DOCTORAL THESIS

On the Detection and Mitigation of Cache Logic Side Channel Attacks

Ameer Shalabi

TALLINNA TEHNIKAÜLIKOOL
TALLINN UNIVERSITY OF TECHNOLOGY
TALLINN 2026

TALLINN UNIVERSITY OF TECHNOLOGY
DOCTORAL THESIS
82/2026

On the Detection and Mitigation of Cache Logic Side Channel Attacks

AMEER SHALABI

TAL TECH PRESS

TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technology
Department of Computer Systems
This dissertation was accepted for the defence of the degree 12/11/2026

Supervisor:
Professor Jaan Raik
Department of Computer Systems, School of Information Technology
Tallinn University of Technology
Tallinn, Estonia

Co-supervisor:
Professor Peeter Ellervee
Department of Computer Systems, School of Information Technology
Tallinn University of Technology
Tallinn, Estonia

Opponents:
Prof Giorgio Di Natale
TIMA
Université Grenoble Alpes
Grenoble, France

Prof Maria Mushtaq
Communications and Electronics (Comelec)
Telecom Paris
City, Country

Defence of the thesis: 23/11/2026, Tallinn

Declaration:
Hereby I declare that this doctoral thesis, my original investigation and achievement, submitted for the doctoral degree at Tallinn University of Technology has not been submitted for doctoral or equivalent academic degree.

Ameer Shalabi

---
signature

European Union
European Regional Development Fund

Investing in your future

Copyright: Ameer Shalabi, 2026
ISSN 2585-6898 (paperback)
ISBN 978-9916-80-602-9 (paperback)
ISSN 2585-6901 (PDF)
ISBN 978-9916-80-601-2 (PDF)
https://doi.org/10.23658/taltech.82/2026
Printed by Koopia Niini & Rauam

Shalabi, A. (2026). *On the Detection and Mitigation of Cache Logic Side Channel Attacks* [TalTech Press]. https://doi.org/10.23658/taltech.82/2026

TALLINNA TEHNIKAÜLIKOOL
DOKTORITÖÖ
82/2026

Vahemälu kõrvalkanali rünnete avastamine ja leevendamine

AMEER SHALABI

TAL TECH PRESS

Contents

List of Publications 7

Other Publications 8

Author's Contributions to the Publications 9

Abbreviations 10

1 Introduction 11
- 1.1 Motivation 12
- 1.2 Problem Formulation 14
- 1.3 Contributions 15
- 1.4 Thesis Organization 16

2 Background 17
- 2.1 Cache Memories 17
  - 2.1.1 Cache Hierarchy 17
  - 2.1.2 Cache Mapping 18
- 2.2 Logical Side-Channel Attacks 20
  - 2.2.1 Types of Cache Attacks 20
  - 2.2.2 Modeling Cache Attacks 21
- 2.3 Chapter Summary 22

3 Related Works 23
- 3.1 Cache Attack Detection 23
- 3.2 Mitigation of Cache Attacks 24
  - 3.2.1 Partition-Based Mitigation 24
  - 3.2.2 Randomization-Based Mitigation 24
  - 3.2.3 Adaptive Mitigation 25
- 3.3 Comparison of Existing Approaches 26
- 3.4 Chapter Summary and Research Gap 26

4 Experimental Setup 28
- 4.1 Target System 29
- 4.2 Experiments 30
- 4.3 Data Collection 30
- 4.4 Data Analysis 31
  - 4.4.1 Detection 31
  - 4.4.2 Mitigation 32
- 4.5 Chapter Summary 32

5 Leakage Detection 33
- 5.1 Proposed Methodology 33
  - 5.1.1 Threat Model 33
  - 5.1.2 Leakage Model 34
  - 5.1.3 Verification template 34
  - 5.1.4 Deriving miss/hit behaviour patterns 35
  - 5.1.5 Creating cache leakage template 36
- 5.2 CLD design and architecture 36

5.2.1 End-to-end example of CLD operation ... 37
5.2.1.1 Step 1: Leakage scenario identification ... 37
5.2.1.2 Step 2: Leakage template classification ... 37
5.2.1.3 Step 3: CLD FSM evaluation ... 37
5.3 Chapter Summary ... 38

6 Mitigation Mechanisms ... 39
6.1 SCAAT ... 39
6.1.1 Design and Functionality ... 39
6.1.2 Cache Coherency Considerations ... 39
6.1.3 Impact of Mitigation on Cache Behaviour ... 41
6.1.4 Requirements for $SCAAT_mem$ ... 42
6.1.5 Attack model and SCAAT vulnerability ... 42
6.2 RSCAAT ... 44
6.2.1 RSCAAT control logic ... 44
6.2.2 Self-Remapping Content Addressable Memory ... 45
6.2.3 SR_CAM LFSR ... 47
6.2.4 Randomization considerations ... 48
6.2.5 SR_CAM security ... 49
6.2.6 RSCAAT integration considerations ... 49
6.3 Chapter Summary ... 50

7 Experimental Results ... 51
7.1 Detection Analysis and Results ... 51
7.2 Mitigation Analysis and Results ... 53
7.3 Chapter Summary ... 59

8 Conclusion ... 61
8.1 Summary and Conclusions ... 61
8.2 Future Research Directions ... 62

List of Figures ... 63

List of Tables ... 64

References ... 65

Acknowledgements ... 71

Abstract ... 72

Kokkuvõte ... 74

Appendix 1 ... 75

Appendix 2 ... 81

Appendix 3 - RSCAAT: A recursive mechanism to mitigate cache side channel attacks. 89

Curriculum Vitae ... 109

Elulookirjeldus ... 111

List of Publications

The present Ph.D. thesis is based on the following publications that are referred to in the text by Roman numbers.

I 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

II A. Shalabi, T. Ghasempouri, P. Ellervee, and J. Raik. Cld: An accurate, cost-effective and scalable run-time cache leakage detector. In *2021 24th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, pages 127–132. IEEE, 2021

III A. Shalabi, T. Ghasempouri, P. Ellervee, and J. Raik. Rscaat: A recursive mechanism to mitigate cache side channel attacks. *IEEE Access*, 14:52342–52359, 2026

Other Publications

A. Shalabi, K. Paul, T. Ghasempouri, and J. Raik. Nv-sp: A new high performance and low energy nvm-based scratch pad. In *2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 54–59. IEEE, 2020

Author's Contributions to the Publications

I In I, I designed and implemented the logical side channel attack mitigation architecture. I created the testing environment and designed the methodology used to develop the mitigation mechanism. I created the model and hardware description of the mitigation mechanism. I performed verification and synthesis of the mitigation mechanism hardware. I was the main author and wrote the majority of the manuscript for publication. I created the simulation script and carried out simulations. I designed and created the figures and tables.

II In II, I designed and implemented the logical side channel attack detector model and architecture. I designed the testing environment and testing conditions. I created the attack and detection models used for the design of the detector. I created the hardware implementation of the detector. I performed verification and synthesis of the detector hardware. I was the main author and wrote the majority of the manuscript for publication. I created the simulation scripts for simulating attack scenarios as well as simulated cache behavior. I designed and created the figures and tables.

III In III, I identified a vulnerability in the design of SCAAT. I designed and implemented the redundant logical side channel attack mitigation architecture as well as its control logic. I created the testing environment and designed the methodology used to develop the mitigation mechanism. I created the model and hardware description of the mitigation mechanism. I performed verification and synthesis of the mitigation mechanism hardware. I was the main author and wrote the majority of the manuscript for publication. I created the simulation script and carried out simulations. I designed and created the figures and tables.

Abbreviations

AES	Advanced Encryption Standard
AMAT	Average Memory Access Time
CAM	Content-Addressable Memory
CLD	Cache Leakage Detector
CPU	Central Processing Unit
DRAM	Dynamic Random Access Memory
FSM	Finite State Machine
HPC	Hardware Performance Counter
IPC	Instructions Per Cycle
LFSR	Linear Feedback Shift Register
LiD-CAT	Lightweight Detector for Cache ATtacks
LLBC	Low-Latency Block Cipher
LLC	Last Level Cache
LSCA	Logical Side-Channel Attack
MMU	Memory Management Unit
MPKI	Misses Per Kilo Instructions
PARSEC	Princeton Application Repository for Shared-Memory Computers
PoC	Pile of Cores
ProcID	Process Identifier
RAM	Random Access Memory
RSCAAT	Recursive Secure Cache Alternative Address Table
SCAAT	Secure Cache Alternative Address Table
SPEC CPU	Standard Performance Evaluation Corporation CPU Benchmark Suite
SPLASH	Stanford Parallel Applications for Shared Memory
SRAM	Static Random Access Memory
SR_CAM	Self-Remapping Content Addressable Memory
SVA	SystemVerilog Assertions

1 Introduction

The ever increasing speed of microprocessors has long surpassed that of computer memory. One can attribute this disparity to the inherent differences between those two technologies. While such differences can be enumerated, it all boils down to their design targets: microprocessor designers are interested in gaining speed, while memories' core design is concerned with capacity. Thus, an ever widening processor-to-memory speed gap has emerged. This is known as the Von Neumann bottleneck [6]. Now, it is the primary hurdle to improving the performance of computer systems. Microprocessor designers, however, have access to an important resource that is halfway in between memory and the Central Processing Unit (CPU). While Register Files are typically limited in size, they are fast and situated very closely to the processing core [48]. This allows some degree of flexibility and eased the wide difference in speeds between the CPU and memory. Register files are a sufficient method for storing important information near the CPU for when it is needed. However, Register files have too low capacity to close the speed gap between memory access latency and processing speed.

To address this issue, Static Random Access Memory (SRAM) based cache were introduced as early as the 1960s [39]. It was only in 1993 when on-chip cache were first used [38]. The computer cache was a low capacity, high-speed memory device that sought to minimizing the difference in throughput between it's contemporary processors and system Random Access Memorys (RAMs). Nevertheless, the rate of increase of processor speed did not slow down. As processing speeds increased, several caches were needed for the slow main memory DRAM technology to catch up. The introduction of multi-core processors only exacerbated the situation. In response, system designers did what system designers do best: add more memory.

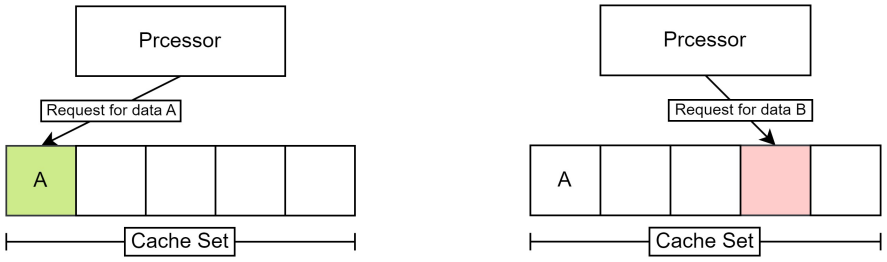
Today, no modern processor can operate without several layers of cache. Increasing in size and latency, a faster and smaller private cache, called the Level-1 or L1 cache, is placed neatly inside the processing core [7]. Several slower and larger shared caches, called Level-2 or L2 cache, Level-3 or L3 cache and so on, are placed between the different cores. The slowest and largest is the Last Level Cache (LLC); a half-way-between the computer processor and the main system memory [7]. Those caches effectively reduce the latency associated with data retrieval from the slow main memory.

