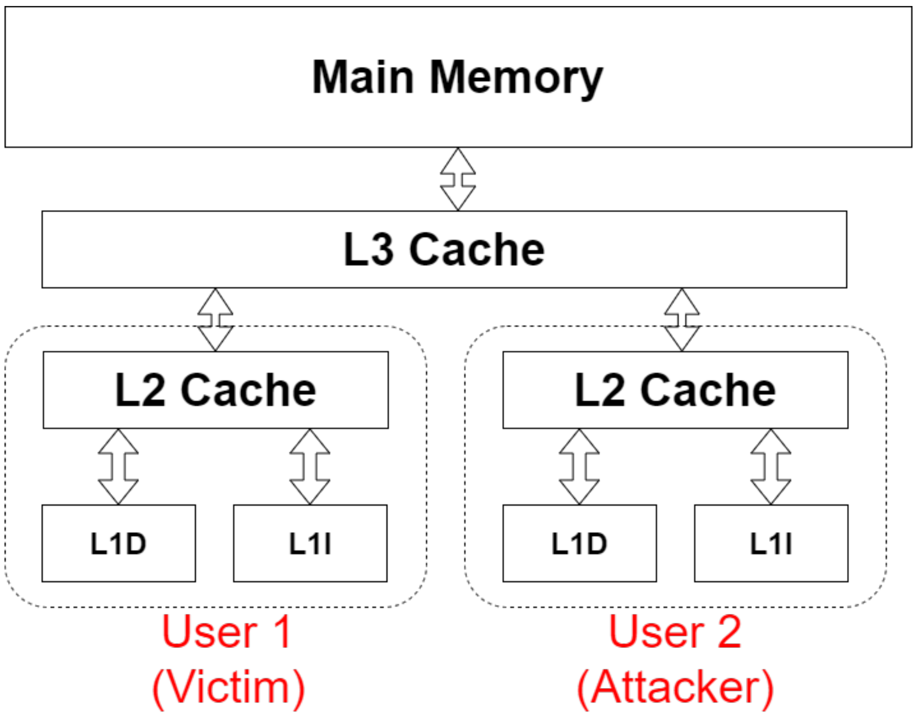


Figure 4.1: The memory hierarchy of the threat model.

We assume the attacker has the following abilities: The attacker has the ability to allocate themselves on the same machine as the victim. The attacker knows the LLC's cache configurations, such as the cache architecture, associativity, and cache size. Another assumption is that the attacker can trigger the victim process but cannot access the victim's addresses directly. This assumption aligns with previous work [60]. It is worth noting that even if we do not make this assumption, the attack can still be successfully implemented. The attacker just needs to wait for the victim themselves to start the process. Making this attacker-friendly assumption can accelerate the attack process so that, as a countermeasure designer, we can know the upper limit of the attack and provide sufficient protection to the cache.

We also assume the cache is configured as below: Each core has its L1 and L2 caches, which are not shared with other cores; only the LLC is shared between cores. This is a practical assumption which has been applied to modern commercial desktop or server CPUs, for example, an AMD 5995wx processor [3]. The memory hierarchy under such a threat model is shown in Figure 4.1.

In terms of the LLC (L3 cache), we make the same assumptions as the other research [57]. First, the attacker can control the input address of the indexing function, therefore the attacker does not need to consider the virtual and physical address mapping. Second, the attacker cannot obtain the key and the output of the indexing function. Finally, the attacker cannot affect the re-keying functionality of the cache.

4.3 CEASER-SH Architecture

CEASER-SH is based on CEASER-S cache [60] and ScatterCache [80]. Both caches apply a cryptographic function to encrypt the address of the cache line. This achieves randomised remapping. Hence the attacker can not deduce the mapping and the cache line residency. The major difference is that CEASER-SH applies logical associativity. Therefore, we first explain the logical associativity unit, which is used to calculate the next logical way within the logical set for the corresponding cache way. Then, we introduce another important part of the CEASER-SH cache, the replacement policy. CEASER-SH requires modifications on the replacement policy because it needs to select the replacement victim's cache set and cache way. Finally, we explain the structure of the CEASER-SH cache.

4.3.1 Logical Associativity Implementation

The logical associativity module is implemented as shown in Figure 4.2. The input of the logical associativity module is the encrypted index bits of the home set. The bits are sent to two multiplexers. Multiplexer M1 selects the input of the adder between the encrypted index bits and the output of the registers. The adder has the exact length of the encrypted index bits and keeps adding one to the input. The output of the adder is connected to the registers, which store the calculated index bits. Multiplexer M2 selects the output between the encrypted index bits of the home set and the output of the registers. The control signal *Bypass* only goes to logic 1 during the first clock cycle when the input is valid. *Inc* is the output of the comparator between $H_{counter}$ and H_{reg}, which are not shown in this figure. H_{reg} stores the logical associativity of the cache, and $H_{counter}$ counts which logical way is currently calculated. When the index bits of the last logical way have been calculated, *Inc* goes to logic 0, and $H_{counter}$ is reset for the next access. Based on the implementation, the logical associativity is performed by accessing multiple cache sets. If a cache hit is achieved, the corresponding set and way number are temporarily stored in the registers and used to handle the cache requests or responses after all searching is finished.

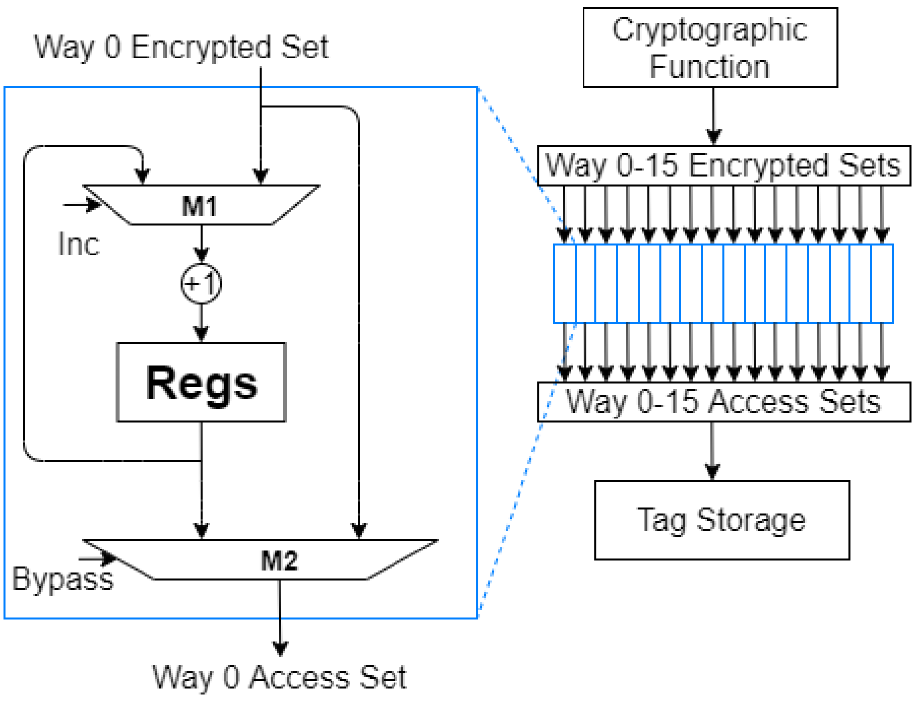


FIGURE 4.2: The implementation of logical associativity.

As we have mentioned in Section 2.1, there are two cache access modes for accessing tag and data: parallel and serial. This differs from how the cache handles the tag and data access. Since the tag is checked before the data is accessed for a serial-access cache, the data storage is only accessed once. Hence, increasing the logical associativity by one does not affect the data access. The randomisation remapping cache with logical associativity needs to add extra cycles for the additional tag accessing when the logical associativity is increased. An example of the access procedure is shown in Figure 4.3. The encrypted index bits of the access are $S1$ and $S2$ in ways 0 and 1, respectively. The logical associativity is 3. The tag is searched in $W0S1$, $W0S2$, $W0S3$, $W1S2$, $W1S3$ and $W1S0$. After the tag searching, the cache hit signal and the number of the set that stores the searched cache line are then sent to the data storage for access. For example, in Figure 4.3, the accessed cache line is stored in $W0S3$. Therefore, only $W0S3$ is accessed in the data storage.

However, for the parallel-access cache, the tag and data are accessed at the same time after the encrypted index bits are available. The data access normally requires a few cycles. Therefore, we cannot wait for the second data access until the first data access is finished. Nevertheless, we can initiate the data and tag searching of one set simultaneously. Because the tag comparison results are always available before the data, if the tag comparison leads to a miss, the next set number is directly sent to both the tag and data storage for the next search. In other words, the data output of the previous search is ignored. Another example is shown in Figure 4.4. The encrypted index bits of the access are still $S1$ and $S2$ in ways 0 and 1, respectively. The logical associativity is 3.

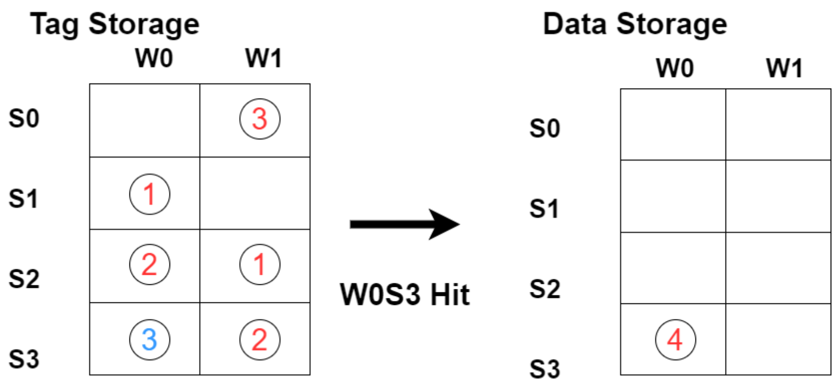


FIGURE 4.3: An example of tag-access while the logical associativity is applied and the cache is under the serial access mode. Circled number indicates the order of accesses. The cache hit is labelled in blue.

The cache first searches the data and tag of $W0S1$ and $W1S2$. After the tag comparison results in a cache miss, $W0S2$ and $W1S3$ are sent to both tag and data storage for the new access. Although the search of $W0S1$ and $W1S2$ is still propagating in the data storage and will be available after a few cycles, the cache will ignore its result. As a result, as the serial-access cache, the parallel-access cache also requires extra cycles for the additional tag access.

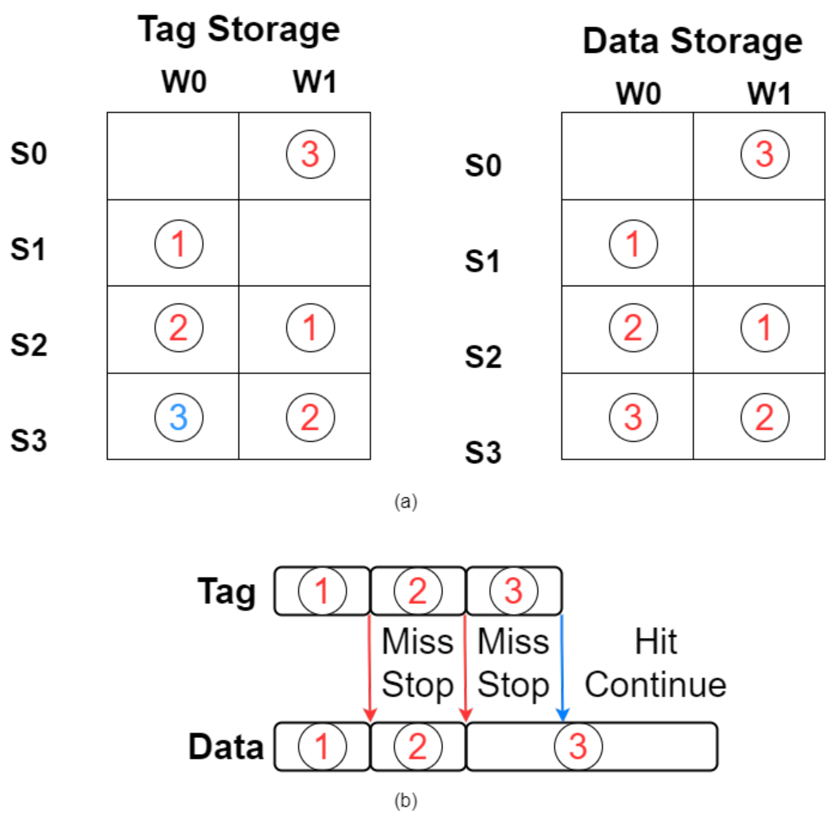


FIGURE 4.4: (a) An example of tag access while the logical associativity is applied and the cache is under the parallel access mode. Circled number indicates the order of accesses. The cache hit is labelled in blue. (b) The access timing of each round under the parallel access mode.

Overall, in terms of timing, logical associativity requires $(H-1)$ times the tag access latency no matter what cache access mode is applied.

4.3.2 Replacement Victim Selection

4.3.2.1 Replacement Policy

In a conventional cache and most randomisation remapping caches, the replacement policy determines the cache way of the targeted victim cache line when a cache line replacement occurs. Many replacement policies have been utilised to improve cache performance, such as LRU, LFU, and BIP [61]. However, other research [57] pointed out that an attacker could optimise the profiling stage when preparing a contention-based attack, for example, a cache with LRU. To construct an eviction set with the same eviction probability on the same randomisation remapping cache, for example on a CEASER-S cache, a cache with the random replacement policy always requires more cache accesses than a cache with the LRU replacement policy. This is because most replacement policies make victim decisions based on the previous access status of the cache lines within the cache set. An attacker may use this as a shortcut and occupy the entire targeted cache set very easily by accessing it in a specific order with a small number of cache lines. For example, an attack [44] uses a doubly-linked list to store the constructed eviction set. During the attack on the cache with the LRU replacement policy, the attacker only needs to access this list in reverse order so that no self-eviction can happen. Furthermore, this increases the effectiveness of the eviction set usage. A cache with a random replacement policy only selects the victim cache way based on the randomness it has been provided with. Therefore, the replacement becomes probabilistic. No matter the order in which the attacker accesses, there is always a high probability that self-eviction occurs. This will push the attacker to prepare more eviction cache lines before implementing the actual attack. Also, from the implementation perspective, since CEASER-SH is designed for full partitions, a random replacement policy is essential for partition selection.

4.3.2.2 Victim Set Selection

The replacement victim cache line is selected within the targeted cache set in a conventional cache. However, the victim could be any logical cache way with all logical cache sets in the CEASER-SH cache. Therefore, while selecting the victim cache line for replacement, both the cache set number and the cache way number need to be calculated.

As mentioned in Section 4.3.2.1, CEASER-SH applies a random replacement policy. Ideally, if the logical associativity is H, to perform the random selection in the cache set, the victim cache set number can be calculated as equation 4.1. However, the offset may require the modulo operation of a random number from the random number generator. The hardware implementation of the modulo operation requires multiple cycles for calculation [22]. The extra cycles could further increase the latency of the cache access.

$$Set_{victim} = Set_{Home} + Offset(0 \leq Offset \leq H - 1) \tag{4.1}$$

Therefore, instead of calculating the modulo of the random number, we directly use or add multiple numbers that are truncated from the generated random number to obtain the offset. In detail, when the logical associativity is 2^n, the offset could be directly obtained from the truncated number. When the logical associativity is not 2^n, the offset number needs to be added. For example, if $H = 3$, the offset could be 0, 1 or 2. Hence we truncate 2 one-bit numbers from the random number and add them together. Similarly, if $H = 7$, the offset is calculated by adding 2 two-bit numbers, etc. An offset calculation table is provided in Table 4.1.

Table 4.1: Examples of victim-set offset selection functions for different logical associativity.

LA Size	Victim-Set Offset Selection Functions
1	0
2	Random[0]
3	Random[0]+Random[15]
4	Random[1:0]
5	Random[1:0]+Random[15]
6	Random[1:0]+Random[15]+Random[14]
7	Random[1:0]+Random[15:14]
8	Random[2:0]

It is worth noting that the victim selection of both the cache set and cache way is started when the encrypted set is ready. Therefore, the addition operation of the offset calculation can be finished before the cache hit result becomes valid. Otherwise, in the worst case, the offset can be pre-calculated even before the access since the victim range is known and the victim selection is entirely random.

4.3.3 Implementation of CEASER-SH

As we discussed at the beginning of this section, CEASER-SH is based on the CEASER-S cache [60] and ScatterCache [80]. From our evaluation in Section 3.4, we have shown that the latency and the security of the function itself are the major considerations of the indexing function. In the CEASER-SH cache, we again use PRINCE [19] as the indexing function.