The cache, while revolutionary, came at a cost. Caches are designed for both speed and capacity. Despite improvements in cache access speeds, capacity limitations remains a big issue [49]. Speed is important, but rising throughput requirements increasingly put pressure on designers to integrate more memory into the system. With increasing capacity, the complexity of memory management, a job typically given to the Memory Management Units (MMU) [54], increases. This complexity indirectly contributes to the increasing severity of memory latency. Memory latency is the time interval between the initial data request by the processor and the arrival of that data at the processor's internal data registers [50]. High latency is a strong indicator of poor performance. However, the penalties of memory latency are now going beyond performance degradation: it is becoming a possible security vulnerability [71].

Nonetheless, this variance in latency is not a defect. It is an inevitable feature of multi-level cache systems and it is here to stay. Thus, the most effective way to secure the cache against exploitation of this time variance is to detect suspicious access patterns to the different caches in the multi-level cache system and prevent adversaries of using such patterns.

1.1 Motivation

When a processor looks for data, it checks its nearest cache. If the data is there, we call that a **cache hit** and data is delivered at very low latency. When data is not present in the cache, we call that a **cache miss**. Figure 1 demonstrates a cache hit (1a) and a cache miss (1b) events. The penalty of a cache miss grows quickly when the data is not found at further cache levels. Thus, an inherent issue in multi-level cache systems is the significant variance between hit latency and miss latency [8]. This variance in timing is one of many issues that need to be addressed before multi-level cache systems are completely secured against threats from adversaries. A malicious actor can observe the timing difference between accesses. This timing difference can be used to deduce behavioral patterns of the cache, leaking information that can be used to access sensitive data that should not be available to such actors [22].



(a) The processor requests data A and finds it in the cache, it is a cache hit.

(b) The processor requests data B and does not find it in the cache, it is a cache miss.

Figure 1: Demonstration of a cache hit and a cache miss.

In a multi-processor system, several processing cores are utilized to run different processes at the same time. This allows more processing power while increasing efficiency in resource utilization. Nonetheless, problems emerge when these processors start to compete for available resources. The most precious resource is memory. When several cores attempt to access and use the memory for their respective jobs, memory space becomes contested. As a result, the probability of a cache miss increases when a processor attempts to access data stored in the shared memory. A higher rate of LLC misses is inevitable, causing higher latency during data retrieval. This difference between a hit and miss latency is hugely problematic. The difference in latency can be large enough to be measurable. One can carefully observe the behavior of the cache system and probe the timing difference at each cache level. An analysis of the probed system by an adversary can deduce the behavior of the shared caches and predict their next move. Such deductions can be used to infer sensitive information and system secrets. The Advanced Encryption Standard (AES) has already been a victim of such an attack [10].

This type of attack is called a timing attack. This is only one example of the ways to exploit this inherent timing issue in multi-cache systems. Instigating conflicting accesses using the multi-process feature of multi-core processors [53] can force a process to reuse a memory location. The instigator is then able to collect information on the targeted process behavior. Another way to obtain information is by tracing the cache's access patterns to access information by an adversary [43][45].

At a time when information is exchanged at staggering rates, securing this information is critical. The world relies on this exchange of information to be trustworthy. Cache at-

tacks that target public-key schemes [57][25][29] and symmetric cryptography [71][26][40] challenge this trust. Secrets must be kept safe when exchanged between systems and cache attacks are putting such safety at risk.

There is no doubt that the shared cache in a multi-core, multi-cache system is vulnerable. Not by design shortcomings, but by the way it operates. A fundamental solution must be devised to secure it. A detection system must be used to predict and detect such attacks. Furthermore, a mechanism to mitigate and eliminate the advantage given by Logical Side-Channel Attacks (LSCAs) to adversaries must be implemented.

LSCAs are particularly dangerous. They are not exactly a design flaw. They are methods to deduce what the system is doing and what it is doing it with at a given stage of execution. Their danger comes from the exploitation of a system's normal expected behavior rather than an abnormal behavior [71]. Therefore, they are extremely hard to predict, detect, and effectively fight against.

Typically, L1 and L2 caches are reserved for private use by their respective cores. Thus, they are safe from any tampering by an adversary. However, when it comes to shared caches, especially the LLC, the cache is more prone to intrusions. The LLC's purpose is to increase efficiency by sharing memory space among processing cores. This core feature is the root of the LLC's information leakage vulnerability [7] when processes start to compete for resources.

By sharing the same cache memory space, processes are unavoidably able to discern another process' use of the cache. In a modern system where processes access the cache at high rates, it is bound that a process' use of one portion of memory will collide with that of another, allowing both to observe the patterns of such access [18]. This event is called **cache contention** and, in a multi-processor system, it is inevitable [85]. Figure 2 presents a possible cache contention scenario, where processes A and B share the same cache.

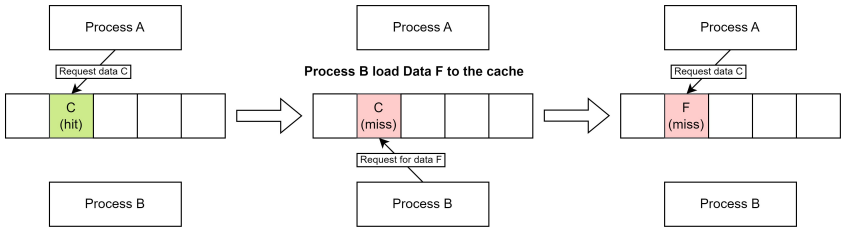


Figure 2: Demonstration of a possible cache contention scenario.

Cache contention refers to the interference introduced by one process running on one core to the cache access of another process running on a different core. Conflicting accesses will occur in shared cache locations (cache set/cache index). Any information regarding the cache operations before, during, and after such conflicting accesses is considered leaked information. Observing cache behavioral patterns when cache contention occurs is particularly valuable to an adversary. Methods have been developed to take advantage of cache contention to leak cache information whether by measuring cache access timing [70], instigating conflicting accesses [78, 30], or by simply tracing accesses to the LLC [4].

Such methods, and many others, pose a grave danger to multi-core, multi-cache systems of all kinds. They are powerful tools when used properly by system adversaries. It is becoming imperative to equip shared caches with the appropriate mitigation strategies to effectively deal with them.

The focus of this work is to explore ways to devise such strategies. On-line prediction of cache contention can be used to predict access patterns that can be exploited by LSCAs and create early warning for their presence during execution. This detection method allows the implementation of mitigation mechanisms that interfere with those contention-based access patterns and disrupt them before they can be exploited.

1.2 Problem Formulation

Logical Side-Channel Attacks primarily target system features to extract sensitive information [85]. Multi-level caching uses several levels of cache designed to optimize the system's data propagation. By using large caches shared among the processing cores, the speed gap between the main system memory and the multiple cores can be narrowed at lower design costs. However, because modern systems rely on multi-processing, the running processes will attempt to use the shared cache at the same time, and thus competition over resources is inevitable, resulting in cache contention.

This can cause any process running on the system to exhibit certain cache access patterns that can be used to leak information. There is a finite number of possible interactions between processes trying to access the same cache resource. By analyzing these interactions, it is possible to identify which cache access patterns pose the leakage threat and which access patterns are safe. Once those patterns are identified, methods of detection and tracking of those patterns can be devised. Such method can be used to create a hardware architecture able to operate and detect those patterns during run-time.

Detection is half of the story. Detecting access patterns that can be taken advantage of by LSCAs does not prevent them. A method of mitigating and eliminating those patterns from occurring is needed. This method must take into consideration the already large computer systems in place. Thus, a mitigation method must be added to the cache system with as little additional hardware as possible.

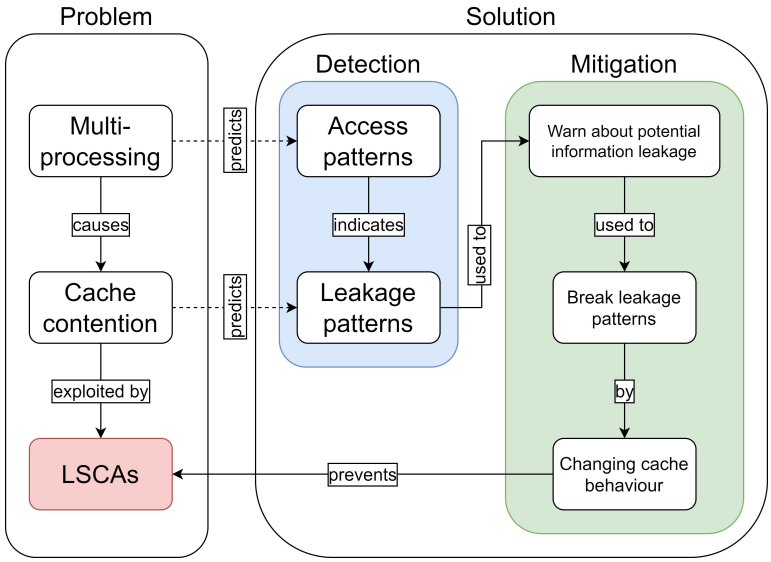


Figure 3: Problem and solution formulation.

Figure 3 presents the problem addressed in this thesis and the steps taken for formulating the solution as follows:

- The problem stems from the design aspects of modern multi-processing schemes. When multiple processes compete for resources in the shared caches, cache contention occurs, which in turn can be exploited by LSCAs.
- The multi-processing features, on one hand, can inform about the predictability of access patterns to the cache by competing processes. On the other hand, cache contention allows us to create predictable leakage patterns in the cache.
- By correlating access patterns from multi-processing features and leakage patterns caused by cache contention, we can create early warnings concerning potential information leakage.
- Such warnings can be used to break leakage patterns by changing the cache behaviour during execution, preventing LSCAs from exploiting cache contention to leak information.

The key questions this thesis sets out to answer primarily revolve around the relationship between LSCAs and cache contention. The thesis attempts to address how such relationship is exploited by adversaries, how to detect LSCAs using predictable cache contention behavior, and how mitigation methods can be implemented to counteract this relationship. Is it possible to use cache contention to predict LSCAs before they manifest? If cache-contention-based detection is possible, how effective can it be? What methods of mitigation can be used to defend the cache? What is the cost of such mitigation?

1.3 Contributions

This thesis aims to address the problem described in the previous section. The thesis proposes a novel detection method as well as two mitigation mechanisms to detect and mitigate LSCAs. The thesis goes beyond the state of the art by presenting the following contributions:

- Formulate an accurate leakage detection model with high positive correlation between cache access patterns and potential information leakage in the cache.
- Design and implement a cost-effective Cache Logical Side-Channel Attacks Detector (CLD) architecture utilizing run-time cache output signals with low overhead in power and area.
- The design of a specialized Content-Addressable Memory (CAM) is introduced and implemented with built-in re-mapping redundancies capable of eliminating predictable primitive polynomial Linear Feedback Shift Register (LFSR) trace.
- A non-invasive, access re-mapping system for mitigation of cache attacks called Secure Cache Alternative Address Table (SCAAT).
- Recursive Redundant Secure Cache Alternative Address Table (RSCAAT), a vulnerability-free mitigation mechanism with a robust control logic and higher reactivity to cache behavior.
- Evaluation of the detector and mitigation components using benchmarks chosen from several benchmark suites showing performance while introducing significantly higher levels of security to the cache by accurately eliminating leakage patterns as well as recursively remapping vulnerable accesses.

1.4 Thesis Organization

The remainder of this thesis is organized as follows:

- Chapter 2 gives introductions into the main topics of this thesis and familiarize the reader with basics of cache memories, multi-level caching, and Logical Side-Channel Attacks.
- Chapter 3 presents previous research and related works and discusses them and their relevance to the core concepts and ideas explored during the creation of this work.
- Chapter 4 presents the experimental set up. It shows the main components of the environment used in a step-by-step setup implemented to generate the final results.
- Chapter 5 exhibits the work done for the detection of Logical Side-Channel Attacks and presents the novel methodology used to create an online leakage detector.
- Chapter 6 thoroughly presents two novel Logical Side-Channel Attacks mitigation mechanisms that make use of previous leakage detector to ease the effects of attacks on the cache.
- Chapter7 presents and discusses the results of the experiments done with both the leakage detector and mitigation mechanisms.
- Chapter 8 presents the conclusion of this work and further discussion about future work.
- Appendix 1 presents the publication of SCAAT: Secure cache alternative address table for mitigating cache logical side-channel attacks [60].
- Appendix 2 presents the publication of CLD: An accurate, cost-effective and scalable run-time cache leakage detector [62].
- Appendix 3 presents the publication of RSCAAT: A recursive mechanism to mitigate cache side channel attacks [63].

2 Background

This chapter presents concepts used throughout this thesis. It presents an overview of cache memory systems and their role in modern computer architectures, cache hierarchies, and cache mapping techniques. The chapter then introduces Logical Side-Channel Attacks, with particular emphasis on cache-based attacks. Different categories of cache attacks are discussed alongside the mechanisms they use to exploit cache behavior and infer sensitive information. Finally, existing models used for analyzing cache attacks is reviewed, providing the foundation for the detection and mitigation techniques presented in later chapters of this work.

2.1 Cache Memories

Cache memory, also called cache, is a type of semiconductor memory used to store the frequently accessed and used instructions and data. The primary benefit of cache is to allow the CPU quicker access to the most important instructions and data during execution [36]. The cache itself is a high-speed Random Access Memory (SRAM) module expansion to the computer's slower, typically Dynamic Random Access Memory (DRAM), Main Memory. The cache holds a copy of the most accessed information stored in the main memory. In order to have the cache as close as possible to the CPU cores, it is typically small in size and capacity. Nonetheless, the cache access latency is also very small, allowing the CPU cores to locate and retrieve the stored data within the cache at high speed. Modern systems utilize several levels of caches at a time [36].

When the CPU requires access to an instruction or to data, the on-chip cache is the first checked for the information. If the information is not located in the on-chip cache, higher level caches are checked. If such information is not present in the cache hierarchy, it is retrieved from the main memory. Cache memory is important as it provides an efficient way for the CPU to retain information close by and guarantees minimal latency when retrieval is needed [36]. Aside from this feature, the cache is a valuable point for evaluating computer performance. A high cache hit-to-miss ratio is indicative of a high performance system [66].

2.1.1 Cache Hierarchy

Caches come in all sizes. They have different uses and are placed to serve different purposes. Figure 4 shows a typical multi-level cache arrangement. The first cache to be found is the Level-1 cache, or L1 cache. The L1 cache is the primary cache that the processing core directly communicates with. It is where the CPU expects to find both instructions and data during execution. Modern L1 caches, however, are not a single memory module. In fact, most processors have two on-chip L1 caches: the Instructions Cache (L1 I) and the Data Cache (L1 D) [33]. As the names suggest, these specialized caches store distinct information. This allows customization of those caches to their respective information type. On one hand, instructions are sequential information that usually loaded into the cache in order. On the other hand, data changes at the different stages of execution, making one word of data, typical 32-bit long, less frequently reused compared to it's instruction word counterpart. By separating the instructions from data, cache efficiency is increased. In this configuration, the processing core is able to look for information much faster compared to a unified instructions and data L1 cache [7].

The Level-2 cache, or L2 cache, is larger than L1 cache, but is slower. It is unified with both instructions and data in the same memory space. While L1 cache is always inside the processing core core, L2 cache can be either embedded on the CPU chip or can be a

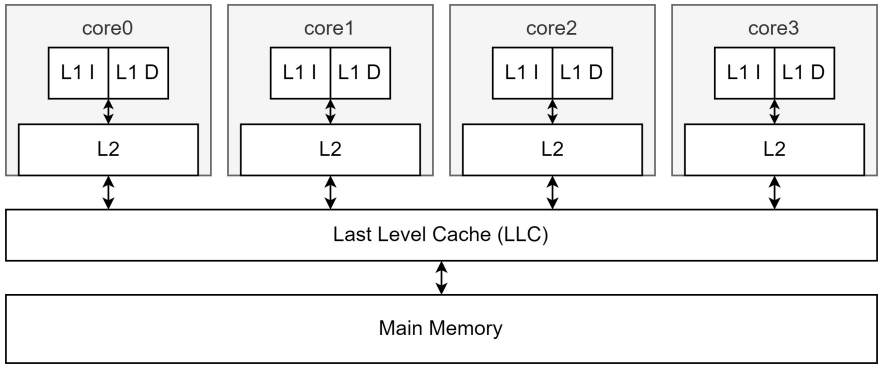


Figure 4: Multi-level Cache Hierarchy.

separate chip connected to the CPU thought a dedicated high speed bus [36]. This allows the L2 cache to be connected to the core while not being slowed down by the otherwise busy traffic on the main system bus.

The Level-3 cache, or L3 cache, is at least double the size of L2. It is much more spacious, but far slower. The L3 cache is usually is the highest cache level. For that reason it is commonly refereed to as the Last-Level Cache, or the LLC. The LLC is where several cores, and in some cases all the cores, share the same memory space. Figure 4 shows the multi-core, multi-level cache configuration used as the experiments' target system, discussed in Section 4.1.

2.1.2 Cache Mapping

Cache Mapping refers to the internal structure and organization of the cache. It is the technique that is used to bring data into the cache space. The internal structures of the cache is continuously developed towards better performance to keep up with the fast evolving processing systems. In order to organize the data properly, the cache is divided into several data structures. A **cache word** is composed of 32-bits or 64-bits of data. Several cache words are organized into a **cache block**, the primary unit of data transfer between the cache and main memory. A **cache set** is the physical space inside the cache where one or several cache blocks are stored and accessed. Each cache block is referenced using a cache address [33].

Figure 5 shows how a cache address is divided into different parts that describe where the data should be stored in the cache and later can be retrieved from. Typically the address is made up of a **tag** that serves as a unique identifier for each cache block, a cache **index** that identifies which cache set a block can be found, and an **offset** that specifies which word inside that block is being requested. The cache memory space itself is divided into a large number of cache sets that are structurally organized based on different mapping schemes.

To organize data inside the cache, it traditionally has three different mapping schemes:

- **Direct mapped cache** was among the first mapping techniques to emerge for organizing the cache [37]. To this day, lower levels of cache across different types of processors still use this technique. Direct mapped cache has a simple concept: each block of data is stored in exactly one specified cache set. Figure 6a shows how in direct mapping, each cache set contains the tag of the block to identify which block

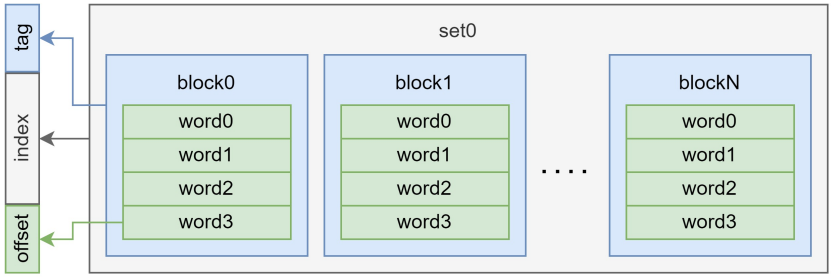


Figure 5: Cache Address.

is being store at the set, a valid bit is used to indicate if the data stored at a given set is valid, and the data itself.

- **Fully associative cache** mapping, shown in Figure 6a, has a similar set structure as the direct mapping, but unlike direct mapping, using fully associative cache allows a cache block to be stored at any available cache set.
- **Set associative cache** mapping, or k-way associative cache mapping, shown in Figure 6b, combines advantages of direct mapping and fully associative mapping [65]. A block can be stored at a predetermined cache set, similar to direct mapping, but a cache set is divided into a number of ways. This allows a block to be stored at different ways, similar to fully associative. Each way can hold one block at a time. Allowing several blocks to be stored at the same cache set.

	tag	v	data
set0			
set1			
set2			
set3			
...	...	...	...
setN-1			
setN			

(a) Structure of a direct mapped / set associative cache.

		tag	v	data
set0	way0			
	way1			
set1	way0			
	way1			
...	...	...	...	...
setN	way0			
	way1			

(b) Structure of a 2-way set associative cache.

Figure 6: Demonstration of cache organization schemes for set-associative caches.

For this work, a set associative cache mapping is used for higher levels of cache in the experiments' target system, discussed in Section 4.1. This is largely for two reasons. **a)** Set associative cache mapping combines advantages from both direct mapping and fully associative mapping. If Set associative mapping fails in a given test environment, it is indicative that other mapping techniques may fail in the same environment. **b)** Set associative cache is very common at higher levels of cache, which is where also the majority of side channel attacks occur in the cache system due to cache to contention. And as such, using a set associative cache for the experiments conducted for this work creates a more realistic environment where the detection and mitigation mechanisms used in this work can be tested.