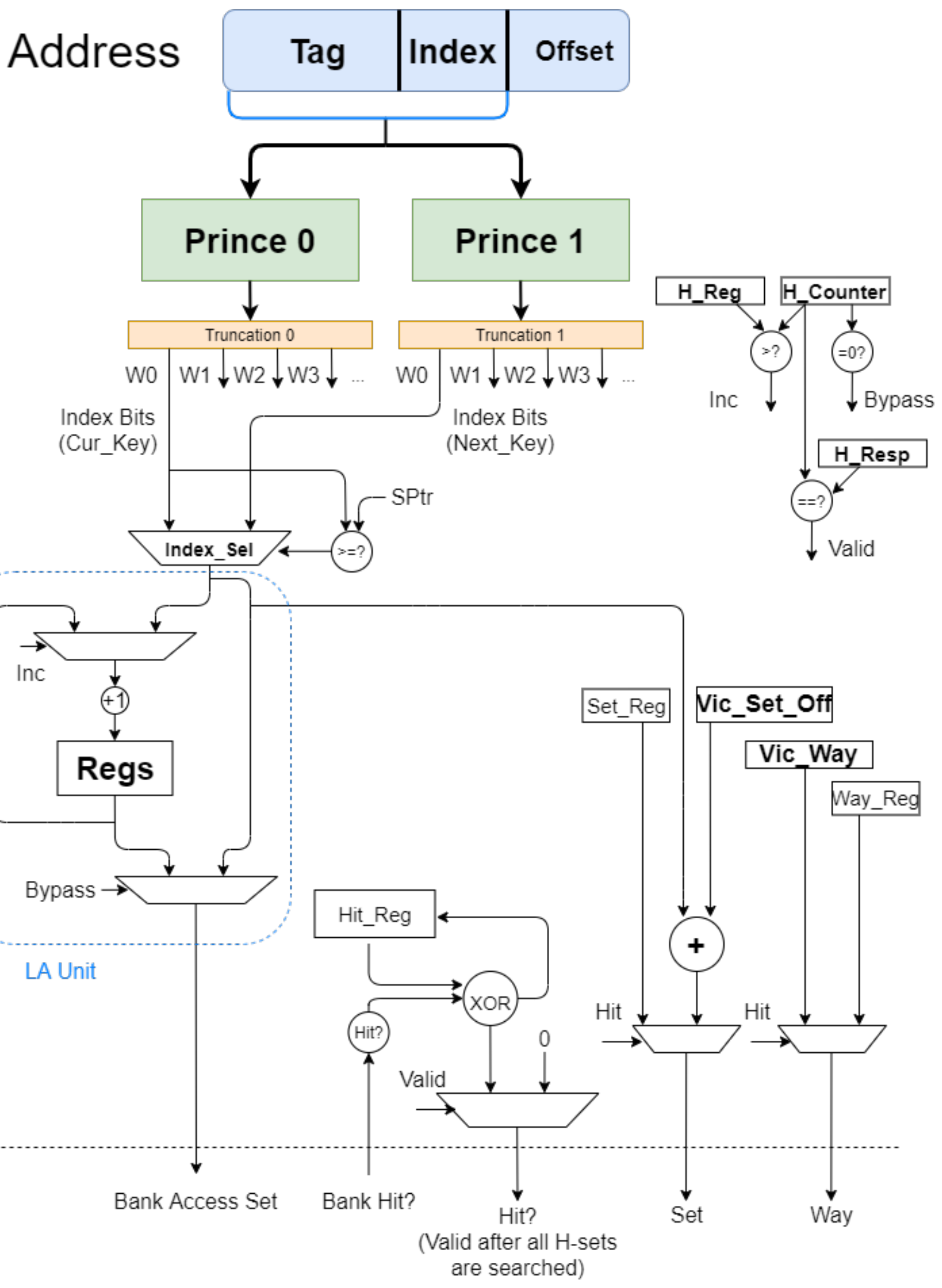


FIGURE 4.5: The structure of the CEASER-SH cache, including the cryptographic function, the logical associativity unit and the victim selection unit.

Figure 4.5 shows the implementation of the CEASER-SH cache controller. Similar to the CEASER-S cache [60], the CEASER-SH cache applies two indexing functions that have different keys. The SPtr register stores the current remapping set. After the comparison, the corresponding index bits are selected by the multiplexer **Index_Sel**. Later, the index bits are sent to the logical associativity unit of each cache way. By comparing the **H_Counter** and **H_Reg**, CEASER-SH either sends the bank access request to the next following cache set or directly sends the home set and prepares for the next cache access.

After the bank access requests are sent to the cache banks, the CEASER-SH cache waits for the response from the cache banks. Since the same cache line should only exist as one copy in the last-level cache, the CEASER-SH cache can receive at most one cache hit. The cache reads H responses and records the hit results until the last response arrives. The number of responses received is stored in **H_Resp**. The actual cache hit signal is sent after all H sets are accessed because this can prevent timing differences caused by the location of cache lines. This is achieved by a multiplexer controlled by the **Valid** signal. Hence, the attacker cannot infer which logical cache way the victim cache line belongs to.

If all responses are cache misses, the CEASER-SH cache selects the replacement victim as we described in Section 4.3.2, and then outputs the victim set and victim way as the set and way results. Otherwise, it outputs the set and way of the accessed cache line.

4.4 Security Evaluation

In chapter 3, we discussed how logical associativity could decrease the success rate of a contention-based attack when the eviction set has a fixed size. However, since some randomisation remapping caches, including CEASER-SH, apply the re-keying functionality, the constructed eviction set is only valid before the next re-keying happens. The re-keying period is measured by the number of cache access per full cache re-keying. For example, a re-keying period of $9N$ indicates the full cache is remapped (or re-keyed) after the cache is accessed $9N$ times, where N is the number of cache lines in the cache. For an $8MB$ cache with a $9N$ re-keying period, N is calculated as $8 \times 1024 \times 1024 \div 64 = 131,072$ (with 64 Bytes cache line size), and the cache is fully remapped after $1,179,648$ cache accesses.

Therefore, in the attack on a randomisation remapping cache with a re-keying functionality, the attacker may not be able to construct an eviction set with the expected size. Besides the re-keying period, the cache configurations could also affect the size of the constructed eviction set.

As discussed in Section 3.3, we implemented our own security evaluation simulator in C++. To evaluate our CEASER-SH cache, we simulate its security in two steps. In the first step, the simulator repeats Prime+Prune+Probe (PPP) until the given re-keying period is reached. During each round of PPP, the address of the evicted cache line is added to the eviction set as a member. All the addresses of the eviction set members are stored. In the second step, the simulator takes the eviction set constructed within the re-keying period to implement the Prime+Probe attack. Each eviction set is tested in $100k$ rounds, and the overall success rate of the attacks is recorded. Hence, the success rate represents the success rate of the contention-based attack under the corresponding re-keying period. Since we apply the identical parameters in the PPP profiling and assumptions as the other research [57], the evaluation results of the CEASER-S from that research can directly be used to compare with our CEASER-SH cache.

Although we only tested the Prime+Probe attack, it also indicates the security against an Evict+Time attack. As discussed in Section 2.3.3, the root of both attacks is the same. However, the Prime+Probe attack achieves higher fidelity and is easier to implement than the Evict+Time attack. Therefore, we only use the Prime+Probe attack to evaluate CEASER-SH cache security. This is aligned with other secure cache design evaluations [49].

TABLE 4.2: The success rates of the Prime+Probe attack on CEASER-SH cache with Optimal K (Opt K) values under different re-keying periods (RKP).

H	1	2	3	4	5
RKP \ Opt K	16	16	32	64	128
9	1.00%	0.35%	0.21%	0.20%	0.14%
10	1.17%	0.41%	0.27%	0.23%	0.18%
15	1.78%	0.65%	0.40%	0.35%	0.26%
20	2.40%	0.90%	0.56%	0.47%	0.34%
22	2.80%	1.03%	0.61%	0.52%	0.39%
25	3.14%	1.14%	0.68%	0.58%	0.44%
29	3.68%	1.34%	0.79%	0.68%	0.52%
30	3.75%	1.41%	0.83%	0.72%	0.54%
35	4.45%	1.64%	0.98%	0.83%	0.63%
40	5.05%	1.86%	1.12%	0.93%	0.71%
45	5.65%	2.12%	1.29%	1.07%	0.78%
50	6.27%	2.28%	1.41%	1.15%	0.88%
75	9.63%	3.59%	2.28%	1.78%	1.38%
100	12.26%	4.76%	2.92%	2.26%	1.78%
200	23.15%	9.39%	5.78%	4.53%	3.52%
1000	73.34%	39.39%	26.26%	20.65%	16.49%

For the simulation, we set the following parameters. In terms of the cache configurations, the cache size is set to $8MB$, the associativity is 16 ways. The cache replacement policy is the random replacement policy. For the profiling parameters, the aggressive pruning starts after the fifth round of pruning. This parameter is suggested elsewhere [57]. In the attack, we make the same assumption as Section 3.3.1: we assume there is only one targeted victim cache line. An attacker can use all of the time for profiling before re-keying. The attacker could trigger the access of the targeted cache line itself. After triggering the victim cache line, the attacker can somehow evict it from the cache, e.g.

flush the entire cache. These assumptions allow the attacker to find more members of a PCE set within the re-keying period and achieve the theoretically highest success rate in a Prime+Probe attack. In other words, these assumptions are friendly to the attacker. Under such conditions, CEASER-S with full partition may need to reduce the re-keying period to about $9N$ to maintain 2-year security against contention-based attacks [57].

It is worth noting that although we apply the completely same assumptions and attack method as the other research [57], the results we measure are slightly different. In our evaluations, instead of measuring the success rate of constructing a PCE set with a 95% eviction probability within a fixed number of cache accesses, we measure the eviction success rate (or the eviction probability) of a constructed PCE set within a fixed number of cache accesses. There are two reasons for applying such a measure in our experiment. First, the success rate of constructing a PCE set with a 95% eviction probability within $22N$ is too low to measure in an experiment, for example, the success rate within $9N$ re-keying period is 2^{-32}. It is impractical to construct 2^{32} eviction sets and measure all of their eviction probabilities. Second, the selection of a 95% eviction probability may not be necessary, an attacker may decide to bear with the eviction set with a lower eviction probability. Nevertheless, both measures reflect the difficulty of finding eviction cache lines on a randomised remapping cache. If the average P_E of the constructed eviction sets on a cache configuration is higher, which indicates the contentions are easier to find, the probability of constructing an eviction set with $P_E = 95\%$ on a cache configuration must be higher.

Using the simulator, we performed security evaluations of CEASER-SH with different logical associativity. Also, for comparison, we simulated the security of CEASER-S with full partitions. We built 10 PCE sets under each configuration (300 PCE sets total). Each PCE set is tested $100k$ times in Prime+Probe attacks; hence each configuration was tested with one million Prime+Probe attacks. We set different K values for different configurations. The K value, which is the number of initial candidate cache lines, has been explained in 2.6.1.2. The simulation results in Table 4.2 shows the highest success rates of the Prime+Probe attack on the CEASER-SH cache with different logical associativity under different re-keying periods (RKP). The optimal K (OPT K) is the K value that achieves the highest sccess rate in the corresponding logical associativity. Based on the results in Table 4.2, we summarise the results in Table 4.2 into Table 4.3, which shows the equivalent security level between CEASER-S and CEASER-SH with different logical associativity. The equivalent security level indicates that the CEASER-SH cache with a re-keying period could provide at least the same level of security against contention-based attacks as the CEASER-S cache with another re-keying period. The re-keying period values in the table are equivalent to the re-keying periods of CEASER-S. A smaller re-keying period indicates better protection against contention-based attacks. For example, CEASER-SH with logical associativity 2 and a 100N re-keying period has equivalent security to CEASER-S with a re-keying period of 38N. This result is bold in

Table 4.3. Alternatively, increasing logical associativity while maintaining the re-keying period can improve security against contention-based attacks.

Table 4.3: The re-keying periods in the table are multiples of N, where N is the number of cache lines. These values are equivalent to the re-keying periods of CEASER-S. RKP (N) is the re-keying period of CEASER-SH, which is the number of accesses per re-keying. LA is the logical associativity.

	RKP (N)					
LA	**22**	29	40	50	75	**100**
2	**9**	12	16	19	30	**38**
3	< 9	< 9	10	13	19	23
4	< 9	< 9	< 9	10	15	19
5	< 9	< 9	< 9	< 9	12	15

However, the re-keying periods and cache configurations not only have an impact on the security against contention-based attacks but also on the performance of the cache. We will next compare the performance of both caches.

4.5 Performance Evaluation

4.5.1 Simulation Setup

In this section, we evaluate the performance of the CEASER-SH cache. For comparison, we implemented both CEASER-S and CEASER-SH on the gem5 simulator [15] for the performance evaluations. Both caches were set as full partitions. In the rest of this thesis, when we mention CEASER-S or CEASER-SH, we specifically refer to caches with full partitions. In other words, the number of cache ways equals the number of cache partitions, so each cache way has its own unique cache mapping.

We used the ARM O3 CPU model in the gem5 simulator [15] running at a clock speed of 3GHz. The cache configurations included three levels: L1 cache with a size of 32KB and an associativity of 4, L2 cache with a size of 512KB and an associativity of 8, and L3 cache with a size of 8MB and an associativity of 16. Both L1 and L2 caches are set-associative and not shared between cores. In our simulations, we took into account the delay of PRINCE, which is 3 clock cycles, which aligns with Mirage [63]. We also considered the impact of logical associativity on cache latency. This is further discussed in Section 4.5.2. For comparison purposes, we also tested a conventional set-associative LLC with a BIP replacement policy, which serves as the baseline in our evaluation [80].

For the simulation workloads, we applied two benchmarks used in some other security cache design research [80, 49]. The first is GAP Benchmark Suite [10], and the other is PARSEC Benchmark Suite [14]. We used all programs in GAP benchmarks. We applied $-g16 - k16$ as the parameter, where g is the scale and k is the degree of the graph generation. This aligns with ScatterCache[80]. For the PARSEC Benchmark Suite, we used 6 programs including blackscholes, canneal, fluid, freqmine, streamcluster and swaptions. We did not use MiBench [29], since the workload is too small and the performance variations on an 8MB LLC are negligible. Although some related research used SPEC2006/2017 as benchmarks, we did not use them because they are not open-source benchmarks.

4.5.2 Simulation Results

We simulated both caches with different re-keying periods. As discussed in Section 4.5.1, we considered two different situations. The results in section 4.5.2.1 are the simulation results based on our synthesised logic with Nangate45nm PDK, which shows CEASER-SH requires one clock cycle per logical associativity increment. For some cache implementations, the tag look-up time could be more than one clock cycle. Additionally, we evaluated the CEASER-SH cache which requires 2 clock cycles per logical associativity increment in Section 4.5.2.2.

4.5.2.1 Performance With One Extra Clock Cycle

The performance simulation results of the CEASER-SH cache with one clock cycle per logical associativity increment are shown in Figure 4.6 and Figure 4.7. In this case, the cache latency can be calculated as the sum of the base cache latency, the encryption cipher latency, and the logical associativity increment. For example, when the logical associativity is increased to 3 from 1 (no logical associativity), the cache latency is set to 45 clock cycles, including 40 cycles as the base cache latency, 3 cycles as the encryption cipher latency, and 2 cycles due to the logical associativity increasing. Based on this latency, the Cycle Per Instructions (CPI) and the Misses Per Kilo Instructions (MPKI) of CEASER-SH with a different re-keying period are shown in Figure 4.6 and Figure 4.7.

The MPKI and CPI are shown for logical associativities ranging from 1 to 5 (H1-H5). The re-keying period is measured by the number of cache accesses, where N indicates the size of the cache. For example, a re-keying period of 40N in a cache with 131072 cache lines (8MB if the cache line size is 64B) is $5,242,880$ cache accesses. Both figures show the normalised results based on a set-associative cache with same size and the BIP replacement policy [61].

As shown in Figure 4.6 and Figure 4.7, the MPKI and CPI increase dramatically when the re-keying period is less than 22N. When the re-keying period decreases, the cache is

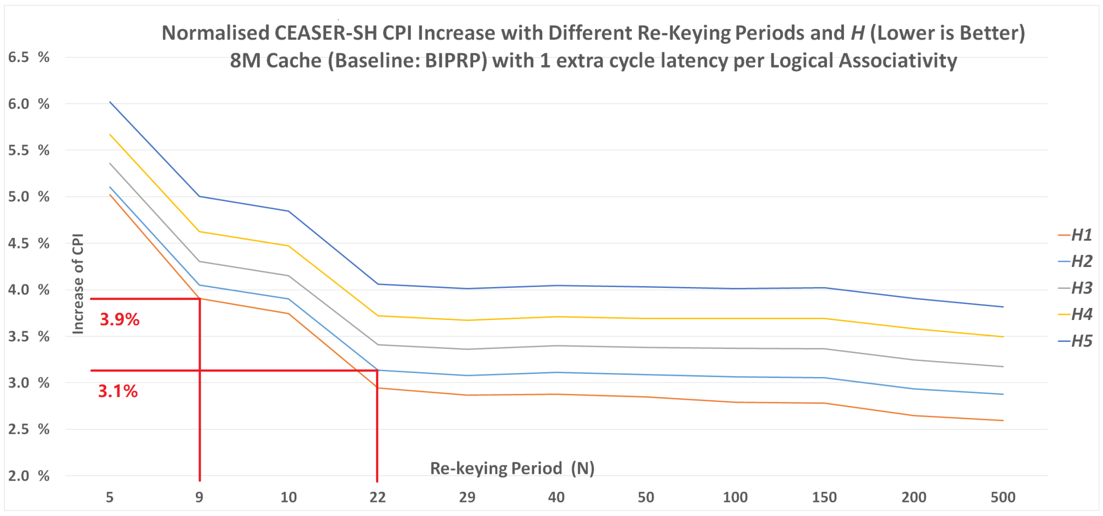


FIGURE 4.6: The normalised CPI increase of CEASER-SH with one clock cycle tag access time.