2.2 Logical Side-Channel Attacks

A Logical Side-Channel Attack is a security exploit that targets specific physical characteristics of a chip or a system to extract information. Such characteristics include timing, power, and the state of the system at different stages of execution. They are a very dangerous threat to systems that implement cryptographic algorithms. This danger comes from the fact that these side channels do not target a system design or the software used to execute on this system. Rather, they aim to collect and/or influence the behavior of the system. By measuring and exploiting a system's behavior, information is inevitably leaked.

2.2.1 Types of Cache Attacks

Depending on the side-channel information leakage being exploited by an attacker, cache attacks are commonly categorized into three types: time-driven, trace-driven, and access-driven. In a time-driven attack, the source of information leakage is the execution time of a process and its cache accesses' timing. A cache set can be targeted to induce a change in the cache behavior. Figure 7 demonstrates an example scenario of a time-driven attack called the Evict+Time attack [70]. Figure 7 illustrates how the attacker first evicts data from cache set **s1** and then measures the execution time of the target process. If the target subsequently accesses data mapped to **s1**, the eviction causes a cache miss and increased access latency. The observed timing difference allows the attacker to infer the target's use of cache set **s1**. Other examples of such attacks are collision timing attacks [12] and Bernstein's attack [10].

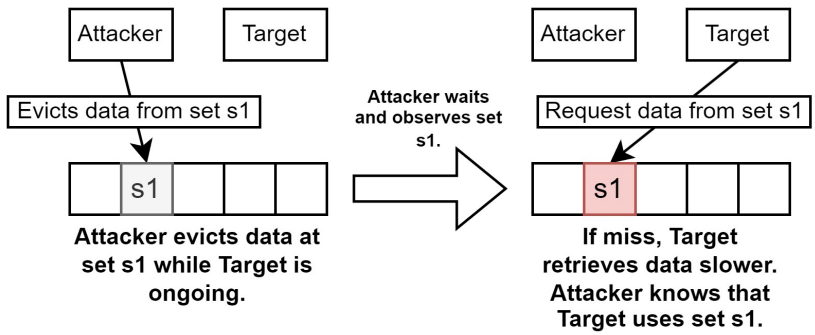


Figure 7: Example scenario of an Evict+Time attack.

In a trace-driven attack, the attacker records cache traces of a target process. Using these traces, sensitive information is deduced and collected. This can be done by generat-

ing traces of static and dynamic power [24] or system monitors [4] while a target process is being executed.

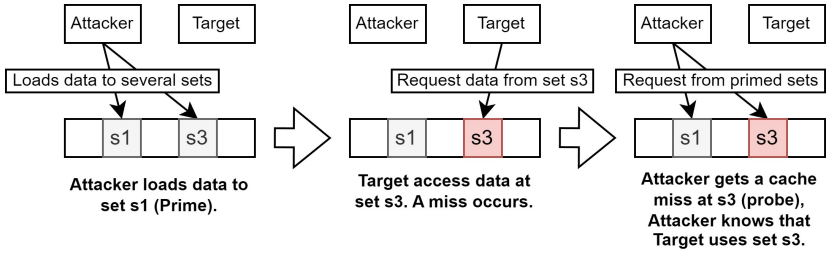


Figure 8: Example scenario of a Prime+Probe attack.

In an access-driven attacks [70, 78, 30], an attacker process is used to observe the cache access pattern of a target process. In this attack, the attacker collect information regarding sensitive cache addresses accessed by the target process. This can be done by measuring cache hits and misses of a given location in the cache. If an access by a target process removes cache data of the attacker process, the attacker can collect information on the access pattern of the target process. This is done by correlating the cache hits and misses (of both attacker and target processes) with the time required for a given location in memory to be accessed.

Figure 8 demonstrates an access-driven attack called the Prime+Probe attack [70]. The attack is performed in three steps: the attacker process loads data into predetermined sets **s1** and **s3**. This step is the priming step. The attacker waits for the target process to access cache sets **s1** and **s3**. The attacker accesses sets **s1** and **s3** after the target process. By observing the behavior of the cache, the attacker can deduce the access pattern of the target process. This can lead to secret information, such as locations of cryptography libraries, to be collected by the attacker. Examples of such attack include the Prime+Probe, Flush+Reload [78], and Flush+Flush [30] attacks.

2.2.2 Modeling Cache Attacks

Information leakages, using the previously mentioned cache attacks, can be modeled using several approaches. The most straight-forward approach is to simulate attacks on a given system to evaluate the information leakage vulnerabilities of the system [17]. However, simulation comes at considerable disadvantages. Extensive knowledge of the attack process and protection against such attacks is needed. Due to the complexity of simulating hardware and software together, this technique can prove very expensive and time consuming.

Due to the highlighted drawbacks of simulation in security verification, works apply formal verification techniques to model different cache attacks and verify cache attacks' occurrences in a target system. Work presented in [21] described an exhaustive set of 28 attack formulas in a three-step model. The formulas were further analyzed in [22]. Table 1 summaries these attacks using a sequence of three accesses. Each access is performed by an actor (attacker or victim). Furthermore, each access is also categorized by knowledge possessed by attacker regarding the victim. In Table 1, column ID presents the id of each formula. Column $attack formula$, demonstrates the sequence of accesses performed by actors. Accesses performed by each actor are defined as follows:

- $V1$: victim accesses a location known to the attacker;

Table 1: Attack formulas [21] analyzed in this work.

ID	attack formula	*	ID	attack formula	*
1	$Vx \to Ar \to Vx$	-	15	$Vx \to Vx \to Ar$	d
2	$Vx \to Vr \to Vx$	-	16	$Ar \to Vx \to Vr$	d
3	$Ar \to A1 \to Vx$	-	17	$Vr \to Vx \to Vr$	d
4	$Vr \to A1 \to Vx$	-	18	$Vx \to Vx \to Vr$	d
5	$A1 \to A1 \to Vx$	-	19	$Ar \to Vx \to A1$	e
6	$V1 \to A1 \to Vx$	-	20	$Vr \to Vx \to A1$	e
7	$Vx \to A1 \to Vx$	a	21	$A1 \to Vx \to A1$	f
8	$Vx \to A1 \to Vx$	b	22	$V1 \to Vx \to A1$	-
9	$Vr \to V1 \to Vx$	b	23	$Vx \to Vx \to A1$	e
10	$A1 \to V1 \to Vx$	b	24	$Ar \to Vx \to V1$	b
11	$V1 \to V1 \to Vx$	b	25	$Vr \to Vx \to V1$	b
12	$Vx \to V1 \to Vx$	c	26	$A1 \to Vx \to V1$	-
13	$Ar \to Vx \to Ar$	d	27	$V1 \to Vx \to V1$	c
14	$Vr \to Vx \to Ar$	d	28	$Vx \to Vx \to V1$	b

* Known attacks: **a**) Evict+Time[53]; **b**) Cache Collision[12]; **c**) Bernstein[10]; **d**) Flush+Flush[30]; **e**) Flush+Reload[78]; **f**) Prime+Probe[53].

- Vx: victim accesses a location unknown to the attacker;
- Vr: victim flushes a location unknown to the attacker;
- $A1$: an access by the attacker to a cache location;
- Ar: a flush by the attacker of a cache location.

However, these models offer no description of the resulting behavior of the cache (hits or miss) when such accesses are performed. Additionally, each of such access sequences do not reveal whether such behavior is indicative of information leakage. Furthermore, the model makes several assumptions that can be proven difficult to ascertain. More specifically, the assumptions that **i)** the attacker's behavior is known and/or detectable and **ii)** what knowledge the attacker possesses regarding the access behavior of the victim.

2.3 Chapter Summary

This chapter introduced the principles of cache memory, the organization of multi-level cache hierarchies, and common mapping schemes, with emphasis on set-associative caches due to their prevalence in modern processors. The chapter then examined concepts required to understand cache-based side-channel vulnerabilities and types of cache Logical Side-Channel Attacks used to exploit such vulnerabilities. Finally, approaches for modeling cache attacks were discussed highlighting modeling techniques along with their limitations in representing attacker behavior and interpreting cache activity. These insights establish the foundation for the detection and mitigation strategies explored in the later chapters of this work.

3 Related Works

Addressing cache vulnerabilities is a pressing concern and a central priority in hardware security. A wide range of defensive strategies have been proposed to increase cache security with varying degrees of success. This section presents these efforts in two subsections. Section 3.1 discusses several cache attack detection strategies. Section 3.2 evaluates mitigation strategies by grouping them into three architectural categories: partition-based, randomization-based, and adaptive mitigation strategies.

3.1 Cache Attack Detection

In recent years, a wide range of cache attacks [10, 53, 31, 79] were presented. The central focus of detecting cache attacks is the characterizing of system behaviors that lead to information leakage. To this end, methods of qualifying and quantifying the vulnerability of the cache to information leakage were proposed [16, 44, 83, 80]. Some of those approaches, for example, used information-flow analysis and static cache analysis [44] for quantifying cache side-channels. In [16], a software based approach is used to quantify the amount of information that can leak through memory performance and cache-access statistics. Moreover, [44] uses a combination of quantitative information-flow analysis and static cache analysis for quantifying cache side-channels.

However, for LSCAs, the case is a bit different. The system behavior is not compromised since those attacks do not rely on side-channels that normally do not interfere with system behavior. In fact, it is not unusual that an attack is only detected after the attacker has successfully completed their attack. Therefore, a reliable and accurate run-time detection of information leakage is crucial for the security of modern digital systems.

One way to detect such attacks is by monitoring the cache behavior. A few publications have reported offline and online cache monitoring for security purposes. On one hand, offline monitoring [21, 22, 35] comes with little overhead and no hardware implementation. Detection of attacks using such approaches relies heavily on the coverage of valid attack scenarios and are only effective after attacks had occurred. Online monitoring approaches [19, 5, 9, 13, 51, 84], on the other hand, yield higher attack detection than offline monitoring. These monitoring approaches collect information for internal hardware monitors, such as CPU Hardware Performance Counters (HPCs) [19, 5, 9], to detect different types of LSCAs. However, the overhead in cost and performance of implementing such approaches in a number of devices can prove limiting. Nonetheless, the aforementioned approaches often only target a specific attack and its variations. They require additional offline analysis to detect attacks, further adding additional overhead in cost and performance. One run-time attack detection device, A Lightweight Detector for Cache ATtacks (LiD-CAT) [56], used models to characterize the information leakage by exploiting common cache attacks' characteristics. However, such approach was shown to yield a high percentage of false positives.