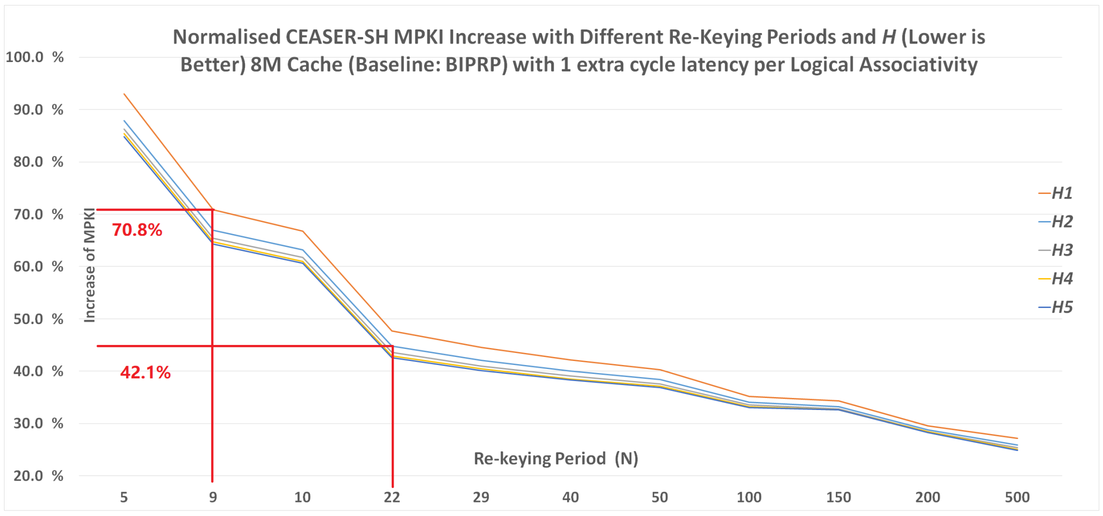


FIGURE 4.7: The normalised MPKI increaseof CEASER-SH with one clock cycle tag access time.

remapped more frequently, which leads to more cache misses. Hence, the CPI increases sharply. The miss rate shows very similar results as the MPKI, therefore we do not show the miss rate results here. Most workloads follow this trend except Streamcluster, which has lower CPI and MPKI when the re-keying period is low. The reason appears to be that Streamcluster does not reuse most of its data since its MPKI under all cache configurations is close to 1. From Table 4.3 and the cache performance, we found that increasing the logical associativity can both maintain security against contention-based attacks and outperform CEASER-S under some configurations. Those configurations are shown in red in Table 4.3. For example, CEASER-SH with a re-keying period of $22N$ and logical associativity of 2 has better security against contention-based attacks than CEASER-S (CEASER-SH with associativity of 1) with a re-keying period of $9N$. Both configurations, CPI and MPKI are labelled in Figure 4.6 and Figure 4.7. The results show that the performance penalty of CEASER-SH is a 3.9% increase in CPI and 70.8% increase in MPKI. The performance penalty of CEASER-S achieves a 3.1% increase in CPI and 42.1% increase in MPKI. As a comparison, CEASER-SH achieves a 0.8% lower CPI, and a 28.8% lower MPKI compared to CEASER-S. Note that the comparison of MPKI and CPI is relative to the baseline configuration mentioned in the previous paragraph. Overall, CEASER-SH can provide the same or even better protection against contention-based attacks while performing better when an advanced profiling method is applied in an attack and pushes the re-keying period of the CEASER-S cache under $50N$ accesses per time.

4.5.2.2 Performance With Two Extra Clock Cycles

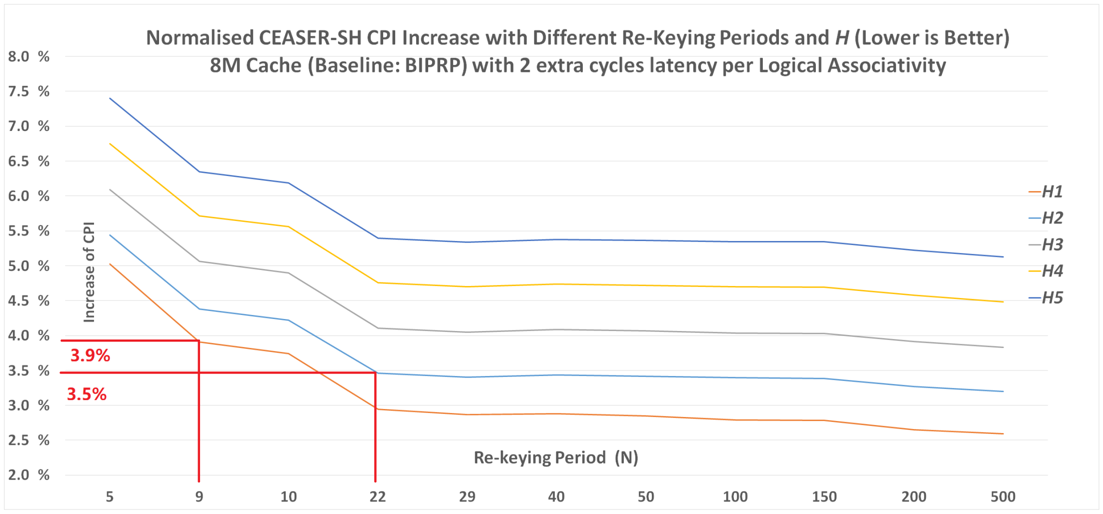


FIGURE 4.8: The normalised CPI increase of CEASER-SH with two clock cycles tag access time.

We also evaluate the performance of the CEASER-SH cache when each tag access needs two clock cycles. All simulation configurations are identical to Section 4.5.2.1 except for the cache access latency. The CPI and MPKI are shown in Figure 4.8 and Figure 4.9. Compared to the performance of the CEASER-SH with 1 clock cycle latency per logical associativity, which was discussed in section 4.5.2.1, the cache with 2 clock cycles showed a major difference in CPI. The impacts on the MPKI are negligible. Since each cache access requires a longer time, this slows down the overall L3 cache access. As a result, the overall time to finish an instruction is longer.

Nevertheless, from the simulation results shown in Figure 4.8 and Figure 4.9, the CEASER-SH cache can still achieve better performance under such a high latency while providing equivalent or better security against contention-based attacks. For example,

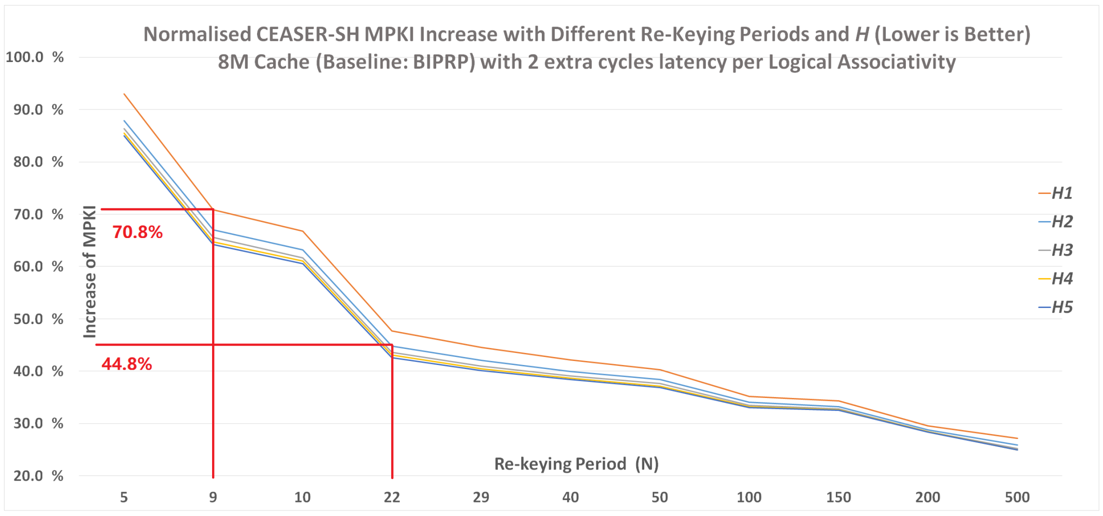


Figure 4.9: The normalised MPKI increaseof CEASER-SH with two clock cycle tag access time.

CEASER-SH cache with a re-keying period of $22N$ and logical associativity of 2 achieves about a 0.5% lower CPI, and a 26.1% lower MPKI compared to CEASER-S with a re-keying period of $9N$.

4.6 Hardware and Power Overhead Estimation

Providing concrete values for hardware and power overheads is always hard. This is due to many reasons. First, we cannot find an existing open-source L3 cache design. Certainly, building an L3 cache from scratch is impractical for the project. The only available and most suitable cache design we found is the SiFive inclusive cache [23]. This is an L2 cache which is used in the Rocket-Chip [6] and BOOM Chip [90]. Both of them are well-known open-source RISC-V core implementations. Second, we cannot access the most advanced technology library used to fabricate commercial processors. Therefore, we can only estimate the area and the power overhead of the CEASER-SH cache like other secure cache designs [80, 63]. As a result, similar to Mirage [63], we first estimate the storage and power overhead based on CACTI 6.0 [50], and then estimate the area and power overhead of the additional control logic with the open-source technology library Nangate45nm PDK [69].

4.6.1 Hardware Overhead Evaluation

4.6.1.1 Storage Overhead

The storage overhead of the CEASER-SH is from the additional tag bits, which is the same as the ScatterCache [80]. In a conventional cache, the index bits are not stored as part of the tag. The address of the cache line could be recovered directly from the tag stored and the set number of the cache. However, the set number is the encrypted index bits. Therefore, the original index bits must be stored as part of the tag, or the encrypted index bits must be decrypted before recovering the address. Like other randomisation remapping caches, we choose the former solution because the latter solution takes another 3 clock cycles due to the decryption.

We set the physical address space as 46-bit, aligned with Mirage [63]. We set the cache line size as 64B, the L3 cache as 8MB, and 16 ways. The storage comparison between the CEASER-SH and a conventional cache is shown in table 4.4. In the conventional cache, 13 bits are used as index bits, and 6 bits are offset bits. Hence, the length of the tag bits of the conventional cache is 27 bits. Including 2 status bits, each cache line requires a 29-bit tag entry in a conventional cache. There are 131072 cache lines in the cache. Therefore the size of tag storage should be 3,801,088 bits which is 464KB. The CEASER-SH cache requires storing the full tag, which is 40 bits. Including the status bits, each cache line needs a 42-bits tag entry. The total size of tag storage is 672KB. Since we do not need to modify the data storage, the data storage of either the conventional cache or the CEASER-SH requires 8192 kB. The total storage sizes of both caches are 8656KB and 8864KB, respectively. Compared to the conventional cache, CEASER-SH only requires 2.4 % extra storage, which is minor. Compared to Mirage [63], which requires about 20% more storage overhead, the increase is negligible.

We use CACTI 6.0 [50] to evaluate the area increase of the additional storage. We also set the technology size as 45nm, the same as our technology library used to synthesise the control logic of the CEASER-SH cache. The increased 208 kB storage enlarges the data and tag array by $2.88mm^2$. Compared to the area of the entire storage, the area is increased by about 5.1%, which is practical.

4.6.1.2 Control Logic And Overall Hardware Overhead

Like other randomisation remapping caches, CEASER-SH requires cryptographic functions and re-keying to compute the encrypted index bits for cache line search. As mentioned, we use PRINCE as our indexing function. Since the cache needs to compute the corresponding index bits based on both the current key and the next key, two PRINCE functions are needed. After adding the logical associativity module and victim selection module based on SiFive inclusive cache [23], we synthesised the entire control logic with the Nangate45nm PDK [69]. The extra logic requires about $0.025mm^2$, which is 31329 GE. (The area of a NAND2X1 gate in Nangate45nm is $0.798um^2$.)

Since both CACTI 6.0 and our synthesised results are based on 45nm technology, we estimate the total area of both storage and control logic increase is less than $2.9mm^2$. Compared to a conventional cache, the area is increased by 3.4%. Such area change is minor compared to the entire die of the processor. If the CEASER-SH design is synthesised with state-of-the-art technology, the design is certainly practical in terms of hardware overhead. For example, the AMD Zen2 processor has a $74mm^2$ 7nm CPU compute die [51].

We did not consider the area reduction due to the change of replacement policy. The random replacement policy is easier to build than other replacement policies since it does not need to store any cache line usage information. Hence, using the Random replacement policy could compensate for some area growth.

Table 4.4: The overhead comparison of a conventional cache and a CEASER-SH cache with the same configurations.

	Cache Size 8M	**Baseline** Set Associative	**CEASER-SH**
Storage Overhead	Tag Bits	27	27
	Status Bits	2	2
	Set Bits	-	13
	Total Tag Entry Bits	29	42
	Tag Store Size	464kB (100%)	672kB (144.8%)
	Total Storage	8656kB (100%)	8864kB **(102.4%)**
	Area	55.6 (100%)	58.4 **(105.1%)**
Additional Logic Overhead	**Area**	-	31329 GE $0.025mm^2$
Overall	Area	$85.84mm^2$	$88.72mm^2$ (+3.4%) ($+2.88mm^2$)

4.6.2 Power Overhead Evaluation

Similar to the hardware overhead evaluation, we analyse the power consumption in two parts: cache storage and extra logic. Based on the results of CACTI 6.0 [50], we show that the conventional cache with the same configurations shown in Section 4.6.1 has $2.01W$ leakage power. The dynamic power is $0.306nJ$ per access. As we assumed in the gem5 simulation, the clock frequency we applied is 3GHz, and the conventional L3 latency is 40 clock cycles. Based on the performance evaluation results, we estimated the dynamic power is about $0.02W$. Hence the total power consumption of the conventional cache is $2.03W$. We notice that the leakage power of the storage dominates the power usage of the entire cache.

The CEASER-SH has another 0.06W leakage power for the addition of the tag storage, and the dynamic power for just accessing both the tag and data array is $0.331nJ$ per access. Since the different logical associativity can lead to different dynamic power consumption, we estimate the power estimation when the logical associativity is 4. While the extra control logic is synthesised with the maximum frequency by Nangate45nm PDK, the total power consumption of a CEASER-SH, which works with $H = 4$, requires about $2.154W$ power. As a comparison, compare to a conventional cache, Mirage [63] increases the power by about 21%, which is triple that of CEASER-SH cache.

Compared to the conventional cache, the required extra power is $0.124W$. A modern desktop CPU, for example, AMD5995wx, needs $280W$ power under maximum workloads [3]. Compared to an entire CPU, the power increase in CEASER-SH is negligible.

Nevertheless, the power of the CEASER-SH cache can still be optimised. For example, when the logical associativity is 4, the CEASER-SH ideally needs to search all 4 cache sets. In the real implementation, the designer could skip the search after a cache hit is reached since there could only be no cache line or just one cache line that matches during the entire search. However, skipping the search does not mean the cache is ready for the next cache access. The cache still needs to be idle, and pretends it is still searching the cache line, because maintaining the same cache access latency is necessary. Otherwise, the timing difference between a cache hit and miss might be abused as a vulnerability. This modification does not have any changes to the timing. As a result, such optimisation will not decrease the protection against contention-based attacks. Since we do not consider any other side-channel attacks, we do not consider if this optimisation could become vulnerable to other side-channel attacks, such as power or radiation attacks.

4.7 Summary

In this chapter, we proposed the CEASER-SH cache to mitigate cache contention-based attacks. Compared to the existing randomisation remapping cache designs, CEASER-SH provides more flexibility in balancing the performance of the cache and the security against the contention-based attack. The main idea of the cache is applying logical associativity that we proposed in chapter 3 to the existing, CEASER-S cache [60]. The logical associativity allows the cache line to be allocated in its original mapped set and the $H - 1$ sets. Due to two properties of logical associativity, reconfigurability and overlapping, CEASER-SH can provide better performance while having equivalent or better security than the CEASER-S cache under a much stronger contention-based attack. For example, based on our simulation results, a CEASER-SH cache with logical associativity 2 achieves about a 0.8% lower CPI, and a 28.8% lower MPKI compared to CEASER-S while providing equivalent security against contention-based attacks.

For the overhead estimation, we used CACTI6.0 to evaluate the power and hardware overhead of the additional storage required in the CEASER-SH cache design. The additional control logic to perform the logical associativity is based on the SiFive inclusive cache. The control logic is synthesised with the Nangate45nmPDK. From the results, we estimate the overall area addition is about $2.9mm^2$, which is increased by 3.4%, including the storage overhead. The power overhead of the CEASER-SH is also evaluated in the same way. Since the dynamic power of the CEASER-SH cache is related to the logical associativity, we show a CEASER-SH cache with logical associativity as 4 requires $0.124W$ more power than the conventional cache. Compared to the entire CPU, the power and area overhead are acceptable. Compared with Mirage [63], CEASER-SH has much less hardware and power overhead.

Chapter 5

SEA Cache

In this chapter, we propose another randomisation remapping cache called the SEA cache. This cache focus on applying the third property of logical associativity, which is alienation. The property has been discussed in Section 3.2.4. We first explain the motivation for the SEA cache. Then, we discuss the SEA cache architecture and its implementation. Later, we evaluate the SEA cache in terms of performance, security against contention-based attacks, hardware overhead, and power overhead.

5.1 Motivation

Using the CEASER-SH cache, we have shown that logical associativity cqn be used to counter contention-based attacks in the last-level cache. The flexibility of associativity allows the cloud vendor or the privileged users to balance the performance and the security themselves. However, the logical associativity in CEASER-SH is modified globally. In other words, increasing the logical associativity affects all users who share the machine. Even if some users do not run any security-sensitive programs, they still need to suffer higher LLC latency if somebody else requires high logical associativity.

Accessing multiple cache sets in the CEASER-SH cache also increases the access latency. Reducing the access latency can enhance the cache performance. Parallel accessing to different cache sets within each cache way, which has been applied in PhantomCache [73], can significantly improve the access latency. Hence, the cache performance is enhanced.

Based on these ideas, we propose another cache design based on CEASER-SH. We name the cache the Scattered Elastic-Associativity Cache (SEA cache). Such a cache allows logical associativity changes locally and distinctly between users. Also, the SEA cache supports both parallel and serial bank access. The threat model of the SEA cache is identical to that of CEASER-SH, which was discussed in 4.2. This design makes the randomisation remapping LLC design more feasible.

5.2 SEA Cache Architecture

5.2.1 Introduction to the SEA cache

The general design idea for the SEA cache is similar to that of the CEASER-SH cache in chapter 4, which applies logical associativity to a randomisation remapping cache. This section will focus on the major architectural difference between the SEA cache and the CEASER-SH cache. As mentioned in section 5.1, the SEA cache is proposed to reduce the unnecessary latency for some users who do not need extra protection, while another user raises the logical associativity to enhance their security against contention-based attacks. To achieve this, we need to allow different users to have different logical associativity so that the users can make their own decision between security and performance, instead of the vendor.

This is a significant change to the existing randomisation remapping cache. Most of these cache designs [60, 80], defeat contention-based attacks by setting a re-keying period. The constructed eviction set is only valid before the next re-keying happens. Therefore, the attacker must finish the profiling and the attack within the period. The re-keying applies globally and affects all users because such a parameter is restricted to each physical cache set and has no relation to the user or the process. The logical associativity provides wider flexibility to unprivileged users. Hence, they can decide their own trade-off between performance and the security of the cache.

Since the L2 cache is not shared between cores and each core is only allocated to one user at run time, the LLC only needs to know what logical associativity is applied to each core when one of the L2 caches requests data from LLC.