Some security verification techniques modeled different cache attacks. The most straightforward approach is to simulate attacks on a given system to evaluate the information leakage vulnerabilities of the system [17]. While security verification using simulation of attacks can give important insight on system security, simulation comes at considerable disadvantages. Extensive knowledge of the attack process and protection against such attacks is needed. Due to the complexity of simulating hardware and software together, this technique can prove very expensive and time consuming. Furthermore, the effects of attacks and protection methods on the system's behavior upon the discovery of new attacks may become unpredictable. Work presented in [21] described an exhaustive set of

28 attack formula in a three-steps model, were further analyzed in [22] and verified in [27]. These attack formulas are described using a sequences of cache accesses, identification of an attacker or victim that preforms these accesses, and knowledge that the attacker possesses regarding the victim. However, this model offer no description of the resulting behavior of hits and misses that the cache experiences when such attacks are performed. Furthermore, this model does not offer an evaluation on whether such behavior is indicative of information leakage. Additionally, the model makes several assumptions that can be proven difficult to ascertain. More specifically, the assumptions that **i)** the attacker's behavior is known and/or detectable and **ii)** what knowledge the attacker possesses regarding the access behavior of the victim.

3.2 Mitigation of Cache Attacks

To address the increase threat of LSCAs, several mitigation approaches were introduced. Each approach provides a unique insight into the complexity of LSCAs mitigation.

3.2.1 Partition-Based Mitigation

Partition-based mitigation aims directly at limiting, or eliminating, cache contention between a secure process and an attacker process. This is done by separating where each of these processes can access, i.e. a secure process and an attacker process are assigned predetermined available cache sets that can be accessed by either process. These cache sets can be allocated either statically [81, 82, 34] or dynamically [42, 23, 77, 20, 46]. Partitioning the cache is an excellent method for preventing the majority of external interference. It is very effective against leakages resulting from cache contention caused by flush operations and evictions. Nevertheless, it does not prevent internal interference and leakages resulting from the process' own access patterns.

For example, the statically partitioned cache presented in [34] and the dynamically partitioned SHARP cache [77] are vulnerable to variations of Cache Internal Collision attacks [22, 10]. Chameleon Cache presented in [69] dynamically switches how the cache is organized and divided into ways, effectively partitioning the cache under certain conditions. Similarly, SassCache [28] controls which cache set belongs to which security domain, segmenting the cache and rearranging the cache ways to isolate high-security loads from interference. Both Chameleon Cache and SassCache show strong resilience to attacks using dynamic and deterministic cache partitioning. Nonetheless, cache partitioning, in general, inherently introduces significant cache performance degradation [47]. This is largely due to under utilization of available cache space, the restricted number of cache sets available to each process at a time, and the time that is required to dynamically re-partition the cache (for dynamically partitioned caches).

3.2.2 Randomization-Based Mitigation

Randomization-based mitigation targets disassociating the cache internal state (locations, cache miss, cache hit, etc.) and the cache operations (read, evict, flush, etc.) from any observable behavior (access patterns, timing, etc.). To achieve randomization, most approaches use a random number generator [52] and a look-up table to keep track of the re-mapped location to avoid coherency issues. Randomization can also be achieved using encrypting such as the case for CEASER [55], MIRAGE Cache [58], ScatterCache [76], and GIR-cache [67].

Such designs use randomization in different ways to protect about information leakage. CEASER employs a Low-Latency Block Cipher (LLBC) to translate the physical line address into an encrypted one, ScatterCache's keyed mapping scatters cache lines unpre-

dictably across cache sets, while GIR-cache uses indirect replacement with randomized placement to break predictable access patterns.

While randomization-based approaches are simple in principle, they come with considerable disadvantages. Look-up tables that track the changes to the access pattern to maintain coherent access to the cache can become extremely large when used with large-sized caches such as the LLC [41] and almost always leave the cache vulnerable to some variations of side-channels attacks [47]. Caches that use cryptographic randomization can be strong against Prime+Probe attacks but do not prevent Flush+Reload and Flush+Load attacks when the cache is shared. Furthermore, cryptographic randomization requires additional hardware to facilitate mapping/remapping, adding area overhead and causing performance degradation.

3.2.3 Adaptive Mitigation

Partition-based mitigation relied on the knowledge of which data requires secure locations. Determining what data needs to be stored in secure locations is achieved at the software level. Such software approaches are vulnerable to cache contention and require offline analysis of cache access patterns. Randomization-based mitigation scrambles the access pattern to the cache but introduces performance and area overheads. However, to achieve better performance and security, adaptive mitigation aims to protect sensitive data (similar to partition-based mitigation) and interfere with the access patterns (similar to randomization-based mitigation) at run-time.

This comes with two advantages: First, caches that use this technique operate only to protect sensitive data from cache contention, lowering performance overheads and concentrating the impact of mitigation to data that is under threat. The Catalyst cache [46], for instance, classifies data as software-managed secure pages that can only be accessed from secure cache partitions, resulting in an acceptable performance overhead. While this is reminiscent of partition-based mitigation, its fundamental principle relies on the distinction between data that needs protection from data that does not. This approach still relies on software, which may prove too complex to design according to the needs of the cache and hard to implement in some circumstances [22].

Second, by only mitigating attacks on sensitive data, more flexible hardware solutions can be adopted, potentially targeting a wider variety of threats, while maintaining acceptable overheads.

Additionally, using hardware solutions circumvents the need for software support that can be costly in both design-time and design-cost. This requires architectural infrastructure to facilitate the detection of threats and mitigating them. Detecting cache information leakages at run-time manifests both advantages. A cache leakage detector, can be used to detect cache information leakages, preventing any exploitation of cache side-channels by adversaries. Detection of such threats enables the use of mitigation mechanisms upon detection of threats. Using attack and cache information leakage detection systems (also referred to as monitoring systems) gives way to the development of secure caches without the need to allocate secure partitions or require area-expensive look-up tables. Monitoring systems can be incorporated to any cache design, making them a flexible solution for detection of adversarial activities as opposed to secure cache architectures with fixed security features that generally come at a high area cost and performance overhead [75, 52].

Table 2: Comparison of representative cache attack detection and mitigation approaches.

Work	Type	Approach	Assumptions	Overhead	Main Limitation
HPC-based detectors [19, 9, 5]	Detection	HPC monitoring	Attack signatures visible in HPCs	Low–Moderate	Attack-specific; requires training
LiD-CAT [56]	Detection	Leakage characterization	Leakage patterns accurately modeled	Moderate	High false-positive rate
Attack Formula Models [21, 22, 27]	Verification	Formal attack modeling	Attacker behavior known	None	No mitigation; strong assumptions
SHARP [77]	Mitigation	Dynamic partitioning	Security domains known	Moderate	Reduced cache utilization
SassCache [28]	Mitigation	Cache partitioning	Security domains known	Moderate	Performance degradation
CEASER [55]	Mitigation	Cryptographic randomization	Secret mapping unknown	High	Limited against shared-cache attacks
ScatterCache [76]	Mitigation	Randomized mapping	Secret key protected	Moderate–High	Additional mapping logic
Catalyst [46]	Mitigation	Adaptive secure pages	Sensitive pages identified	Moderate	Requires software support
Proposed Work	Detection + Mitigation	Run-time leakage detection	No attacker knowledge	Low	Evaluated on modeled leakages

3.3 Comparison of Existing Approaches

The reviewed detection and mitigation techniques address cache side-channel attacks from different perspectives and therefore exhibit different trade-offs in terms of security coverage, implementation complexity, and system overhead. Detection-oriented approaches primarily focus on identifying attack activity or information leakage patterns, whereas mitigation-oriented approaches seek to prevent or limit information leakage through architectural modifications. Table 2 summarizes the main characteristics of representative approaches discussed in this chapter.

As shown in Table 2, existing solutions typically focus on either attack detection or attack mitigation. Detection approaches often require prior knowledge of attack characteristics or rely on offline analysis, whereas mitigation approaches generally incur performance, area, or software-support overheads. These trade-offs motivate the development of lightweight solutions that can provide both detection and mitigation while maintaining low implementation cost.

3.4 Chapter Summary and Research Gap

This chapter reviewed several of the existing approaches for detecting and mitigating Cache Logical Side-Channel Attacks. Detection techniques discussed in this chapter range from offline analysis and formal verification to run-time monitoring using hardware performance counters and leakage characterization models. While these approaches can successful identify suspicious activity or quantify information leakage, they either assume knowledge of the attacker, operate offline after the attacks had already occurred, only target some attack types, or exhibit high rates of false-positive.

Furthermore, mitigation techniques were categorized into partition-based, randomization-based, and adaptive approaches. Partition-based methods effectively reduce cache contention but cause performance degradation due to reduced cache utilization. Randomization-based methods aim at obscuring cache behavior which typically require additional hardware structures and are vulnerable to certain attack variants. Adaptive approaches provide a promising compromise by activating protection mechanisms only when required,

thereby reducing unnecessary overhead.

Despite significant progress, an important research gap remains. Existing detection mechanisms generally focus on identifying attacks without providing integrated mitigation. At the same time, many mitigation techniques rely on offline support, expensive hardware, and incur measurable performance degradation. Furthermore, several approaches target specific attack types rather than the broader problem of cache information leakage. These limitations motivate the development of a lightweight run-time architecture capable of detecting cache information leakage patterns and selectively activating mitigation with minimal area and performance overhead. The following chapters present the proposed architecture and evaluate its effectiveness in addressing these challenges.

4 Experimental Setup

This chapter presents the experimental environment and methodology used to evaluate the cache leakage detection mechanism (CLD) and the mitigation techniques SCAAT and RSCAAT proposed in this work. The experimental setup is shared across these components to ensure consistent evaluation of detection accuracy, mitigation effectiveness, and system overhead.

In this chapter, the overall environment used in this work is explained. Figure 9 shows the different components of experimental methodology used to create the environment and the flow of data needed to create the final results. A target system is used to evaluate the implemented detection and mitigation mechanisms as a baseline reference. A selected set of benchmarks are selected to be executed with a standard multi-processor, multi-cache system. The system behavior is simulated using the Multi2Sim simulator [72], which generates cache activity representative of the selected workloads. A set of experiments are conducted to simulate the behavior of the system in a set environment. A custom cache trace generator is used to extract different sets of data during the execution and generate the required cache traces. The generated traces and cache-state data are then used to drive three open-source Pile of Cores (PoC) LLC cache models [15]: a baseline cache, a cache augmented with SCAAT, and a cache augmented with RSCAAT. Finally, the resulting data are collected and analyzed to evaluate the effectiveness, overhead, and security properties of the proposed detection and mitigation mechanisms.

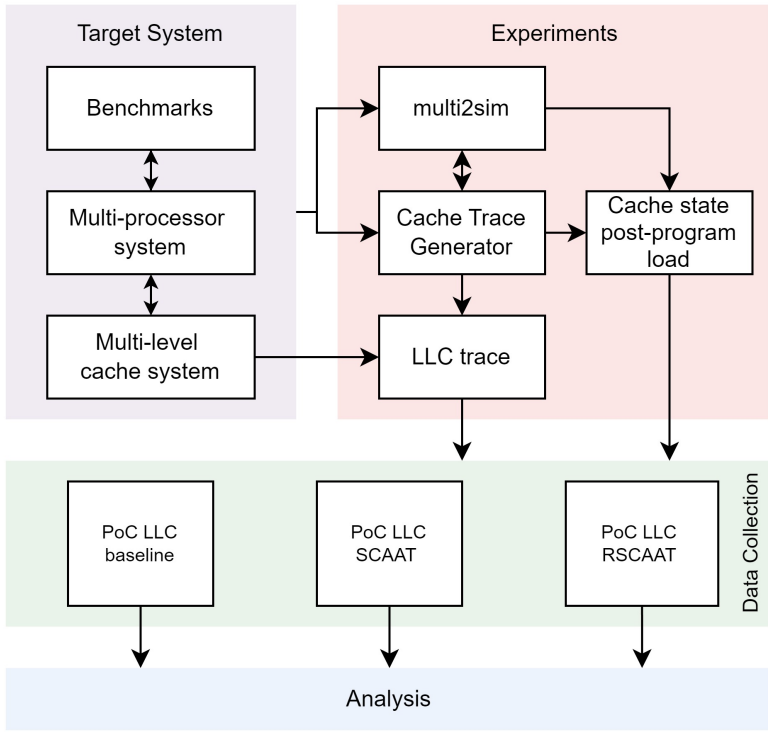


Figure 9: Experimental methodology and data flow.

Table 3: System Configuration [63].

Processor Configuration		
Multi-core setup	4-core, inclusive multi-level caching	
Core parameters	32-bit x86 , 3.3 GHz, multi-threaded*	
Cache parameters		
Cache dimensions	64B cache block, 16 word block	
Private Caches	I/D L1	L2
	32KB direct-mapped 2 cycles latency	256KB 8-way associative 4 cycle latency
Shared Cache (LLC)	4 MB, 16-way associative, 20 cycle latency	

*Only enabled for benchmarks that support multi-threading.

4.1 Target System

For this work, a 4-cores, 32-bit, x86 [1] , inclusive multi-level caching processor is used to simulate the workloads used in the experiments. A 16-way, 4 Megabytes Last Level Cache (LLC) is used as shared cache between these cores. This configuration allows high occurrence potential of cache contention in the LLC. Having several cores running at the same time reflects the expected performance and behavior from a multi-core processor. At the same time, it demonstrates how cache contention occurs in the LLC. However, while the LLC is the target cache to be observed, lower levels of cache are needed. Two private lower level caches are used. The L1 instruction and data caches are both a 32 Kilobyte direct-mapped cache while the L2 is a single 256 Kilobyte 8-way cache. These two lower level caches are included in the architecture as buffer between the LLC and each core. This is important because some benchmarks are not entirely cache hungry and will be sufficiently executed with low access rates to the LLC. Furthermore, a full cache hierarchy will also exhibit a normal and expected data propagation throughout the multi-level cache system, serving to balance the access frequency to the LLC.

Table 3 shows the system configuration used in the target system set up. This system configuration allows 4 workloads to be executed at the same time. The main benefit of this is to have several workloads with different LLC access patterns. By having different access patterns, the LLC would experience different types of cache contention when such workloads' access patterns interact, giving a better look and wider range of results. To maximize this effect, 26 different workloads from the Standard Performance Evaluation Corporation CPU Benchmark Suite (SPEC CPU) 2006 [68], Stanford Parallel Applications for Shared Memory (SPLASH3) [59], and Princeton Application Repository for Shared-Memory Computers (PARSEC) [11] benchmark suites are used. In addition, a combination of 4 unique workloads are assigned to each core respectively. Each combination of workloads would have at most two benchmarks from each suite, ensuring a wide range of access pattern to the LLC while making sure that different benchmarks with different behaviors are used. Each workload is included in at least three different combinations. Using the Python Library random generator functions [73], 25 4-workload combinations matching the aforementioned scheme are generated.

4.2 Experiments

Once the target system and cache system are implemented and all workload combinations are generated, they are fed into the multi2sim simulator. Multi2sim executes all the workload combinations until each of the workloads has either finished completely or has reached a minimum of 1 million LLC accesses. The inputs to each workloads, both as parameters for execution or as input data, are adjusted to execute properly without causing resource hogging. This is necessary due to the absence of an Operating System that would normally prevent one process of hogging the cache. This is only done to workloads that have exhibited this behavior, while other workloads with no such behavior are run with their given recommended inputs in their respective benchmark suite guide. In addition, the multi2sim is configured to start executing each workload at the point when the whole initial program state of all the workloads is loaded.

In order to record the CPU-side accesses to the LLC, a Cache trace program is added to the multi2sim source code. This tracer program records all accesses to the LLC. Namely, it records the type of access (write or read), the accessed address, and an ID for the core that is requesting the access in a trace file.

The generated trace files are validated using an independent Python-based cache replay simulator that re-executes LLC accesses and compares the results with the Multi2Sim output. Validation is performed by matching key metrics, including total LLC accesses, per-core access counts, cache hit/miss rates, and access frequency distributions. If discrepancies are observed, the workload is re-simulated and traces regenerated until consistency is achieved. The validated traces are then used to derive the baseline distribution of cache accesses per location and to extract the cache state after the initial program loading phase.

To evaluate CLD, the access traces generated from the multi2sim simulation are fed into a generic cache shown in Figure 10. Using a SystemVerilog Assertions (SVA) implementation of the formal verification template proposed in [27], CLD's performance and accuracy was evaluated and compared to the state-of-the-art cache attack detector LiD-CAT [56].

Two experiments were conducted on a 16kB 4-way set associative data cache. First, CLD and LiD-CAT were used to detect information leakage and attacks in six benchmarks: SJENG and ASTAR from SPEC CPU 2006 [68], RADIX and VOLREND from SPLASH3 [59], and FACESIM and VIPS from PARSEC [11] benchmark suites. The detection made by SVA was used as the baseline for this experiment. The reason for choosing these benchmarks is to test the CLD with different cache access behaviors that these benchmarks exhibit.

The Second experiment consisted of injecting 10 types of attacks, shown in section 7.1, into a 128-bit key Advanced Encryption System (AES) algorithm at random to determine the detection accuracy of the CLD. In both experiments, 100000 accesses were sampled. Cache access traces for each of the benchmarks were extracted using the multi2sim simulator [72] and then used as input addresses to the cache.

4.3 Data Collection

Once the traces are produced, the cache accesses are fed into a cache model incorporating the detection and mitigation mechanisms. For every combination, all the possible occurrences of cache contention and possible information leakage are detected for one workload at a time. This provided an overview of the patterns of information leakage that each individual workload exhibits.

Once all simulation traces are fed into the cache model, new traces of cache accesses are generated. These new traces contain the accesses to the cache post detection and mit-

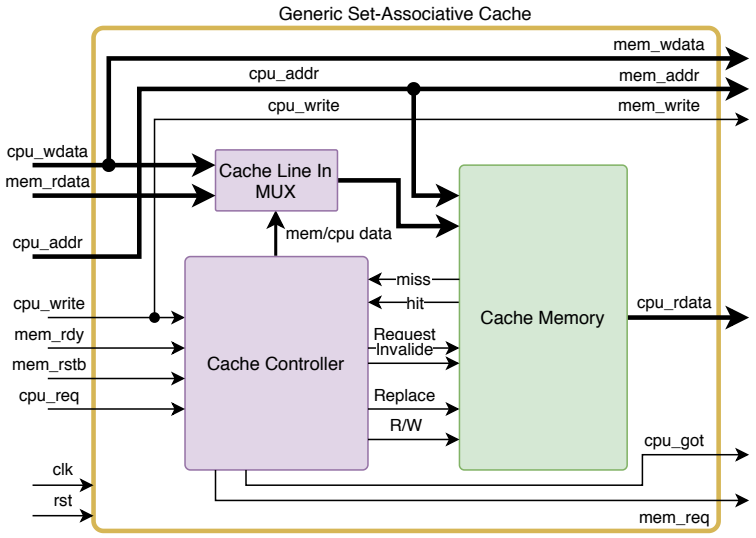


Figure 10: Generic cache architecture block diagram used in this work [63].

igation integration. They are used to generate the distribution of number of accesses per location for each workload after using the mitigation mechanism, allowing more detailed analysis of the behavior of the cache before and after using mitigation was used.

As for the cache model, the open-source Pile of Cores (PoC) IP core cache [15] was used. The reasoning behind this choice is the ability to parameterize the cache its generic set-associative nature. Fig. 10 shows the block diagram of the generic cache architecture used. A Generic Set-Associative Cache design is adopted with a Cache Controller to facilitate the operations of the cache, a Multiplexer (Cache Line In MUX) for reading data from the main memory and writing data from the CPU, and a Cache Memory block to store and retrieve data as needed.

4.4 Data Analysis

Once the required data is collected, the performance of the detection and mitigation is analysed. The results produced by multi2sim and then validated using independent simulation are used as a baseline. Once the detection and mitigation are implemented using the cache model, the results are compared to the baseline. Three metrics are used to evaluate the detection and mitigation: the effects on performance before and after mitigation, the security of the mitigation mechanisms, and the effect on the behavior of the cache once mitigation is activated.

4.4.1 Detection

Leakage detection and accuracy

The leakage detection metric is used to determine the effectiveness of CLD's detection model. By running different workloads exhibiting different cache access patterns and comparing the detection rates of CLD with offline detection, the leakage detection of CLD can be evaluated and its accuracy determined.

4.4.2 Mitigation

Impact on Cache Behavior

The impact on cache behavior metric evaluates the effects of using the mitigation and if such mitigation can be determined to be effective against information leakage. When the workloads are executed, the distribution of accesses per cache location are recorded. This allows to point where each workload performs accesses the most and observe whether conflicts with other workloads are happening and what accesses could cause information leakage. The distribution of accesses per cache location is then compared between baseline execution and execution with mitigation for each workload. This can show the impact on the cache access patterns that each workload exhibits before and after the detection and mitigation of information leakage and the effects it has on the overall cache behavior.

Security

The security metric checks whether the mitigation mechanism is vulnerable for any type of side channel attacks or other types of attack models that might be used to circumvent the mitigation. This metric is measured using known methods and vulnerability of the current implementations and designs of used mitigation mechanisms.

Performance

The performance metric focuses on the hit/miss rate of each of the designs under test. A standalone cache performance is measured under the normal execution conditions with no detection and mitigation. This allows to observe how the cache behaves when different set of workloads are executed at once. The cache contention and their resulting hit and miss rates are recorded for each workload. Since one workload can appear in at least three different combinations of executed workload, as described in Section 4.2, the average of hit and miss rates are taken for each workload for each design, which also includes averages of the baseline. Furthermore, the averages of each workload for each design is then normalized to the baseline average. This is done to give a clear view of the change in the hit and miss rate before using mitigation (baseline) and after using mitigation.