Since the SEA cache needs to access multiple cache-sets simultaneously, the hardware implementation of logical associativity need to be modified. Based on the PhantomCache [73] design, we enable parallel access to multiple cache-banks for performing multiple cache-set simultaneous access. Nevertheless, in the SEA cache, the number of cache sets that the cache accesses may not be a multiple of the cache bank numbers. For example, the logical associativity of users A and B are 1 and 3, respectively. If the cache can only issue 8 parallel access to the 8 cache banks, all the requests from A and B must be performed to all 8 banks. This results in wasted power. To optimise the SEA cache, we make an enhancement to make fewer parallel accesses to the banks possible. Hence, in the example, the cache will only send the requests to 1 bank or 3 banks, depending on the origin of the access. We explain parallel access to multi-cache-banks enhancement in Section 5.2.2.2. We will introduce the modifications to logical associativity and how these affect the cache timing and power consumption.

To provide a finer balance between the performance and the security against contention-based attacks, we also extended the SEA cache to the process-level protection by bringing the modified Security Domain Identifier (SDID) bits into the pages. The SDID was first introduced in Newcache as a Trust Domain Identifier (TDID) [43], which was used to provide different cache mappings to processes within different domains. In our SEA cache, having the SDID can allow the hypervisor to distinguish the logical associativity of each page. There are two benefits: First, only security-sensitive processes need to be protected by higher logical associativity. This improves the overall performance and may reduce power consumption due to fewer cache set accesses. Second, this enables data sharing between users who share the same logical associativity. The data sharing allows users to share the same libraries so that the memory can save some space for not holding multiple copies of the same library for different users.

5.2.2 SEA Implementation

As discussed in Section 5.2.1, compared to the CEASER-SH cache, the SEA cache has the following features: 1. Alienation: Different users are allowed different logical associativity. 2. Parallel Bank Access: When logical associativity is higher than 1, the SEA cache automatically sends requests to multiple cache banks in parallel and achieve low latency access. 3. Smart Bank Access: For different logical associativity, the SEA cache can issue a suitable number of requests to each cache bank, which balances the power consumption of the cache. To meet these goals, we explain the implementation of the SEA cache.

5.2.2.1 Alienation Implementation

To allow users to have different logical associativity, each user should have a dedicated register to store its logical associativity setting. In the worst case, each user on the server is allocated just one core. In this case, the number of registers must be at least the same as the number of cores in the CPU. A modern server processor, such as AMD Threadripper 5995wx, can have 64 cores [3], which will require 64 registers for storage of each user's logical associativity. Nevertheless, when a user is allocated to more than one core, their setting only needs to be stored in just one register. When a request is sent to the SEA cache, the user id is parsed, and the corresponding logical associativity is used for the cache line access.

5.2.2.2 Parallel Bank Access

Based on CEASER-SH, we found that the computation of the logical associativity should take one clock cycle. However, when a new request is sent to the CEASER-SH cache, the ciphertext of PRINCE, which is the home set of the accessed cache line, can bypass the logical associativity module and be sent to the cache bank as a request. In the meantime, the following cache set can be calculated in the logical associativity module. After taking an extra $H - 1$ clock cycles, all logical cache ways within all logical cache sets with size H are sent to cache banks as requests. For a recap, the each of the cache line of the logical cache set is called logical cache ways, which we have explained in Section 3.2.1. An example of the request order of a CEASER-SH cache with 2 ways and $H = 4$ is shown in Figure 5.1(a). The home set of the access is $S1W0$ and $S2W1$. CEASER-SH cache will issue the first round requests to $S1W0$ and $S2W1$, the second round requests to $S2W0$ and $S3W1$, then $S3W0$ and $S4W1$, and last $S4W0$ and $S5W1$. The entire access takes 3 more clock cycles than for ScatterCache [80] with the same configurations.

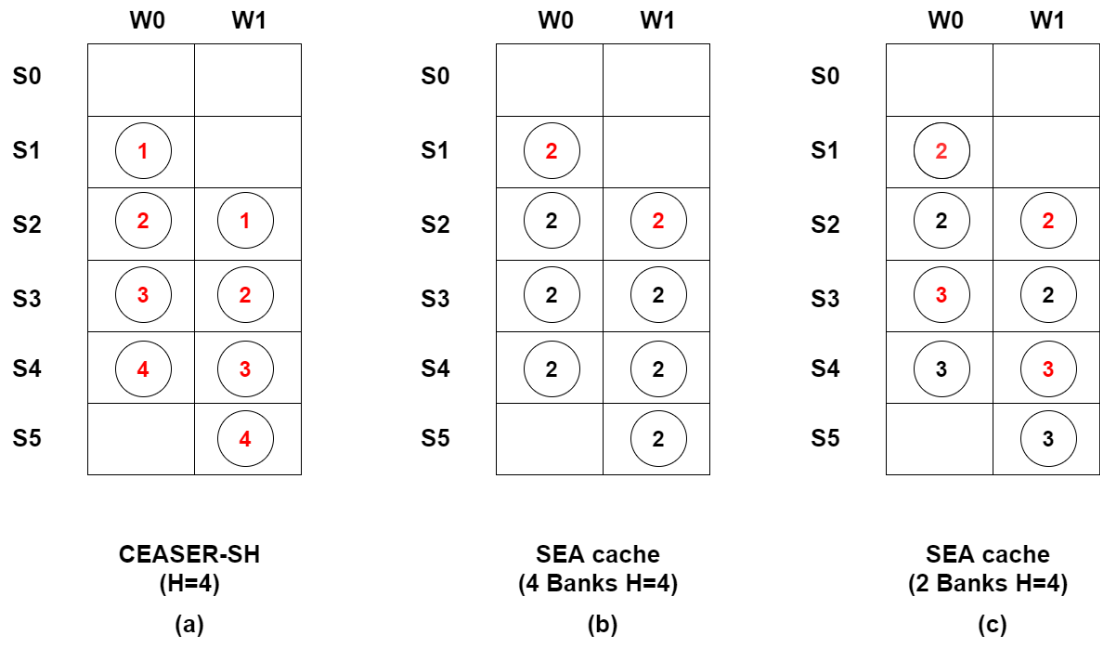


FIGURE 5.1: The request orders of CEASER-SH ($Way = 2$, $H = 4$) and SEA cache with the different numbers of cache banks. The numbers indicate the clock cycle in which a request to access such a cache line is sent to the banks after the home set becomes valid.

To reduce the latency, we applied a similar method to PhantomCache [73], which accesses multiple cache banks in parallel to access multiple cache sets. However, due to the selection of the multiple cache sets, PhantomCache cannot guarantee that all the accessed cache sets in the same round are distributed normally into all the different cache banks. This may cause multiple accessed cache sets to be mapped to the same cache bank. Hence, the PhantomCache may have high latency in some accesses.

Nevertheless, this does not become an issue in the SEA cache. We limit the cache set number in each cache bank so that it can be expressed as equation 5.1. Here, N is the calculation round of the logical associativity module, which starts from 0 and ends when all logical cache ways are requested. Since the set numbers within a logical cache set must be continuous, if the SEA cache issues at most Num_{Banks} requests to the banks, there must be no conflicts in bank access. Comparing CEASER-SH in Figure 5.1(a), we show a SEA cache example with 4 cache banks in Figure 5.1(b). This SEA cache has 4 banks. The SEA cache first spends one clock cycle to compute the set numbers of all requests, and then all the requests are sent to the banks in parallel in one round. The entire access takes only 1 more clock cycle than the ScatterCache [80]. Compared to the CEASER-SH example shown in Figure 5.1(a), the SEA cache with 4 banks requires 2 fewer clock cycles.

$$Set = HomeSet + BankOffSet + (N \times Num_{Banks}) \quad 0 \leq N < \frac{Num_{Sets}}{Num_{Banks}} \tag{5.1}$$

However, even if there are only 2 cache banks, as in the previous example, the SEA cache is still applicable. It needs to separate all cache accesses into two rounds. This is shown in Figure 5.1(c). In the first clock cycle, after the home sets of both cache ways become valid, the SEA cache still needs to compute the set number of the next logical cache way. After the first clock cycle, both the home sets: $S1W0$ and $S2W1$, and the following sets: $S2W0$ and $S3W1$, are sent to the cache banks as requests. Meanwhile, the other cache sets are calculated based on the first-round cache set numbers. After the first round is sent to the banks, the second round cache sets become valid and can be sent to the cache banks. We name the cache sets whose requests are sent to the same cache bank as the requests to the home set (including the home set itself) as **base cache sets**, and other cache sets are named **extension cache sets**. In Figure 5.1, we highlight all base cache sets in red.

In the SEA cache, the lowest logical associativity a user can set is 1, which is no logical associativity. When logical associativity of 1 is applied, we expect the SEA cache to perform the same as Scattercache [80]. In other words, when a user sets their logical associativity to 1, the home set number bypasses the logical associativity module and is directly sent to the cache banks. When the logical associativity of the access user is higher than 1, the SEA cache should access multi cache-banks in parallel. Since only the home set does not have to be computed, all other cache set numbers involved in the first round still need to be computed, which will take one extra clock cycle after the home set is valid. For example, see $S2W0$ and $S3W1$ in Figure 5.1(b,c) have to be calculated.

The logical associativity module is further modified to achieve the specification described above, as shown in Figure 5.2. The SEA logical associativity module has two modules: the Base Module and the Extension Module. The Base Module is used for computing all the base cache sets, and the Extension Module is used for calculating extension cache sets. An 8 banks SEA cache, for example, requires one Base Module and seven Extension Modules in each cache way. By applying these modules, the SEA cache can automatically select the access mode to the cache banks. The serial bank-access mode is selected when the logical associativity is one, and vice versa. Hence, no extra delay is added when the logical associativity is one. This is controlled by a signal called *Bypass*

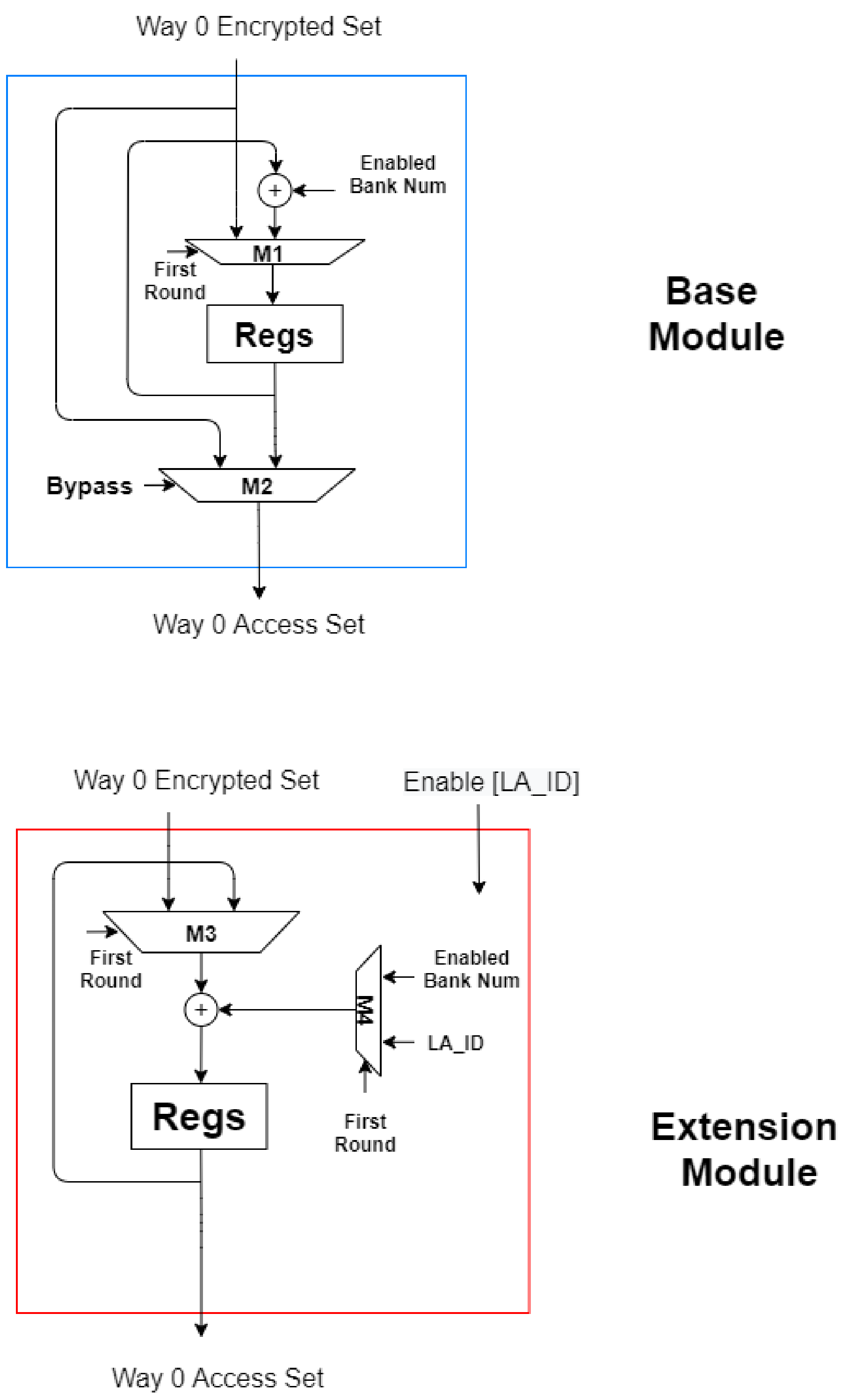


FIGURE 5.2: The implementation of the logical associativity module in the SEA cache.

and the M2 multiplexer. The Bypass signal only goes high when the logical associativity of the access user is set to 1. This is the major difference between the Base Module and the Extension Module. Since all extension cache sets must be computed before sending to banks, they do not need to bypass the logical associativity module. To improve the power consumption, the extension modules can be disabled when they are not used during the access.

When calculating the set numbers, the first round calculation differs from the later rounds. By adding continuous offsets to the home set, all set numbers in the first round can be computed and stored in the registers. However, the later rounds add a constant offset, which is the number of cache banks the SEA cache used, to the previously saved results in the registers. Hence, we add a multiplexer M4 to select the appropriate offset in the calculation. Furthermore, in the parallel bank-access mode, the home set is not bypassed but is saved in the registers. Therefore, the Base module does not require an M4 multiplexer. The values in the registers of the Base module will be sent at the same time as the other first-round cache sets and used for computing later-round cache sets.

5.2.2.3 Smart Bank Access

Unlike PhantomCache [73], which always has high power consumption due to the parallel bank accesses, the SEA cache can partially enable the logical associativity module and send requests to partial cache banks. The simplest example is what we discussed in Section 5.2.2.2. The SEA cache can issue only one request in the serial bank-access mode.

Even in the parallel bank-access mode, the SEA cache can still issue requests to partial cache banks. For example, if the logical associativity is set to 6 but there are 8 cache banks in the SEA cache, the SEA cache can issue 6 requests. This can save more power on bank access. For the logical associativity module, since the Base module is always used in each cycle, it can be enabled all the time. The Extension Modules can normally be disabled and enabled with the corresponding logical associativity.

5.2.3 Data sharing in the SEA cache

5.2.3.1 Data sharing problem

In the default SEA cache, each user has their own logical associativity. However, having different logical associativity for different users could lead to an issue with sharing data. A user with lower logical associativity may not be able to access the shared cache line that was accessed by another user with higher logical associativity. This can cause a fatal error in the memory system due to the data coherence issue. For example, in Figure 5.3, two users (A and B) share the same machine. Each user controls their own VM. The logical associativities for users A and B are 3 and 1, respectively. In Figure 5.3(a), user A accesses the cache line X first. The home set of the cache line X is set 0. However, since the logical associativity of user A is 3, cache line X might be placed at $S0$, $S1$ or

$S2$. In this example, X is placed in $S1$. Figure 5.3(b) shows user B accessing this cache line X after user A's access. Due to their logical associativity, the cache will only search $S0$. This results in a cache miss. Hence, the cache will request another copy of cache line X with the same virtual address. This should never happen in a cache since the cache cannot distinguish these two cache lines X. Particularly if one of the cache lines is modified, this can cause a data coherence problem.

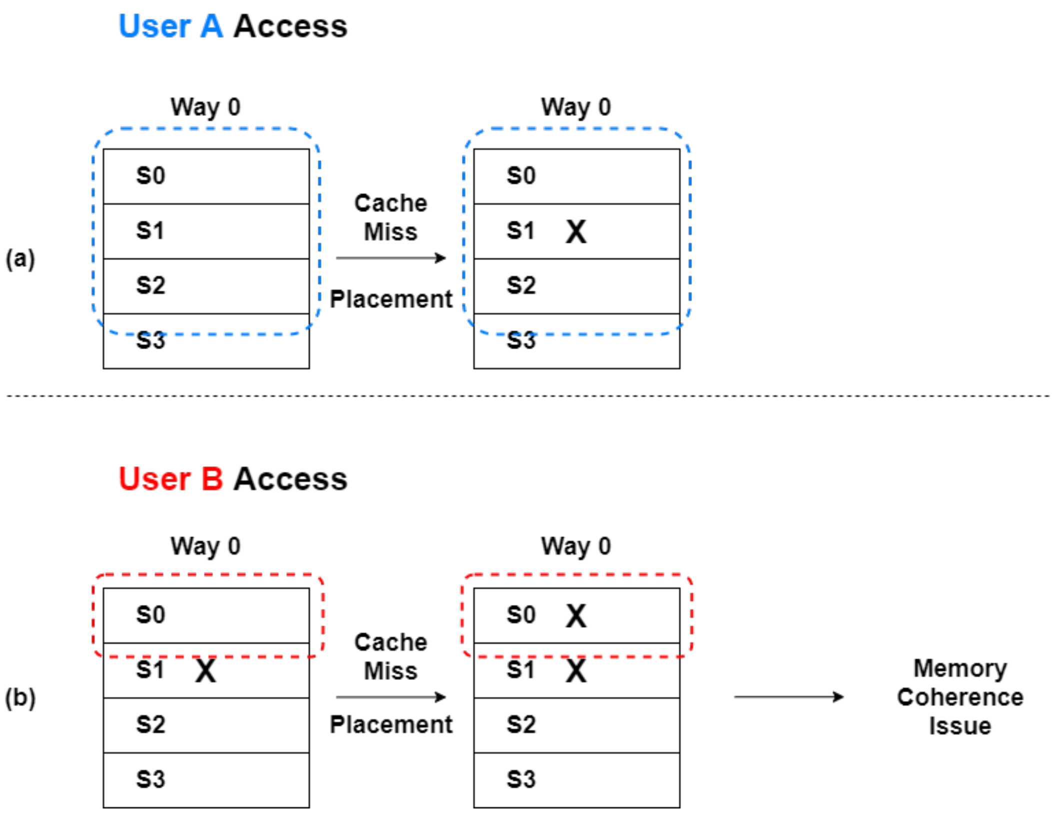


FIGURE 5.3: An example of sharing data problem while two users have different logical associativities, which finally leads to a memory coherence problem.