4.5 Chapter Summary

This chapter presented the experimental framework used to evaluate cache information leakage detection and mitigation presented in this work. The target system, benchmark selection methodology, simulation setup, trace collection process, and cache modeling environment were described in detail. Using a combination of realistic multi-core workloads, cache trace analysis, and controlled attack injection experiments, the framework enables the systematic evaluation of both detection accuracy and mitigation effectiveness.

Furthermore, the data collection and analysis process was defined. Metrics used for evaluating performance, security, cache behavior, and leakage-detection are discussed. The next chapter presents the design and implementation of the proposed cache leakage detection mechanism that is evaluated using this framework.

5 Leakage Detection

The detection of side channel attacks has proved essential to increasing the security of cache. Whereas current methods depend on detecting attacks to evaluate a system's security, a more proactive approach is needed to uncover possible side-channel leakage patterns during run-time. To that end, the Cache Leakage Detector (CLD), an accurate, compact and scalable cache information leakage detector was developed. The work presented in this chapter is mainly based on the work done and published in [62].

Fig. 11 shows the CLD development steps. First, leakage and threat models are created. Using the verification template proposed in [27], hit/miss behavior patterns are analyzed and a simplified template of cache information leakage is generated. The leakage template is then applied to design and implement the cache leakage detector. Subsequently, the effectiveness of CLD is validated by presenting cache leakage detection results for a variety of cache access patterns.

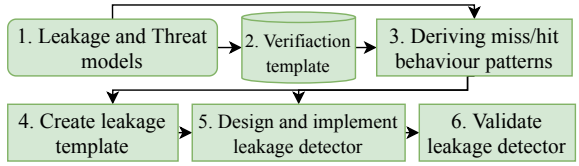


Figure 11: CLD development steps [62].

While the idea of online detector is not new, CLD's architecture brings new features that were not present in previous detection approaches. It proposes a new technique for detecting cache information leakage patterns before such patterns can be exploited by Logical Side-Channel Attacks. An innovative leakage detection mechanism is proposed. Unlike previous approaches, this mechanism operates independently from any assumptions regarding the attackers' behavior. It's high accuracy is attributed to an accurate method that distinguishes a real attack from a potential attack by monitoring cache misses and hits sequences and is verified using SystemVerilog Assertions (SVA).

5.1 Proposed Methodology

The methodology used to develop the Cache Leakage Detector is presented in this section. The leakage and threat models are presented and the verification template is explained. The method of derivation of the possible leakage patterns is presented and the process of creating the leakage template is shown. Finally, the design and implementation of the proposed CLD is discussed.

To develop an accurate leakage detection mechanism, aspects of the system, the threat model, and the leakage model must be defined in an appropriate manner.

5.1.1 Threat Model

The threat model sets forth the vulnerabilities of a given system. This model will describe what can be exploited and used to leak information from the system. The target system is a classical Von Neumann architecture, multiple processors with private caches connected to a shared last level cache (LLC). An Operating System is responsible for resource management and process scheduling. In such system, several processes, each with a unique process IDs (*ProcID*), are being executed concurrently. All processes can access any location of the cache during the process execution. This causes several processes to share the LLC, thereby causing mutual interference. A *Secure Process* (P_S) contains sensitive information and needs to be protected against possible leakage. Any process currently

sharing the cache with P_S is assumed to be an *Interference Process* (P_I). P_I is a process that exhibits a cache access pattern that can lead to information leakage in the execution of P_S as a result of mutual cache interference. The emerging patterns from such mutual interference can be exploited by potential adversaries.

5.1.2 Leakage Model

In order to facilitate leakage detection, a set of leakage scenarios were devised. A leakage scenario is a sequence of three stages: S_1 (setup stage) where the first access initiates a sequence of accesses that can lead to a leakage, S_2 (trigger stage) where a second access causes a mutual interference that leads to a leakage and S_3 (observation stage) is an access where the leakage information can be observed. Each stage of the sequence describes a cache access to the same cache location. These accesses can be performed by either P_S or P_I. The sequence of accesses results in two possible leakage sequences: The first sequence happens when three accesses to same cache location are performed by P_S. This sequence exhibits an access pattern that can be exploited to leakage information by observing the resulting cache behavior. Such leakages can be exploited using the Bernstein's attack [10] and several variations of Cache Collision [12] attack. The second sequence happens when P_I processes interfere with process P_S's execution. Such interference leads to an access pattern by a P_I process that causes cache contention with P_S. Such leakages can be exploited using Evict+Time [53], Cache Collision [12], Flush+Flush [30], Flush+Reload [78], and Prime+Probe [53] attacks and their variations.

Therefore, in both sequences, regardless of what information we have on the attacker, the cache access patterns by processes and the resulting cache contention leads to exploitable information leakage. A leakage can be directly observed whether any information on the attacker is available or not. The access and cache behavior patterns of process P can be indicative of any leakage of information. Therefore, this model relies on obtaining cache output information.

While previous works, such as in [21], require additional information about the attacker's knowledge and intention, the model proposed for CLD removes any assumption regarding an "attacker." Such assumptions are not needed since information leakage from P_S can occur whether an attacker is present or not. Leakage resulting from P_I maliciously interfering with P_S's execution either requires the observation of the behavior of a single process or the resulting cache contention between a targeted process and other processes. Therefore, exploitable information leakage patterns can occur with and without the presence of malicious activity.

5.1.3 Verification template

In this step, leakage scenarios are defined based on the leakage model presented in the previous section. Due to their similarities, verification of the leakage scenarios are adopted from the idea presented in [21]. Nonetheless, [21] assumes that an attacker is present, and defines the actions that an actor can perform depending on the knowledge that attacker has. However, the leakage model presented for CLD, does not make any assumption about the attacker's knowledge or intent. To overcome this, the CLD verification template will use two pieces of information that are readily available. The first is that the P_S process is known. Since P_S is the process being protected and checked for information leakage, it is considered the possible victim – it replaces actions $V1$, Vx and Vr. The second is that any processes other than P_S can cause mutual interference with it, therefore it is considered P_I process and potentially can interfere with process P_S access patterns and it replaces actions $A1$ and Ar.

5.1.4 Deriving miss/hit behaviour patterns

In addition to the issue of attacker's knowledge and intention addressed in the previous section, a significant drawback of the model presented in [21] is that it does not describe the behavior of the cache once accesses are performed i.e. there is no classification of access patterns based on the hit/miss state of the cache.

To address this weakness, an analysis methodology, presented in [27], is adapted to analyze the resulting hit/miss behaviour of the cache when each of the leakage scenarios occurs. As shown in Table 4, each leakage scenario requires three cache accesses, S_1, S_2 and S_3, and each access results in either a cache hit (H) or miss (M). The analysis aims at concluding which hit/miss combination can reveal a possible pattern of accesses that results in information leaking from the cache. As shown in Table 4 each leakage scenario exhibits 8 possible hit/miss combinations which can be accesses by S and I equivalent to an access by P_S and P_I respectively. For instance, SSI is equivalent to a sequence of two access by P_S and one access by P_I. Analysis in [27] classifies each scenario into one of the following three groups:

- Group I (*Invalid*) ; the scenario leads to an invalid hit/miss combination.
- Group $\overline{Attack}$; the scenario leads to a valid hit/miss combination but no exploitable leakage pattern.
- Group $Attack$; the scenario leads to a valid hit/miss combination and an exploitable leakage pattern.

Table 4: Hit/miss combination of all leakage scenarios [62].

S_1	S_2	S_3	Leakage Scenarios							
			SSS	SSI	SIS	SII	ISS	ISI	IIS	III
M	M	M	G_U	G_S	G_U	G_S	G_U	G_S	G_U	G_S
M	M	H	G_U	G_U	G_S	G_S	G_S	G_U	G_S	G_S
M	H	M	G_U	G_S	G_U	G_S	G_U	G_S	G_U	G_S
M	H	H	G_S	G_S	G_S	G_S	G_S	G_S	G_S	G_S
H	M	M	G_U	G_S	G_U	G_S	G_U	G_S	G_U	G_S
H	M	H	G_U	G_U	G_S	G_S	G_S	G_U	G_S	G_S
H	H	M	G_U	G_S	G_U	G_S	G_U	G_S	G_U	G_S
H	H	H	G_S	G_S	G_S	G_S	G_S	G_S	G_S	G_S

* S and I are equivalent to an access by P_S and P_I respectively. SSI is equivalent to a sequence of two accesses by P_S then an access by P_I.
$Unsafe$ group is indicated by G_U inside a red square; $Safe$ group is indicated by G_S inside a green square.

For attacks that are classified in Group $Attack$, all the possible hit/miss combinations of each leakage scenario are tested with using the P_S and P_I sequences described above. Each scenario can be classified into one of the following two groups:

- Group $Safe$ (G_S); A scenario with the resulting hit/miss combination does not lead to an exploitable leakage pattern.
- Group $Unsafe$ (G_U); A scenario with the resulting hit/miss combination leads to an exploitable leakage pattern.

Table 4 shows which hit/miss combination of each leakage scenario belongs to the $Unsafe$ group and which hit/miss combination belongs to the $Safe$ group.

5.1.5 Creating cache leakage template

By combining the results of the hit/miss combination of each leakage scenario from Table 4 and the verification template, a leakage template that only requires immediately available information such as the *ProcID* and the hit/miss output of the cache can be created. For example, let $event_x$ be an event of three accesses to the same cache location L by the same process $ProcID_x$ result in three consecutive misses. If $ProcID_x$ is a secure process, then the leakage scenario in question is SSS. In this case, $event_x$ indicates that a possible cache information leakage scenario at location L that can be exploited by Berstein's or cache collision attacks (e.g attack formulas 11 and 12 in Table 1). The leakage template further minimizes the required information (cache hit/miss and process ID) for leakage detection while encompassing all the possible scenarios that can lead to exploitable information leakage. Using this model, a run-time, compact, and scalable cache leakage detector can be designed.

5.2 CLD design and architecture

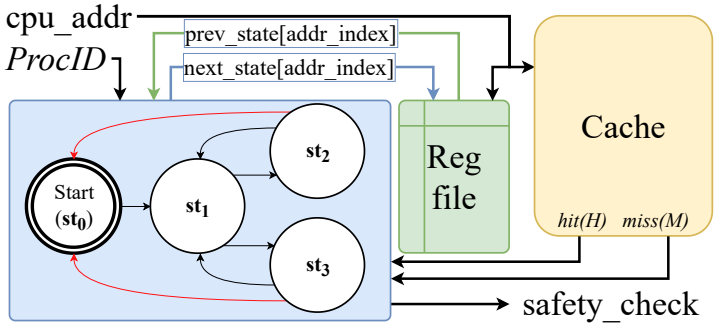


Figure 12: The state transition diagram of the CLD [62].