As a result, we can disable the data sharing between different users in the SEA cache. This is a simple solution to overcome such an issue. This method was also proposed against Reuse-Based attacks [85]. However, the memory system may not be used efficiently because dual or multiple copies of cache lines may be required within the same library, even for read-only. To enable data sharing, the SDID should be used with the SEA cache. We now explain the use of SDID.

5.2.3.2 SEA cache with SDID

To allow cache lines in a page to be shared between users, the SEA cache must guarantee that all cache lines within this page are placed with the lowest logical associativity of all users who share this page. Otherwise, the users who share this page must have the same logical associativity. However, the default SEA cache only distinguishes the logical associativity by the user's id. Adding an identifier on to each page can solve the problem. We noticed a similar identifier, called the security domain identifier (SDID), applied in some secure cache designs [80, 63]. This can also be used but in a different way in the SEA cache to solve the data-sharing problem. Nevertheless, this cannot be achieved without the hypervisor and the MMU, which control the paging.

SDID was used to distinguish between different security domains of the processes. Based on the security requirement of each process, the process is allocated to a different domain by the hypervisor. The SDID and the address of cache lines are treated as the plaintext of the indexing function in these cache designs. Therefore, different security domains lead to distinct mappings even if the same address is accessed. SDID was introduced to enhance the protection against contention-based attacks. However, the later profiling method Prime+Prune+Probe may invalidate the security enhancement of the SDID within those caches.

Ideally, the SDID allows different processes to be mapped with different mappings, and different mappings can increase the complexity of building the eviction set. This technique does affect the traditional profiling method but does not affect Prime+Prune+Probe (PPP) profiling. K cache lines are randomly selected during such a profiling method. As explained in 2.6.1.2, K is the number of the initial candidates in the PPP profiling. The Prime and Prune steps are only affected by the distribution of the mapping from the attacker's perspective since no victim cache line is accessed in these two steps. After these steps, some of those K cache lines may be evicted by themselves. Therefore, the number of cache lines left in the cache is reduced to K'. Hence, the attacker occupies K' cache lines in the cache, the capturing probability of the PPP is only related to the coverage of the K' candidates in the last-level cache. Therefore, the SDID does not influence the attack during the profiling stage. Since all eviction cache lines are guaranteed to contend with the victim target cache line within one or more cache ways, the attack's success rate is decided by the cache way coverage.

In the SEA cache, we can modify and apply the SDID, not for different mappings, but for achieving different logical associativity for each security domain. Based on the security requirements of the user, the process can be allocated to the corresponding security domain. As the default, all users' pages are assigned the SDID bits with the lowest logical associativity that the vendor sets. Hence, those processes that do not require extra protection can share the same logical associativity and data. When a user asks the hypervisor for extra protection against contention-based attacks, the hypervisor modifies the SDID bits that have high logical associativity setting in this process's pages. By applying such modifications, the SEA cache only needs to compare the SDID in the page to determine the logical associativity when accessing a cache line.

By having the SDID, the SEA cache can now provide process-level protection against contention-based attacks. In other words, the SEA cache with SDID further optimises the performance and the security by allowing some processes to be protected by high logical associativity, and other processes can access the cache with relatively low latency.

In a contention-based attack, the attacker may not want their attacking process assigned the same SDID as the victim process. This is because the targeted process must be given an SDID with high logical associativity. By having the same logical associativity, the attacking process does not gain any benefits but suffers from the extra protection due to the overlapping of logical associativity. From the attacker's perspective, they must avoid high logical associativity.

An attacker may want to start a process before the victim so that the cache lines with the process can be loaded into the cache with low logical associativity until the victim user requests the shared data. Hence, the protection is weakened. To overcome this issue, we only allow data sharing within the same protection level (or SDID) and disable data sharing across different protection levels. A duplication page should be created if a page needs to be shared between two protection levels. This should work as a Copy-on-write, which happens when the shared data needs to be written by at least one user in a conventional hypervisor. Since most of the processes share the normal-protection level, only the security-sensitive processes request copies of the page. Two domains are enough for a practical implementation of the SEA cache with the SDID. One domain is set to have the lowest logical associativity, allowing all processes within this domain to achieve the lowest latency. We name it the normal protection domain. The other domain is dedicated to security-sensitive processes. This domain has a high logical associativity setting. We name it the high protection domain. Therefore, a one-bit SDID is sufficient. As a result, this data-sharing strategy in the enhanced SEA cache is much more practical than the default SEA cache.

An example of this is shown in Figure 5.4. Here, the logical associativities of the two security domains are 1 and 3. Before the sharing starts, process A accesses cache line X at the normal protection level, which is shown in Figure 5.4(a). In Figure 5.4(b), process B accesses the same cache line X at the high protection level and triggers the duplication of the shared page. The SDID of each page is assigned with the corresponding protection levels. After the new page is created, the cache line $X1$ is loaded into the cache with logical associativity 3, which is shown in Figure 5.4(c). This strategy does not prevent reuse-based attacks since the attacker can also request high-level protection and share the same page with the victim user. To prevent reuse-based attacks, the SEA cache can be combined with other protection schemes against reuse-based attacks, such as the random fill cache [42].

Overall, by allowing different processes to have different logical associativities, the SEA cache uses the SDID to not only enhance the security against contention-based attacks and the performance, but also to enable data sharing, which is disabled in the default

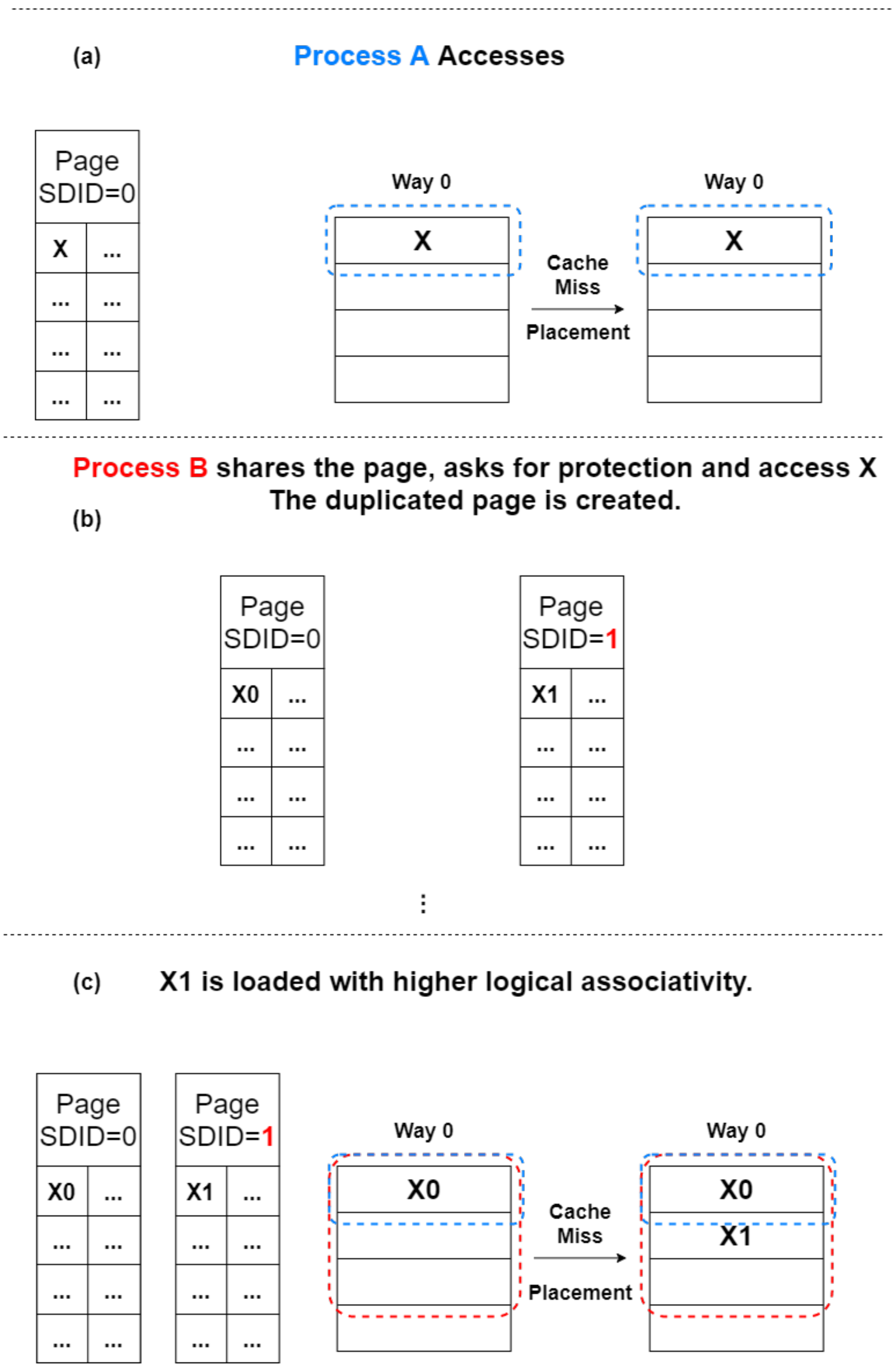


FIGURE 5.4: An example of SDID and logical associativity changes while the data is shared between A and B. Where A is in the $H = 1$ domain (circled in blue) and B is in the $H = 3$ domain (circled in red).

SEA cache. This is entirely distinct from using the SDID in all randomisation remapping cache designs.

5.3 Security Evaluation

5.3.1 Defence Strategy

The defence strategy and the threat model is not changed for the security of the SEA cache. However, since the logical associativity becomes a local configuration, both the attacker and the victim are allowed to set their own logical associativities. The attacker does not know what logical associativity is used by the victim. We name the logical associativity of the domain/user that the attacker is allocated to as AH and the domain/user that the victim is allocated to as VH.

Table 5.1: The success rate of the Prime+Probe attack with the eviction set of 752 on the SEA cache with different attacker logical associativity (AH) and victim logical associativity (VH).

AH \ VH	1	2	3	4	5	6
1	95%	78%	63%	52%	45%	39%
2	77%	68%	56%	48%	42%	37%
3	63%	56%	51%	44%	39%	35%
4	52%	48%	44%	40%	36%	32%
5	45%	41%	40%	36%	34%	31%
6	40%	37%	34%	32%	31%	29%

We further developed our security simulator, which was introduced in 3.3. To evaluate the impact of the eviction rate of the eviction set under different victim and user logical associativities, we use constructed eviction sets with a size of 752 to implement the Prime+Probe attack on the SEA cache with different configurations. Here, we do not consider re-keying. The results are shown in Table 5.1. When VH is fixed, a higher AH leads to a lower success rate of eviction. Therefore, from the attacker's perspective, they have no reason to increase their logical associativity. During the attack, the attacker must keep their logical associativity as 1. Hence, we only need to consider the impact of VH on the success rate. Because the AH is 1, the overlapping of the SEA cache becomes invalidated. Nevertheless, the reconfigurability is still valid, so an increased VH still increases the associativity and makes the contention-based attack harder. When the AH is 1, the eviction probability of the eviction set can be calculated as equation 2.2.

$$P_E = 1 - (1 - (\frac{1}{VH \times Nw}))^{\frac{G}{P}} \tag{5.2}$$

As a comparison, in a CEASER-SH cache, the success rate of applying an eviction set with a size of 752 for the Prime+Probe attack is reduced to about 40% when the global logical associativity is 4, as shown in Table 3.1. However, to achieve the equivalent success rate, the SEA cache requires VH to be adjusted to 6. The increase of VH might increase the victim LLC access latency, but the victim user only requires the higher VH when they need to run security-sensitive processes. Other users who do not need extra protection against contention-based attacks do not need to suffer higher latency. Nevertheless, if necessary, a privileged user may set the lowest logical associativity higher and enable the overlapping for even better protection.

5.3.2 SEA Cache Security Simulation Results

Table 5.2: The success rate of the Prime+Probe attack with optimal K (Opt K) values under different re-keying periods (RPK), the AH is set to 1.

VH	1	2	3	4	5	6	8	16	24
Opt K / RKP	16	32	32	32	64	64	128	64	64
9	1.00%	0.53%	0.36%	0.23%	0.23%	0.19%	0.11%	0.06%	0.05%
10	1.17%	0.58%	0.41%	0.30%	0.27%	0.20%	0.17%	0.06%	0.05%
15	1.78%	0.95%	0.63%	0.43%	0.38%	0.33%	0.24%	0.11%	0.08%
20	2.40%	1.21%	0.81%	0.65%	0.47%	0.44%	0.32%	0.16%	0.08%
22	2.80%	1.38%	0.82%	0.67%	0.58%	0.48%	0.35%	0.18%	0.14%
25	3.14%	1.50%	0.97%	0.84%	0.67%	0.58%	0.41%	0.19%	0.13%
29	3.68%	1.80%	1.26%	0.93%	0.74%	0.59%	0.48%	0.23%	0.15%
30	3.75%	1.89%	1.25%	0.97%	0.77%	0.65%	0.42%	0.22%	0.18%
35	4.45%	2.28%	1.41%	1.10%	0.95%	0.80%	0.56%	0.30%	0.20%
40	5.05%	2.53%	1.73%	1.22%	1.03%	0.84%	0.58%	0.32%	0.22%
45	5.65%	2.91%	1.89%	1.34%	1.13%	0.96%	0.72%	0.35%	0.29%
50	6.27%	3.17%	2.11%	1.59%	1.32%	1.03%	0.82%	0.41%	0.28%
75	9.63%	4.97%	3.22%	2.43%	1.99%	1.66%	1.25%	0.61%	0.44%
100	12.26%	6.38%	4.15%	3.28%	2.62%	2.17%	1.62%	0.75%	0.58%
200	23.15%	12.34%	8.46%	6.35%	5.12%	4.29%	3.30%	1.63%	1.11%
1000	73.34%	48.94%	35.75%	28.04%	23.42%	19.97%	15.28%	7.89%	5.51%

Since the threat model and the parameters of the PPP profiling are identical to those in other research [57] and the CEASER-SH cache in Chapter 4, the success rates of the Prime+Probe attack in these caches can be compared directly.

Considering both the re-keying and the logical associativity in the SEA cache, as with the CEASER-SH cache, we perform simulations under different re-keying periods. As discussed in Section 4.4, the unit of the re-keying period is the number of cache access per time. The number of cache access is evaluated from N, which is the number of cache lines in the cache. In the simulations, we set AH to 1. Similar to the results in Table 4.2, the highest success rates of the attack with different VH, K, and re-keying periods (RKP) are shown in Table 5.2, where K indicates the size of initial candidates for PPP profiling. Optimal K is the K value that provides the highest success rate of the attack under a particular VH.

Compared to the CEASER-SH cache, we show that after the overlapping function of the logical associativity becomes invalid, the attack's success rate under different re-keying periods increases. For example, the attack's success rate under the 1000N re-keying period and H=2 is 39.39% in the CEASER-SH cache, which is in red in Table 4.2. The success rate of the attack under the 1000N re-keying period and $VH = 2$ increases to 48.94%, which is in red in Table 5.2. Nevertheless, a SEA cache with 1000N re-keying period and $VH = 8$ and which has the same latency, can reduce the success rate of the attack to 15.28%. This dramatically enhances the security against contention-based attacks.

TABLE 5.3: The success rate of the Prime+Probe attack with Optimal K (Opt K) values under different re-keying periods(RPK), the AH is set to 8,16 and 24. ($AH = 1$ in the first column is for comparison.)

VH/AH	1/1	8/8	16/8	24/8	16/16	24/24
RKP \ Opt K	16	32	64	16	16	32
9	1.00%	0.10%	0.04%	0.04%	0.03%	0.04%
10	1.17%	0.09%	0.06%	0.05%	0.06%	0.05%
15	1.78%	0.17%	0.09%	0.06%	0.07%	0.05%
20	2.40%	0.24%	0.10%	0.10%	0.10%	0.07%
22	2.80%	0.26%	0.14%	0.11%	0.11%	0.07%
25	3.14%	0.26%	0.17%	0.11%	0.13%	0.10%
29	3.68%	0.34%	0.20%	0.15%	0.13%	0.10%
30	3.75%	0.37%	0.19%	0.17%	0.16%	0.12%
35	4.45%	0.40%	0.25%	0.18%	0.21%	0.13%
40	5.05%	0.44%	0.29%	0.19%	0.23%	0.14%
45	5.65%	0.47%	0.29%	0.23%	0.22%	0.17%
50	6.27%	0.60%	0.31%	0.23%	0.27%	0.22%
75	9.63%	0.89%	0.55%	0.42%	0.42%	0.30%
100	12.26%	1.17%	0.72%	0.49%	0.50%	0.35%
200	23.15%	2.30%	1.47%	1.03%	1.04%	0.76%
1000	73.34%	10.78%	6.95%	4.94%	5.64%	3.84%

Another minor impact is that the attacker cannot deduce the victim's associativity. The associativity value could impact the optimal K of the PPP profiling. For example, from Table 5.2, we find that the optimal K tends to be higher when the logical associativity increases. In the CEASER-SH cache, since the attacker shares the global logical associativity with the victim, they might be able to find the logical associativity by adding a preparation phase before the attack. In the preparation phase, the attacker could Prime+Probe their own cache line and measure the eviction probability to deduce the logical associativity. However, this becomes impossible in the SEA cache since the logical associativity is applied locally. Nevertheless, without knowing the VH, the attacker cannot predict the success rate of the attack when applying an eviction set, which can make the attack implementation even more complicated.

As mentioned, the lowest logical associativity can be set by the privileged users. Table 5.3 shows the success rate of the Prime+Probe attack with Optimal K values under different re-keying periods when AH is set to 8. When AH is forced to be increased due to the lowest logical associativity, the SEA cache can provide even better security against contention-based attacks due to overlapping. As an example, in Table 5.2, when $AH = 1$, $VH = 8$, and the re-keying period is 1000, the attack's success rate is 15.28%, which is in blue. Compared this to the result in Table 5.3; when $AH = 8$, $VH = 8$, and the re-keying period is 1000, the attack's success rate drops to 10.78% (also labelled in blue). Although increasing AH can enable overlapping and provide better security against contention-based attacks, this can also reduce the performance for the cores that are in the normal-protection domain. We will explain this further with performance results in Section 5.4.2.2.

Overall, when different users need to share a computer or server with an SEA cache as the LLC, those users who want to be protected can set their logical associativity high. However, high protection, namely high logical associativity in the domain, can cause higher access latency and lower performance in terms of CPI. As a result, other users would prefer the normal-protection domain that provides the best performance and the lowest cache access latency. From the attacker's perspective, they want to reduce the complexity and increase the success rate of contention-based attacks. Therefore, they would pretend they were normal users, so that they can at least have a lower VH value during the attack. Since an increase of either VH or AH can reduce the attack success rate, the privileged user can also set the minimum logical associativity in VH to enable overlapping, if needed.

5.4 Performance Evaluation

5.4.1 Simulation Setup

In this section, we evaluate the performance of the SEA cache. We use the same cache configurations in section 4.5.1. Here, we evaluate the SEA cache without the SDID. In other words, the alienation function of logical associativity is applied to different cores or users. In the simulation, we set two workloads on two different cores. Each core has been set to a different logical associativity value. Core 0 is allocated to a user requiring extra protection against Contention-Based attacks, and core 1 is allocated to a normal user who expects high performance and no extra protection.

We apply the same benchmarks as in Section 4.5.1. Since we need one workload on each of the cores, we choose two programs from either the PARSEC Benchmark Suite [14] or the GAP Benchmark Suite [10] in each round of the simulations.

In the simulation, we assume there are 8 cache banks. Due to the parallel-bank to the cache banks, we assume that when the parallel-bank access mode is enabled, which means the logical associativity of the current access is more than one, the SEA cache requires one extra clock cycle to compute the extension cache sets. For example, when the logical associativity is one, the cache access latency is 43 clock cycles, and the cache access latency is increased to 44 clock cycles when the logical associativity is 2 to 8. In addition, when the logical associativity exceeds a multiple of 8, the cache access latency is increased by one clock cycle. For example, when the logical associativity is set between 9 to 16, the cache access latency is increased to 45 clock cycles.

We show the simulation results in the following subsections. Like the CEASER-SH cache, we compare the CPI and the MPKI for the SEA cache under different logical associativity.

5.4.2 Simulation Results

We analyse two different configurations. Also, we assume two security domains are used in the SEA cache. We set core 0 in the high-protection domain and core 1 in the normal-protection domain. In the first configuration, we set AH, which is the attacker's logical associativity (or logical associativity in the normal-protection domain here), constant at 1 and vary the VH, which is the victim's logical associativity (or logical associativity in the high-protection domain here). The results under the first configuration are discussed in Section 5.4.2.1.

A privileged user can set the lowest logical associativity limit for the SEA cache to enable overlapping and enhance its security against contention-based attacks. Therefore, in the second configuration, we set AH to 8 and vary the VH. Ideally, we should compare the CEASER-SH cache with the SEA cache. However, the CEASER-SH does not support parallel-bank access mode, which will cause significantly high latency compared to the SEA cache. Therefore, we used the SEA cache with $VH = AH$, which is equivalent to a CEASER-SH cache with parallel-bank access mode, instead of directly using the original CEASER-SH for comparison. This can provide a much fairer comparison. We choose two examples with VH and AH of 16 and of 24. The simulation results are shown in Section 5.4.2.2.

5.4.2.1 AH Set To 1

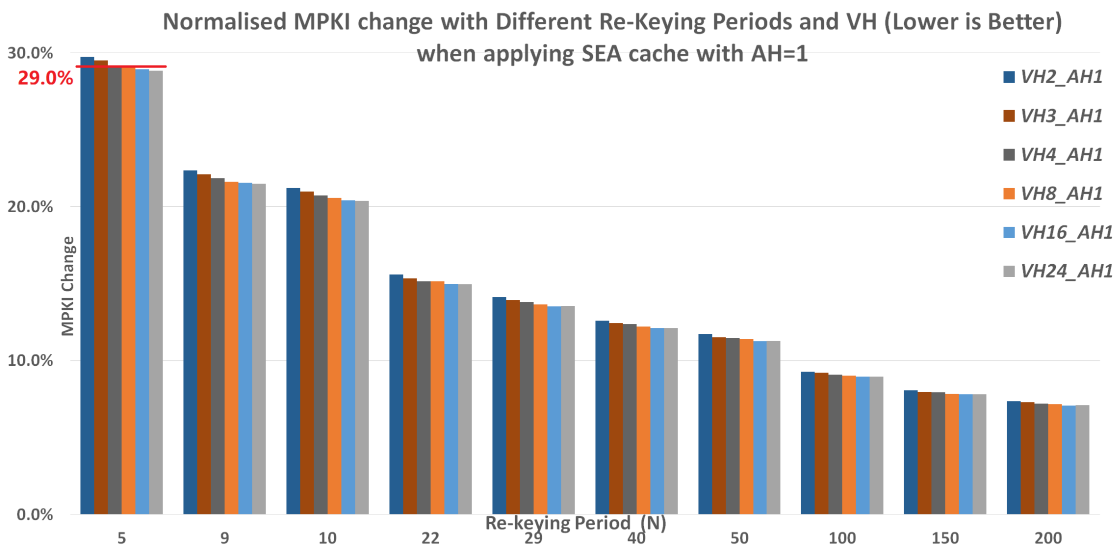


FIGURE 5.5: The MPKI of the SEA cache, AH is set to 1.

The MPKI of the SEA cache simulation, when the AH is set to 1, is shown in Figure 5.5. When the re-keying period is reduced, the different VH values all significantly reduce their MPKI. This trend is the same as for the CEASER-SH cache shown in Figure 4.5. Meanwhile, Figure 5.5 shows that the SEA caches with a higher VH lead to a slightly lower MPKI. For example, when the re-keying period is 5N, the MPKI of the SEA cache is above 29% when $VH2\_AH1$ and below 29% when $VH24\_AH1$. This is because the large logical associativity provides more potential placement positions for the cache lines from core 0 only. Furthermore, due to the cache lines from core 0 being more scattered when VH is larger, the contentions between the cache lines from these two cores are reduced. Hence, the overall SEA cache MPKI and the miss rate are slightly reduced when increasing the VH. Since the miss rates follow the same trend as the MPKI, they are not shown here.

The CPI of both core 0 and core 1 are shown in Figure 5.6 and Figure 5.7, respectively. As mentioned at the beginning of Section 5.4.2, the user who requires high protection against contention-based attacks is allocated to core 0 which has high logical associativity, whereas the other user is allocated to core 1 which has low logical associativity with normal protection. From the CPI results for core 0, which are shown in Figure 5.6, we can see that the SEA caches with VH 2 to 8 ($VH2\_AH1$, $VH3\_AH1$,$VH4\_AH1$, $VH8\_AH1$) have approximately the same CPI under all re-keying periods. The CPI for core 0 with $VH16\_AH1$ and $VH24\_AH1$ are always higher than for the SEA caches with VH 2 to 8. This is because the SEA caches with $VH16\_AH1$ and $VH24\_AH1$ both have a higher latency when core 0 sends the requsts due to the high logical associativity. For example in Figure 5.6, when the $RKP = 5N$, the CPI in core 0 increases by about 0.4% when the VH is increased from 2 to 24. The CPI for core 1 is distinct from the

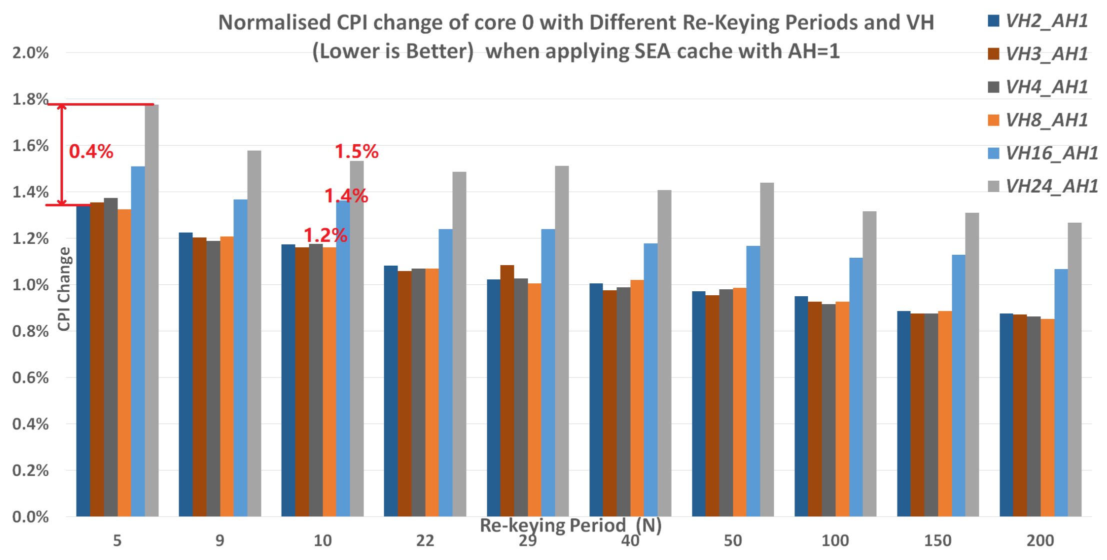


FIGURE 5.6: The CPI of core 0 for the SEA cache. AH is set to 1. Core 0 has different logical associativity, and the logical associativity of core 1 is set to 1.

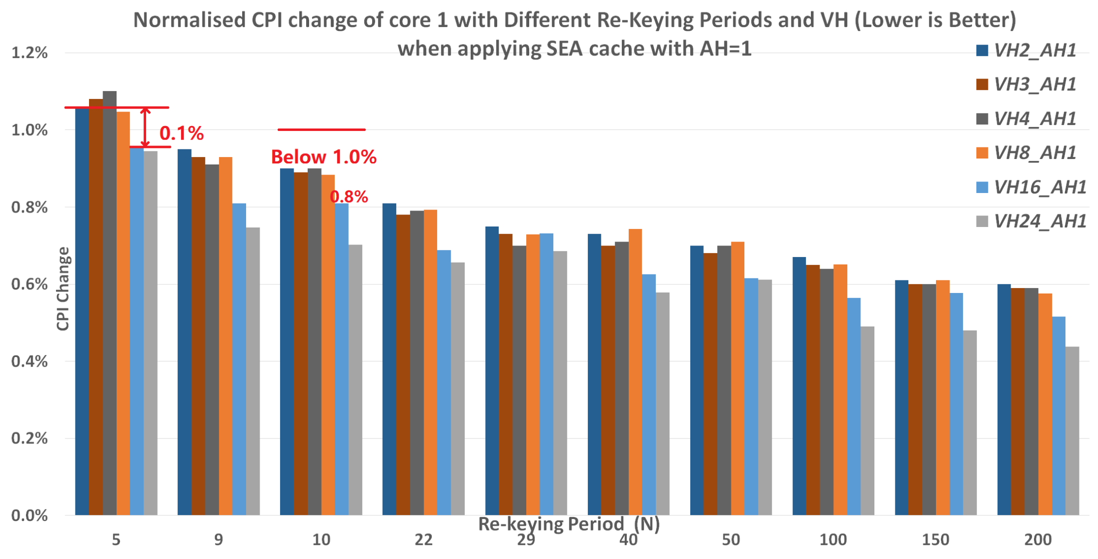


FIGURE 5.7: The CPI of core 1 for the SEA cache. AH is set to 1. Core 0 has different logical associativity, and the logical associativity of core 1 is set to 1.

CPI in core 0, which is shown in Figure 5.7. The CPI for core 1 is approximately independent of the value of VH. The CPI in core 1 with $VH16\_AH1$ and $VH24\_AH1$ are even slightly lower than VH 2 to 8. This reduction of the CPI could be caused by the MPKI reduction since a lower MPKI can reduce the total miss penalty. Nevertheless, the average impact of the VH value on the CPI in core 1 is just about 0.1%, which can be considered to be negligible.

In summary, from Figure 5.6 and Figure 5.7, we find that an increase in VH, particularly when VH is increased to 16 and 24, only increases the CPI in core 0 and has almost no impact on core 1, no matter the re-keying period. Furthermore, from Section 5.3.2, we know that only increasing VH can also reduce the success rate of contention-based attacks. Therefore, the SEA cache can enhance the protection against contention-based attacks by only changing the victim's logical associativity while providing no performance degradation to other users who do not require that protection.

5.4.2.2 AH Set To 8/16/24

We also simulated the SEA cache with higher AH. In this case, the privileged user may decide to force the overlapping function of the logical associativity and the lowest logical associativity to 8, 16 or 24, which are multiples of the number of cache banks. Hence, the logical associativity in the normal-protection domain (or AH) is set to 8, 16 or 24. Figure 5.8 shows the MPKI with $VH8\_AH8$, $VH16\_AH8$, $VH24\_AH8$, $VH16\_AH16$, $VH24\_AH24$. Compared to the results in Figure 5.5, which shows the SEA caches with $AH = 1$, the SEA caches with these VH and AH configurations show a lower MPKI under the same re-keying period. For example, when the re-keying period is $5N$, the SEA caches with $AH = 1$ and different VH in Figure 5.5 increase the MPKI by about 29%. However, in Figure 5.8, all SEA caches with $AH \geq 8$ increase the MPKI by about 27.7%, which is about 1.3% lower than the former. This is because higher logical associativity for the accesses from core 1 reduces the overall cache misses.

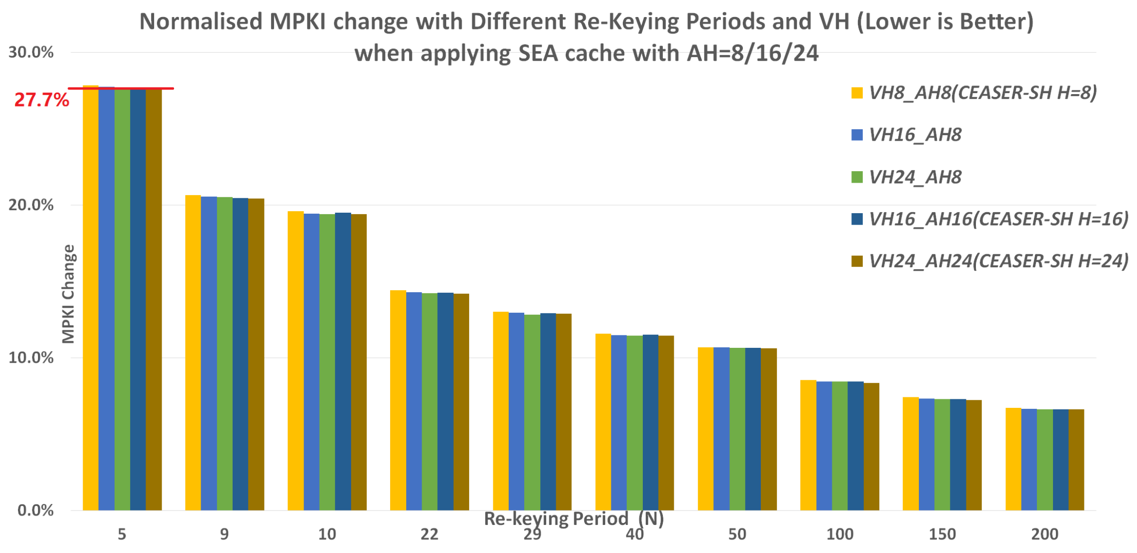


FIGURE 5.8: The MPKI of the SEA cache, AH is set between 8 to 24.

In terms of the CPI, the SEA caches with the same VH and different AH show very similar CPI in core 0 when they have the same re-keying period. For example, when the re-keying period is $10N$, the CPI increase in core 0 with SEA caches with $VH8\_AH1$, $VH16\_AH1$ and $VH24\_AH1$, shown in Figure 5.6, are 1.2%, 1.4%, 1.5%, respectively. The CPI increase in core 0 with SEA caches with $VH8\_AH8$, $VH16\_AH16$ and $VH24\_AH24$ are 1.2%, 1.4%, 1.6%, shown in Figure 5.9. These comparisons show that the increase of AH almost has no effect on CPI in the core 0, which is in the high-protection domain.

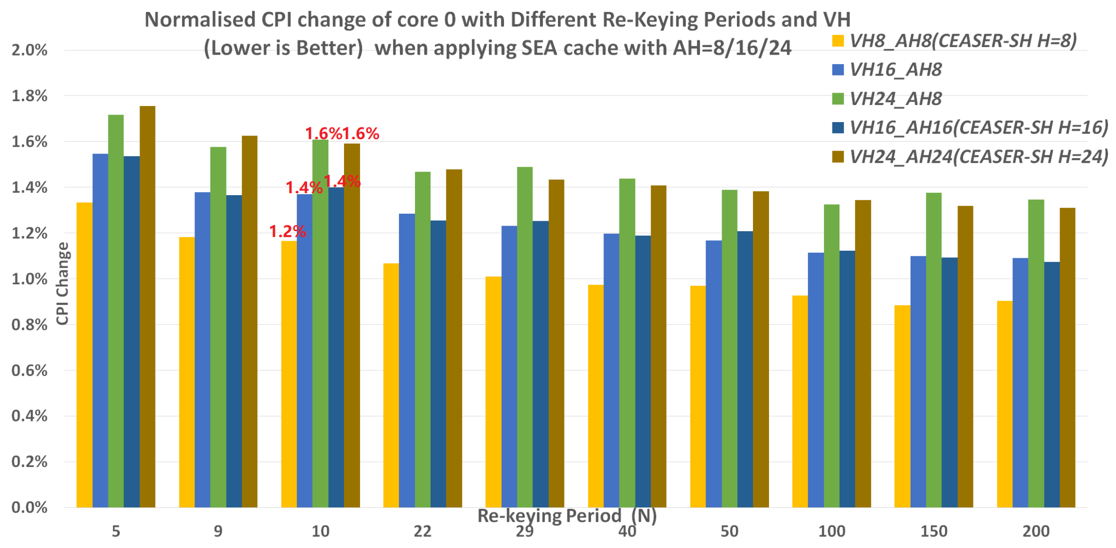


FIGURE 5.9: The CPI of core 0 for the SEA cache, both the AH and VH are set between 8 to 24.

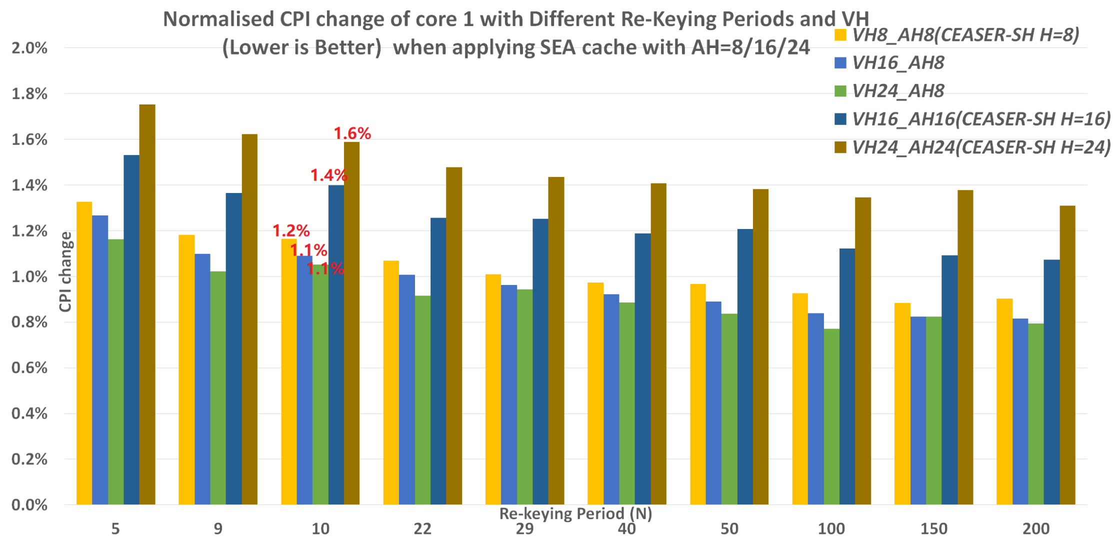


FIGURE 5.10: The CPI of core 1 for the SEA cache, both the AH and VH are set between 8 to 24.

Nevertheless, the same cache configurations show different results in CPI in core 1. For example, when the re-keying period is $10N$, the CPI increase in core 1 with SEA caches with $VH8\_AH1$, $VH16\_AH1$ and $VH24\_AH1$, shown in Figure 5.7, are all below 1%. The CPI increase in core 1 with SEA caches with $VH8\_AH8$, $VH16\_AH16$ and $VH24\_AH24$ are 1.2%, 1.4%, 1.6%, shown in Figure 5.10. These comparisons show that the increase of AH results in the increase of CPI in core 1, which is in the normal-protection domain.

The comparison of SEA caches with $VH8\_AH8$, $VH16\_AH8$, $VH16\_AH16$, $VH24\_AH8$ and $VH24\_AH24$ also shows the same trend. For example, when the re-keying period is $10N$, the CPI increases with these SEA cache configurations in core 0 are 1.2%, 1.4%, 1.4%, 1.6% and 1.6%, shown in Figure 5.9; and in core 1 are 1.2%, 1.1%, 1.1%, 1.4% and 1.6%, shown in Figure 5.10. SEA caches with the same VH show the same CPI in core 0 (high-protection domain) and the same AH shows the same CPI in core 1.

Overall, combining the summary in Section 5.4.2, we can conclude that **the change of AH or VH only affect the cache performance for the cache accesses from their own security domains.**

As mentioned at the beginning of Section 5.4.1, since the CEASER-SH cache does not support parallel-bank access mode, the extremely high latency can significantly increase the CPI of the cache, which is not a fair comparison. Therefore, instead of comparing the SEA cache with the CEASER-SH cache, we compare the SEA cache with $VH \neq AH$ and $VH = AH$. From a simulation perspective, the SEA cache with $VH = AH$ can be directly treated as a CEASER-SH with parallel-bank access mode. For example, in terms of cache performance, CEASER-SH cache with $H = 8, 16, 24$ and parallel-bank access mode is equivalent to SEA cache with $VH8AH8, VH16AH16, VH24AH24$. These are also labeled in Figure 5.8, Figure 5.9 and Figure 5.10. Again, it is worth noting that CEASER-SH and SEA cache have different hardware implementations, so such an equivalence only applies in the simulation.

During the evaluations, we need to consider both the cache performance and security against contention-based attacks. This is because changing the logical associativity can improve either cache performance or security. As an example, we compare the SEA cache with $AH1VH16$ and $AH8VH8$. From the security evaluation table shown in Table 5.2, when the re-keying period is $10N$, the attack success rate on the SEA cache is 0.06%. From Table 5.3, the attack success rate on the SEA cache with $AH8VH8$ (equivalent to CEASER-SH cache with parallel bank access mode and $H = 8$) is 0.10%, which is higher than the SEA cache. It is worth noting that the parallel-bank access mode has no effect on the security against contention-based attacks. Therefore, Table 5.3 is still valid.

For the performance, from Figure 5.7, when the re-keying period is $10N$, the CPI increase with the SEA cache with $AH1VH16$ is 0.8% on core 1 (normal-protection domain). From Figure 5.10, when the re-keying period is $10N$, the CPI increase with the SEA cache with $AH8VH8$ (equivalent to CEASER-SH cache with parallel-bank access mode and $H = 8$) is 1.2% on core 1. As a result, under the same re-keying period, the SEA cache achieves better protection against contention-based attacks in the high-protection domain while providing better performance (lower CPI) in the normal-protection domain, even if the CEASER-SH cache also has the parallel-bank access mode.

5.5 Hardware Overhead and Power Estimation

5.5.1 Hardware Overhead

We analyse the hardware overhead of the SEA cache in the same way as for the CEASER-SH cache. For a cache with the same configurations, the SEA cache should require the same storage overhead as the CEASER-SH, discussed in Section 4.6.1.1.

For the control logic, besides the changes on the CEASER-SH cache, the major modifications to the SEA cache are the registers for the logical associativity and the logical associativity module. We assume there are 8 cache banks for the SEA cache. The logical associativity module in each cache way requires one base module and seven extension modules. For a 16-way SEA cache, 16 base modules and 112 extension modules are required. Each base module or the extension module is formed by an adder, registers and two multiplexers. The bit width should be aligned with the index-bits of the SEA cache. For example, for a SEA cache with 8192 cache sets, the bit width should be 13-bit. We assume there are 64 cores. Therefore we need 64 5-bit registers (for allowing maximum logical associativity equals 32.) to store these values.

After synthesising with Nangate45nm PDK, we found that the total logic area overhead is $0.048mm^2$ or 60159 GE. The total area overhead, including the storage, is about $2.93mm^2$. This area overhead is $0.05mm^2$ higher than for the CEASER-SH cache, which is acceptable.

Even if SDID is applied, the hardware overhead is the same. The difference is the use of the additional registers. When SDID is applied, the number of registers depends on how many security domains are required. For example, the design only requires two 5-bit registers for supporting a maximum of 32 logical associativity. Where one domain is the security-insensitive domain, and the other is the security-sensitive domain. The latter could be set a higher logical associativity by the privileged user. Therefore, the SEA cache with SDID does not require extra registers for storing logical associativity. Similar to Scattercache, the SDID can be implemented via the user-defined bits in each page table entry [80], which does not require extra hardware.

5.5.2 Power Overhead Evaluation

Since the logical associativity of each user is different, the dynamic energy required for one access might be different. We assume half of the cores use logical associativity 1, and others use logical associativity 16. Based on the power results from CACTI6.0 [50] and the synthesised circuits, we estimate the power consumption of the SEA cache. The overall power consumption is $2.26W$. Compared to the conventional cache, it requires $0.24W$ extra power. The power optimisation method mentioned in Section 4.6.2 also

TABLE 5.4: The overhead comparison of a conventional cache, a CEASER-SH cache, amd a SEA cache with the same configurations.

	Cache Size 8M	**Set Associative** Baseline	**CEASER-SH**	**SEA cache**
Storage Overhead	Tag Bits	27		
	Status Bits	2		
	Set Bits	-	13	
	Total Tag Entry Bits	29	42	
	Tag Store Size	464KB (100%)	672KB (144.8%)	
	Total Storage	8656KB (100%)	8864KB (102.4%)	
	Area	55.6mm^2(100%)	58.4mm^2 (105.1%)	
Additional Logic Overhead	**Area**	-	31329 GE 0.025mm^2	60159 GE 0.048mm^2
Additional Overall Overhead	Area	-	2.88mm^2	2.93mm^2
Overall	Area	85.84mm^2	88.72mm^2 (+3.4%)	88.77mm^2(+3.4%)

applies to the SEA cache. Furthermore, the CEASER-SH cahe with $H = 8$, in Section 5.4.2.2, consumes 2.25W power. The power consumption is approximately the same as the SEA cache with AH1VH16.

5.6 Summary

In this chapter, we propose a Scattered Elastic-Associativity Cache (SEA cache) to further improve the performance of the CEASER-SH cache. In the default SEA cache, we allocate each core to one specific user at a time. Each user can set their logical associativity based on their requirements. For users who require extra protection against Contention-Based attacks, the higher logical associativity can be set locally. Users who do not require extra protection can keep the default logical associativity and low access latency. To further enhance the performance of the SEA cache, we apply a similar parallel bank-access method as PhantomCache[73]. We also discuss how the SEA cache handles the data sharing with SDID, this also extends the protection from the user level to the process level.

SEA cache can provide better security against contention-based attacks for specific users or processes while providing low latency and better cache performance for other users or processes, compared to the CEASER-SH cache. For example, in our simulations, we set core 0 as the high-protection domain and core 1 as the normal-protection domain. The success rate of the Prime+Probe attack on the SEA cache with $AH1VH16$ and $10N$ re-keying periods is 0.06%, and the success rate on the CEASER-SH cache with the parallel bank-access (equivalent to SEA cache with AH8VH8) and 100 re-keying period is 0.10%. Meanwhile, by checking the performance, the SEA cache increases the CPI in core 1 by 0.8%, compared to the conventional cache. In contrast, the CEASER-SH cache increases the CPI in core 1 by 1.2%. As a result, compared to CEASER-SH, SEA cache can provide better security to users require the high-proteciton and better performance to other normal users.

In terms of the overhead, compared to the conventional cache, we estimate the overall area overhead of the SEA cache is about $2.93mm^2$, including the storage overhead. Hence, the area is still increased by 3.4%. By setting the logical associativity to AH1VH16, the SEA cache requires $0.24W$ more power than the conventional cache. Also, compared to the CEASER-SH cache, the SEA cache with AH1VH16 only requires $0.01W$ extra power. As discussed in 4.6.2, this power overhead is negligible in a commercial processor.

Overall, the SEA cache provides a more flexible configuration to the cloud vendor and the users. Such flexibility allows different users to choose between high performance or high security against Contention-Based attacks. The hardware overhead of the SEA cache design is approximately the same as the CEASER-SH cache. Also, the SEA cache can outperform the CEASER-SH cache in terms of the normal users' performance and security against contention-based attacks, while the power consumption increase is very minor. We conclude that the SEA cache is feasible for the last-level cache design of the modern server CPU.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Cache is fast-access data storage in a computer system. Caches reduce the overall data access time compared to fetching data directly from the main memory. However, a cache sharing between different users can be leveraged by an attacker to retrieve a user's security-sensitive data, such as an encryption key. One type of attack is the Contention-Based attack. Such an attack can help the attacker to check the residency of the targeted victim cache line in the cache. As a side-channel attack, a contention-based attack can be achieved remotely and does not require any privileges. Therefore, this type of attack is dangerous and should be considered in cache system design, especially for a computer processor used as a cloud server.

Many countermeasures have been proposed to overcome contention-based attacks in the last 15 years, such as cache partitioning, cache monitoring and cache remapping. Cache partitioning, for example the NoMo cache [26], physically separates the victim's and the attacker's data in the cache, which guarantees no cache line residency information can be leaked. Although this method prevents contention-based attacks, it is also an ineffective use of cache storage. Cache monitoring is an active method that uses an access monitor to detect if any suspicious behaviour is found in the cache. After suspicious behaviour is detected, the cache activates a self-defence method, for example, terminating the process that triggers the monitoring [77]. However, such behaviour may be triggered by an innocent process. An other countermeasure is cache remapping. By using a cryptographic function to recompute the index-bits of the cache line [59, 80] or a permutation table [78], the cache line is remapped to a different placement in the cache. A cache design that applies cache remapping is known as a randomisation remapping cache. Some randomisation remapping cache designs have also added re-keying to enhance security against contention-based attacks. Cache remapping does not affect the normal use of the cache but requires hardware modifications to the cache and increases the cache access latency. Therefore, many randomisation remapping caches have been proposed as a solution to contention-based attacks.

While countermeasures to contention-based attacks have been developed, the attacks have improved as well. Prime+Prune+Probe [58] was proposed as a new profiling method to accelerate contention-based attacks. By applying the new profiling method, an attacker can construct the eviction set and achieve an attack much faster than many secure cache designs expected. To mitigate this more advanced attack, existing randomisation remapping caches, such as ScatterCache [80], must enable re-keying and lower the re-keying period, which is the period between cache remappings, to enhance the cache security. However, reducing the re-keying period can also cause performance loss.

To mitigate the performance loss due to the decreased re-keying period, we propose a new protection scheme for the randomisation remapping cache, which is called Logical Associativity. This idea added flexibility to the existing randomisation remapping cache. The idea is inspired by the fact that associativity has a huge impact on the complexity of implementing a contention-based attack [86]. By applying the logical associativity, a randomisation remapping cache can increase the logical associativity to balance the security and the cache performance. Logical associativity allows a cache line to be placed not only in the cache set that it maps to but also in the following consecutive cache sets. When the logical associativity is set to 1, the logical associativity of the cache is disabled. There are three properties of logical associativity: Reconfigurability, Overlapping and Alienation. Reconfigurability of logical associativity allows logical associativity to be increased at any time. Overlapping ensures that increasing logical associativity can provide better protection than increasing the cache ways in a randomisation remapping cache. Alienation allows different users to have different logical associativity.

Based on the logical associativity idea, we first propose the CEASER-SH cache. This cache is based on the CEASER-S [60] and ScatterCache [80]. Since in a modern processor, the L3 cache is always shared between different users, who are exposed by a contention-based attack, the CEASER-SH cache is designed as an L3 cache. By utilising the first two properties of logical associativity, CEASER-SH provides flexible performance and protection against contention-based attacks. To evaluate our CEASER-SH design, we simulated it on the gem5 simulator [15]. We used the GAP Benchmark Suite and PARSEC Benchmark Suite. We also built our own security simulator to evaluate CEASER-SH under different re-keying periods and logical associativity. We found that when a more advanced attack pushes the re-keying period lower than $50N$ cache accesses per cache re-keying in the CEASER-S cache, the CEASER-SH cache can provide equivalent or better security while providing better performance by increasing the logical associativity. For example, when the logical associativity is increased to 2 and the rekeying period is set to $22N$ cache accesses per cache re-keying, the CEASER-SH cache achieves equivalent security to the CEASER-S cache with a re-keying period of $9N$ cache accesses per cache re-keying. Meanwhile, the CEASER-SH cache has about

0.8% lower CPI, and about 29% lower MPKI. To evaluate the hardware overhead and power consumption, we used CACTI6.0 [50]. Also, we synthesised the additional logic required for logical associativity and remapping using the Nangate45nmPDK. We estimate the overall area overhead is about $2.88mm^2$, which is a 3.4% increase compared to a conventional cache with the same cache size, including the storage overhead. A CEASER-SH cache with logical associativity 4 requires $0.124W$ more power than a conventional cache. Compared to a commercial processor, for example, an AMD5995wx processor which needs $280W$ power under maximum workloads [3], the power overhead is negligible. By using a state-of-the-art technology library, the overheads will be much less and feasible to the last-level cache design of the modern computer system. This fulfils our first objective.

Although CEASER-SH has improved the performance of the randomisation remapping cache, we found that the logical associativity in CEASER-SH is a global parameter which affects all users' performance and security. Not all users require such extra protection. To only apply the protections to limited users, we extend the CEASER-SH cache and propose a Skewed Elastic-Associativity Cache (SEA cache). This cache applies another feature of logical associativity, which is alienation. In the SEA cache, different users are allowed to have their own local logical associativity settings. Hence, users who do not require extra protection do not need to suffer the performance degradation caused by other users who require extra protection. To further improve the SEA cache performance, we apply for the parallel bank access as in PhantomCache [73]. This design technique reduces the access latency of the SEA cache since the SEA cache can issue at most the same number of requests to cache banks as the number of cache banks. However, in the default SEA cache, having different logical associativity can cause a data-sharing problem. With the default SEA cache, we disabled user data sharing. To provide a better solution, we applied SDID in an enhanced SEA cache to distinguish between protection levels of different pages. We estimate the overall area overhead is about $2.93mm^2$, which is approximately the same as the CEASER-SH cache. The control logic is synthesised with Nangate45nmPDK, and the cache storage is evaluated using CACTI6.0 with 45nm technology. By applying similar evaluation methods as CEASER-SH, we find that the SEA cache can outperform the CEASER-SH cache in terms of the normal user's performance and the security against contention-based attacks, while the power consumption increase is very minor. For example, the success rate of the Prime+Probe attack on the SEA cache with $AH1VH16$ and $10N$ re-keying period is 0.06%, and the success rate on the CEASER-SH cache with the parallel bank-access (equivalent to SEA cache with $AH8VH8$) and $10N$ re-keying period is 0.10%. Meanwhile, the SEA cache increases the CPI in core 1 by 0.8%. In comparison, CEASER-SH increases the CPI in core 1 by 1.2%. As a result, the SEA cache can outperform the CEASER-SH cache in both security and performance with 0.01W extra power. The total power overhead is less than 0.1% of an AMD5995wx processor [3]. This achieves our second objective.

6.2 Future Works

We identify the following three directions for further development and research.

1. In Chapter 4, we selected PRINCE [19], an existing lightweight cipher, as the indexing function for both the CEASER-SH cache and the SEA cache. Nevertheless, PRINCE still requires 3 clock cycle to compute the ciphertext. Reducing the latency of the indexing function could further improve the power and the performance of both the CEASER-SH cache and the SEA cache. Hence, we think building a dedicated indexing function that requires less time to compute could be a direction for further developments. Although the indexing function does not have to be a cryptographic function since the attacker cannot read either the plaintext (Cache line address) or the ciphertext (Encrypted address), the function itself should be invulnerable. Otherwise, the function could be abused and weaken the security of the cache, as for example, LLBC [16].

2. Since we did not find any open-source L3 cache design before we ended the work, we did not do the place and route. Although from the simulation results, we have shown solid evidence of the cache performance and security of both the CEASER-SH cache and the SEA cache, we can only estimate the power and area overhead. The power consumption may only be accurately evaluated on a fabricated processor. The second direction is to implement both caches on a real processor, fabricate and evaluate it. Or evaluate it by post-layout simulations.

References

[1] O. Acıçmez and ç. K. Koç. *Microarchitectural Attacks and Countermeasures*, pages 475–504. Springer US, Boston, MA, 2009. ISBN 978-0-387-71817-0.

[2] O. Acıiçmez and Ç. K. Koç. Trace-driven cache attacks on aes (short paper). In P. Ning, S. Qing, and N. Li, editors, *Information and Communications Security*, pages 112–121, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-49497-3.

[3] AMD. Amd ryzen™ threadripper™ pro 5995wx general specifications, Mar 2022. https://www.amd.com/en/product/11786, accessed on 08 May 2023.

[4] T. Anderson and M. Dahlin. *Operating Systems Principles & Practice Volume III: Memory Management Second Edition.* Recursive Books, USA, 2nd edition, 2014. ISBN 9780985673550.

[5] ARM. *Cortex-A8 Technical Reference Manual.* ARM. https://developer.arm.com/documentation/ddi0344/latest/, accessed on 08 May 2023.

[6] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S.tephen Twigg, H.uy Vo, and A. Waterman. The rocket chip generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html, accessed on 08 May 2023.

[7] J.-P. Aumasson, L. Henzen, W. Meier, and M. Naya-Plasencia. Quark: A lightweight hash. *Journal of Cryptology*, 26(2):313–339, 2013. ISSN 1432-1378. https://doi.org/10.1007/s00145-012-9125-6, accessed on 08 May 2023.

[8] J. Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, aug 1978. ISSN 0001-0782.

[9] J.-L. Baer. *Microprocessor Architecture: From Simple Pipelines to Chip Multiprocessors.* Cambridge University Press, USA, 1st edition, 2009. ISBN 0521769922.

[10] S. Beamer, K. Asanović, and D. Patterson. The gap benchmark suite, 2017. arXiv abs/1508.03619.

[11] F. Bellard. QEMU, a fast and portable dynamic translator. In *2005 USENIX Annual Technical Conference (USENIX ATC 05)*, Anaheim, CA, April 2005. USENIX Association.

[12] D. J. Bernstein. Cache-timing attacks on AES, 2005. http://cr.yp.to/antiforgery/cachetiming-20050414.pdf, accessed on 08 May 2023.

[13] S. Bhunia and M. Tehranipoor. Chapter 8 - side-channel attacks. In S. Bhunia and M. Tehranipoor, editors, *Hardware Security*, pages 193–218. Morgan K., 2019. ISBN 978-0-12-812477-2.

[14] C. Bienia. *Benchmarking Modern Multiprocessors.* PhD thesis, USA, 2011. AAI3445564.

[15] N. Binkert, G. Beckmann, B.and Black, S. K. Reinhardt, A. Saidi, A. Basu, J.oel Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, aug 2011. ISSN 0163-5964.

[16] R. Bodduna, V. Ganesan, P. SLPSK, K. Veezhinathan, and C. Rebeiro. Brutus: Refuting the security claims of the cache timing randomization countermeasure proposed in ceaser. *IEEE Computer Architecture Letters*, 19(1):9–12, 2020.

[17] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe. Present: An ultra-lightweight block cipher. In P. Paillier and I. Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007*, pages 450–466, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-74735-2.

[18] J. Bonneau and I.lya Mironov. Cache-collision timing attacks against aes. In L. Goubin and M. Matsui, editors, *Cryptographic Hardware and Embedded Systems - CHES 2006*, pages 201–215, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-46561-4.

[19] J. Borghoff, A. Canteaut, T. Güneysu, E. B. Kavun, M. Knezevic, L. R. Knudsen, G. Leander, V. Nikov, C. Paar, C. Rechberger, P. Rombouts, Søren S. Thomsen, and T. Yalçın. Prince – a low-latency block cipher for pervasive computing applications. In X. Wang and K. Sako, editors, *Advances in Cryptology – ASIACRYPT 2012*, pages 208–225, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-34961-4.

[20] S. Briongos, P. Malagón, J. de Goyeneche, and J. M. Moya. Cache misses and the recovery of the full aes 256 key. *Applied Sciences*, 9(5), 2019. ISSN 2076-3417.

[21] E. Bugnion, J. Nieh, and D. Tsafrir. *Definitions*, pages 1–14. Springer International Publishing, Cham, 2017. ISBN 978-3-031-01753-7.

[22] J.T. Butler and T. Sasao. Fast hardware computation of x mod z. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 294–297, 2011.

[23] A. Chen, C. Schmidt, H. Cook, W. W. Terpstra, J. Zhao, Y. Lee, and J. Liu. Sifive/block-inclusivecache-sifive, Apr 2021. https://github.com/sifive/block-inclusivecache-sifive, accessed on 08 May 2023.

[24] Morris D. Sha-3 standard: Permutation-based hash and extendable-output functions, August 2015.

[25] J. Daemen and V. Rijmen. The rijndael block cipher. Technical report, Apr 2003. https://csrc.nist.gov/CSRC/media/Projects/Cryptographic-Standards-and-Guidelines/documents/aes-development/Rijndael-ammended.pdf, accessed on 08 May 2023.

[26] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Trans. Archit. Code Optim.*, 8(4), jan 2012. ISSN 1544-3566.

[27] D. Gruss, C. Maurice, K. Wagner, and S. Mangard. Flush+flush: A fast and stealthy cache attack. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721*, DIMVA 2016, page 279–299, Berlin, Heidelberg, 2016. Springer-Verlag. ISBN 9783319406664.

[28] D. Gullasch, E. Bangerter, and S. Krenn. Cache games – bringing access-based cache attacks on aes to practice. In *2011 IEEE Symposium on Security and Privacy*, pages 490–505, 2011.

[29] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, pages 3–14, 2001.

[30] M. S. Haque, S. Vasudevan, A. S. Nihar, A. Easwaran, A. Kumar, and Y.C. Tay. A self-reconfiguring cache architecture to improve control quality in cyber-physical systems. In *2018 IEEE 21st International Symposium on Real-Time Distributed Computing (ISORC)*, pages 116–123, 2018.

[31] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 6th edition, 2017. ISBN 0128119055.

[32] W.-C. Hsu and J. E. Smith. Performance of cached dram organizations in vector supercomputers. *SIGARCH Comput. Archit. News*, 21(2):327–336, may 1993. ISSN 0163-5964.

[33] A. Jaamoum, T. Hiscock, and G. D. Natale. Scramble cache: An efficient cache architecture for randomized set permutation. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 621–626, 2021.

[34] N.P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *[1990] Proceedings. The 17th Annual International Symposium on Computer Architecture*, pages 364–373, 1990.

[35] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra, Q. Huang, K. Kovacs, B. Nikolic, R. Katz, J. Bachrach, and K. Asanovic. Firesim: Fpga-accelerated cycle-exact scale-out system simulation in the public cloud. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 29–42, 2018.

[36] M. Kayaalp, K. N. Khasawneh, H. A. Esfeden, N. Elwell, J.and Abu-Ghazaleh, D. Ponomarev, and A. Jaleel. Ric: Relaxed inclusion caches for mitigating llc side-channel attacks. In *Proceedings of the 54th Annual Design Automation Conference 2017*, DAC '17, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349277.

[37] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19, 2019.

[38] J. Kong, O. Aciicmez, J.-P. Seifert, and H. Zhou. Deconstructing new cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 2nd ACM Workshop on Computer Security Architectures*, CSAW '08, page 25–34, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605583006.

[39] C. A. Lara-Nino, M. Morales-Sandoval, and A. Diaz-Perez. Novel fpga-based low-cost hardware architecture for the present block cipher. In *2016 Euromicro Conference on Digital System Design (DSD)*, pages 646–650, 2016.

[40] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M.ike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX*

Security 18), pages 973–990, Baltimore, MD, August 2018. USENIX Association. ISBN 978-1-939133-04-5.

[41] F. Liu and R. B. Lee. Security testing of a secure cache design. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450321181.

[42] F. Liu and R. B. Lee. Random fill cache architecture. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 203–215, 2014.

[43] F. Liu, H. Wu, K. Mai, and R. B. Lee. Newcache: Secure cache architecture thwarting cache side-channel attacks. *IEEE Micro*, 36(5):8–16, 2016.

[44] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622, 2015.

[45] C. M, K.J.M. Moriarty, B. Plache, and T. Trappenberg. Bank conflict resolution. *Computer Physics Communications*, 83(2):125–129, 1994. ISSN 0010-4655.

[46] C. Maurice, C. Neumann, O. Heen, and A. Francillon. C5: Cross-cores cache covert channel. In M. Almgren, V. Gulisano, and F. Maggi, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 46–64, Cham, 2015. Springer International Publishing. ISBN 978-3-319-20550-2.

[47] P. Megha Mukundan, S. Manayankath, C. Srinivasan, and M. Sethumadhavan. Hash-one: a lightweight cryptographic hash function. *IET Information Security*, 10(5):225–231, 2016.

[48] T. Mell, P.and Grance. The nist definition of cloud computing, September 2011.

[49] M. A. Mukhtar, M. K. Bhatti, and G. Gogniat. Ie-cache: Counteracting eviction-based cache side-channel attacks through indirect eviction. In M. Hölbl, K. Rannenberg, and T. Welzer, editors, *ICT Systems Security and Privacy Protection*, pages 32–45, Cham, 2020. Springer International Publishing. ISBN 978-3-030-58201-2.

[50] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pages 3–14, 2007.

[51] S. Naffziger, K. Lepak, M. Paraschou, and M. Subramony. 2.2 amd chiplet architecture for high-performance server and desktop products. In *2020 IEEE International Solid- State Circuits Conference - (ISSCC)*, pages 44–45, 2020.

[52] D. Ojha and S. Dwarkadas. Timecache: Using time to eliminate cache side channels when sharing software. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 375–387, 2021.

[53] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: The case of aes. In D. Pointcheval, editor, *Topics in Cryptology – CT-RSA 2006*, pages 1–20, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-32648-9.

[54] D. Page. Theoretical use of cache memory as a cryptanalytic side-channel. *IACR Cryptology ePrint Archive*, 2002:169, 01 2002.

[55] K. Pingali. *Locality of Reference and Parallel Processing*, pages 1051–1056. Springer US, Boston, MA, 2011. ISBN 978-0-387-09766-4.

[56] S. A. Przybylski. Chapter 1 - introduction. In S. A. Przybylski, editor, *Cache and Memory Hierarchy Design*, The Morgan Kaufmann Series in Computer Architecture and Design, pages 1–8. Morgan Kaufmann, San Francisco (CA), 1990.

[57] A. Purnal, L. Giner, D. Gruss, and I. Verbauwhede. Systematic analysis of randomization-based protected cache architectures. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 987–1002, 2021.

[58] A. Purnal and I. Verbauwhede. Advanced profiling for probabilistic prime+probe attacks and covert channels in scattercache. *arXiv CoRR*, abs/1908.03383, 2019.

[59] M. K. Qureshi. Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 775–787, 2018.

[60] M. K. Qureshi. New attacks and defense for encrypted-address cache. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 360–371, 2019.

[61] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive insertion policies for high performance caching. *SIGARCH Comput. Archit. News*, 35(2): 381–391, jun 2007. ISSN 0163-5964.

[62] M.K. Qureshi, D. Thompson, and Y.N. Patt. The v-way cache: demand-based associativity via global replacement. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 544–555, 2005.

[63] G. Saileshwar and M. Qureshi. MIRAGE: Mitigating Conflict-Based cache attacks with a practical Fully-Associative design. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1379–1396. USENIX Association, August 2021. ISBN 978-1-939133-24-3.

[64] D. Sanchez and C. Kozyrakis. The zcache: Decoupling ways and associativity. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 187–198, 2010.

[65] N. K. Sehgal, P. C. P. Bhatt, and J. M. Acken. *Cloud Computing and Information Security*, pages 111–141. Springer International Publishing, Cham, 2020. ISBN 978-3-030-24612-9.

[66] A. Seznec. A case for two-way skewed-associative caches. *SIGARCH Comput. Archit. News*, 21(2):169–178, may 1993. ISSN 0163-5964.

[67] A. Shalabi, T. Ghasempouri, P. Ellervee, and J. Raik. Scaat: Secure cache alternative address table for mitigating cache logical side-channel attacks. In *2020 23rd Euromicro Conference on Digital System Design (DSD)*, pages 213–217, 2020.

[68] J. Shi, X. Song, H. Chen, and B. Zang. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 194–199, 2011.

[69] SI2. 15nm open-cell library and 45nm freepdk, accessed in July 2022. https://si2.org/open-cell-library/, accessed on 08 May 2023.

[70] W. Song, B. Li, Z. Xue, Z. Li, W. Wang, and P. Liu. Randomized last-level caches are still vulnerable to cache side-channel attacks! but we can fix it. *arXiv CoRR*, abs/2008.01957, 2020.

[71] W. Song and P. Liu. Dynamically finding minimal eviction sets can be quicker than you think for Side-Channel attacks against the LLC. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 427–442, Chaoyang District, Beijing, September 2019. USENIX Association. ISBN 978-1-939133-07-6.

[72] M. Sundal and R. Chaves. Efficient fpga implementation of the sha-3 hash function. In *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 86–91, 2017.

[73] Q. Tan, Z. Zeng, K. Bu, and K. Ren. Phantomcache: Obfuscating cache conflicts with localized randomization. 01 2020.

[74] T. Unterluggauer, A. Harris, S. Constable, F. Liu, and C. Rozas. Chameleon cache: Approximating fully associative caches with random replacement to prevent contention-based cache attacks. In *2022 IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*. IEEE, sep 2022.

[75] P. Vila, B. Köpf, and J. F. Morales. Theory and practice of finding eviction sets. *arXiv CoRR*, abs/1810.01497, 2018.

[76] H. Wang, H. Sayadi, T. Mohsenin, L. Zhao, A. Sasan, S. Rafatirad, and H. Homayoun. Mitigating cache-based side-channel attacks through randomization: A comprehensive system and architecture level analysis. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1414–1419, 2020.

[77] K. Wang, F. Yuan, L. Zhao, R. Hou, Z. Ji, and D. Meng. Secure hybrid replacement policy: Mitigating conflict-based cache side channel attacks. *Microprocess. Microsyst.*, 89(C), mar 2022. ISSN 0141-9331.

[78] Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. *SIGARCH Comput. Archit. News*, 35(2):494–505, jun 2007. ISSN 0163-5964.

[79] Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, page 494–505, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595937063.

[80] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard. Scattercache: Thwarting cache attacks via cache set randomization. In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, page 675–692, USA, 2019. USENIX Association. ISBN 9781939133069.

[81] Z. Xinjie, W. Tao, M. Dong, Z. Yuanyuan, and L. Zhaoyang. Robust first two rounds access driven cache timing attack on aes. In *2008 International Conference on Computer Science and Software Engineering*, volume 3, pages 785–788, 2008.

[82] M. Yan, B. Gopireddy, T. Shull, and J. Torrellas. Secure hierarchy-aware cache replacement policy (sharp): Defending against cache-based side channel atacks. *SIGARCH Comput. Archit. News*, 45(2):347–360, jun 2017. ISSN 0163-5964.

[83] M. Yan, B. Gopireddy, T. Shull, and J. Torrellas. Secure hierarchy-aware cache replacement policy (sharp): Defending against cache-based side channel atacks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, page 347–360, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450348928.

[84] C. Yang, Y. Guo, and H. Hu. Memwander: Memory dynamic remapping via hypervisor against cache-based side-channel attacks. *IEEE Access*, 7:2179–2199, 2019.

[85] Y. Yarom and K. Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, page 719–732, USA, 2014. USENIX Association. ISBN 9781931971157.

[86] X. Yu, Y. Xiao, K. Cameron, and D. Yao. Comparative measurement of cache Configurations' impacts on cache timing Side-Channel attacks. In *12th USENIX Workshop on Cyber Security Experimentation and Test (CSET 19)*, Santa Clara, CA, August 2019. USENIX Association.

[87] X. Zhang, Z. Yuan, R. Chang, and Y. Zhou. Seeds of seed: H2cache: Building a hybrid randomized cache hierarchy for mitigating cache side-channel attacks. In *2021 International Symposium on Secure and Private Execution Environment Design (SEED)*, pages 29–36, 2021.

[88] Y. Zhang, A.ri Juels, M. K. Reiter, and T. Ristenpart. Cross-vm side channels and their use to extract private keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, page 305–316, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450316514.

[89] Y. Zhang, C. Xu, and X. S. Shen. *Secure Keyword Search*, pages 87–117. Springer Singapore, Singapore, 2020. ISBN 978-981-15-4374-6.

[90] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic. Sonicboom: The 3rd generation berkeley out-of-order machine. May 2020.