Since the *ProcID* and the hit (H) and miss (M) signals are readily available at access time during run time, those inputs can be used to construct a Finite State Machine (FSM) that checks for possible access patterns that can lead to information leakage. The leakage template described in the previous section is used to construct the transitions of the FSM. Table 5 shows the transition table of the FSM based on the *ProcID* and the hit (H) or miss (M) inputs. Each transition changes the state of the FSM from a $prev_state$ to a $next_state$, producing a $safety_check$ signal that indicates if an access pattern belongs to the $Unsafe$ or a $Safe$ scenarios. Fig. 12 shows the CLD's architecture with the corresponding FSM, register file, and cache. $Unsafe$ scenarios are shown as a red line.

Table 5: The transition table of the CLD FSM [62].

FSM transition $prev_state \rightarrow next_state$	Transition condition (FSM input)	safety_check (FSM output)
$st_0 \rightarrow st_1$	(P_S | P_I) & (M | H)	0 (safe)
$st_1 \rightarrow st_2$	P_S & (M | H)	0 (safe)
$st_1 \rightarrow st_3$	P_I & (M | H)	0 (safe)
$st_2 \rightarrow st_1$	P_I & M	0 (safe)
$st_2 \rightarrow st_0$	(P_S & (M | H)) | (P_I & H)	1 (*unsafe*)
$st_3 \rightarrow st_1$	(P_S & H) | (P_I & (M | H))	0 (safe)
$st_3 \rightarrow st_0$	P_S & M	1 (*unsafe*)

Once an access is performed, the hit (H) and miss (M) signals are received from the cache output. The *ProcID* is used to determine whether the process being accessed is P_S or P_I. The output of the register file ($prev_state[addr_index]$), is a 2-bit encoding of the previously stored CLD state for set $addr_index$. This state is used to determine the last state of the FSM. Once all these inputs are received by the FSM, a next state ($next_state[addr_index]$) is generated and stored in the register file.

All the values in the register file are initialized to $Start$ (st_0) state. Since The first access in the leakage sequence corresponds to setup stage (S_1), any access will transition the FSM from st_0 to the trigger stage (S_2) represented by state st_1. The outcome of the trigger stage is dependent on the type of access and the output of the cache and will transition to the observation stage (S_3). S_3 can be one of two possibilities: either that the current observation stage is the third access of an $Unsafe$ scenario or it is the start of a another scenario. If the observation stage is in fact third access of an $Unsafe$ scenario, then the FSM will always transition from st_2 or st_3 to st_0 and generate $safety_check$ signal to indicate the $Unsafe$ scenario. If the observation stage is in fact third access of a $safe$ scenario, then the FSM will always transition from st_2 or st_3 to st_1 since the S_3 can be the first access in another scenario. This design ensures that every access to the cache is checked for potential access patterns that can lead to an $Unsafe$ scenario.

5.2.1 End-to-end example of CLD operation

To illustrate the operation of the proposed detector, consider a secure process P_S and an interference process P_I accessing the same cache location L. Assume the following sequence of accesses:

1. S_1: P_S accesses location L and receives a cache miss (M).
2. S_2: P_I accesses location L and receives a cache hit (H).
3. S_3: P_S accesses location L and receives a cache miss (M).

5.2.1.1 Step 1: Leakage scenario identification Based on the processes performing the accesses, the sequence corresponds to the leakage scenario SIS. The resulting hit/miss pattern is therefore (M,H,M).

5.2.1.2 Step 2: Leakage template classification Referring to Table 4, the hit/miss pattern (M,H,M) for the leakage scenario SIS belongs to the $Unsafe$ group (G_U). Consequently, the sequence is classified as a potentially exploitable cache leakage pattern.

5.2.1.3 Step 3: CLD FSM evaluation The same sequence is evaluated by the CLD FSM. Initially, the detector is in state st_0.

- The first access transitions the FSM from st_0 to st_1.
- Since the second access is performed by P_I, the FSM transitions from st_1 to st_3.
- The third access is performed by P_S and results in a miss (M). According to Table 5, the transition $st_3 \rightarrow st_0$ under condition $P_S \wedge M$ produces $safety_check = 1$.

The generated $safety_check$ signal indicates that the observed access sequence belongs to an unsafe leakage scenario. The warning can then be forwarded to the designated mitigation mechanism for further action. This example demonstrates how a sequence of

cache accesses is transformed into a leakage classification by the template and subsequently detected by the CLD FSM.

By integrating a process-based threat model with a cache-contention-based leakage model, the CLD architecture identifies exploitable access patterns associated with LSCAs-related leakages. When such patterns are detected, CLD issues a warning to a designated mitigation mechanism, enabling the system to react and address the threat in a timely manner.

The compact structure of CLD makes it suitable for use with a wide range of cache sizes and architectures. The functionality of the detector is independent of the workload and the specific behavior of executing processes as it relies only on readily available information, namely the process identifier and cache hit/miss outputs. However, the practicalities involved in the deployment of CLD depend on implementation-specific factors such as target cache organization, timing constraints, and the availability of process-identification information at run time. As a result, while CLD is designed to be scalable and portable across different cache architectures, the implementation's overhead costs and its integration effort may vary depending on the target system.

5.3 Chapter Summary

This chapter presented the Cache Leakage Detector (CLD), a run-time mechanism for identifying exploitable cache leakage patterns. A process-oriented threat model and a cache-contention-based leakage model were first introduced to characterize information leakage independently of attacker assumptions. A verification methodology was then used to analyze hit/miss behavior and derive a leakage template based on observable cache events. Using this template, a compact FSM-based detector was designed and implemented using only process identifiers and cache hit/miss outputs. The resulting architecture enables the detection of unsafe cache access patterns that may lead to logical side-channel leakages while maintaining a lightweight implementation suitable for practical deployment.

6 Mitigation Mechanisms

The detection of LSCAs is only the first step of dealing with them. While run time detection is important to highlight when these attacks may occur during the execution, a run time mitigation is needed to take advantage of such detection. While there are methods of mitigating LSCAs offline, little was done in the way to creating an effective run time mitigation. In this section, two methods of run time mitigation of LSCAs are presented. The first mitigation mechanism, Secure Cache Alternative Address Table (SCAAT), is primarily based on the work presented in [60], while the Recursive Secure Cache Alternative Address Table (RSCAAT) presented later in this chapter extends this work and is primarily based on [63].

6.1 SCAAT

SCAAT [61] aims at mitigating LSCAs at run-time. LSCAs take advantage of predictable cache behavioral patterns to leak information regarding the cache internal state. To alleviate this issue, SCAAT changes the predictable cache behavior when a threat manifests. In this section, the design, functionality, mitigation strategy, and SCAAT's impact on the behavior of the cache is presented. Additionally, a vulnerability of the SCAAT design and potential solutions to it are discussed.

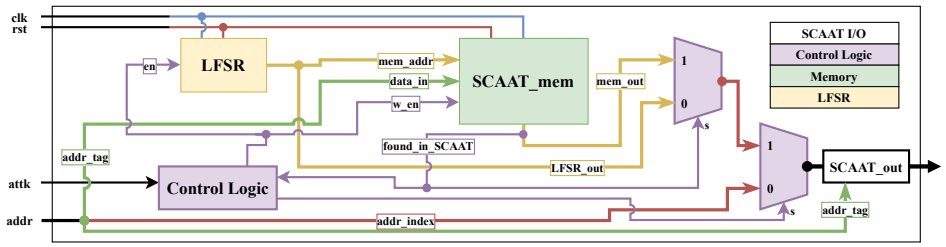


Figure 13: A block diagram of the components and signals of the $SCAAT_unit$ [60].

6.1.1 Design and Functionality

SCAAT uses a real-time mitigation mechanism, called the $SCAAT_unit$, which is only activated when a threat is detected by the monitoring system. Fig. 13 shows the block diagram of the components and signals of the $SCAAT_unit$. $SCAAT_unit$ has two inputs: the $attk$ signal received from the cache monitors and the cache address ($addr$). When the monitoring system detects a threat, it sends the $attk$ signal to a specialized logic that activates the different components of the $SCAAT_unit$. SCAAT mitigates attacks by changing the behavior of the cache at run-time. This is done by substituting the cache address index bits ($addr_index$) with a new randomized location. Changing where the cache is being accessed, the overall timing and access patterns of the cache change as well. This disassociates the internal state of the cache from the observable behavior. Fig. 14 shows the transition diagram of the $SCAAT_unit$'s control logic.

6.1.2 Cache Coherency Considerations

SCAAT aims at mitigating any access patterns that indicate possible information leakage or attacks on the cache. However, to guarantee that SCAAT will not interfere with the internal state of the cache, several cache coherency considerations must be taken into account.

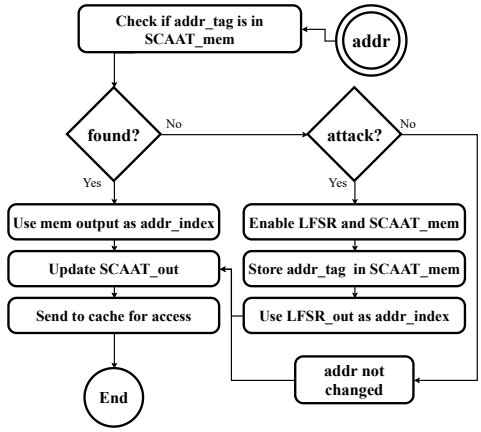


Figure 14: $SCAAT_unit$ control logic transition diagram [60].

First, The design of SCAAT must not change parts of the $addr$ that point to cache data (being accessed or evicted). At the lowest level of cache (L1 instruction and data caches), the $addr_tag$ (reference to cache block) and $addr_offset$ (reference to a word being accessed) reference specific data being accessed in the cache. At higher cache levels (L2 caches and LLC), this only involves the $addr_tag$ since, typically, only complete cache blocks are propagated from higher levels of cache [8]. Modifying the $addr_tag$ bits or $addr_offset$ bits will result in propagating invalid or incorrect data. This in turn causes cache coherency issues due to corrupted cache access requests.

Second, the mitigation strategy is only activated when a signal is received from the cache monitor or when any of the blocks that were re-mapped by SCAAT itself are later accessed. This limits the number of re-mapped $addr_tag$ to only those that are under threat and minimizes the possibility of coherency issues caused by SCAAT

Third, to facilitate correct re-mapping, SCAAT must be able to remember which cache blocks were re-mapped to which location, hence the need for a specialized memory. This is to make sure that future accesses to re-mapped blocks are performed properly, avoiding any coherency issues when re-mapped blocks require updating or need to be evicted from the cache.

Fourth, the mitigation of threats must be done without introducing any additional time overhea to the critical path of the cache. Coherency issues can be avoided by recording re-mapped blocks. However, because SCAAT changes the access addresses, SCAAT's design

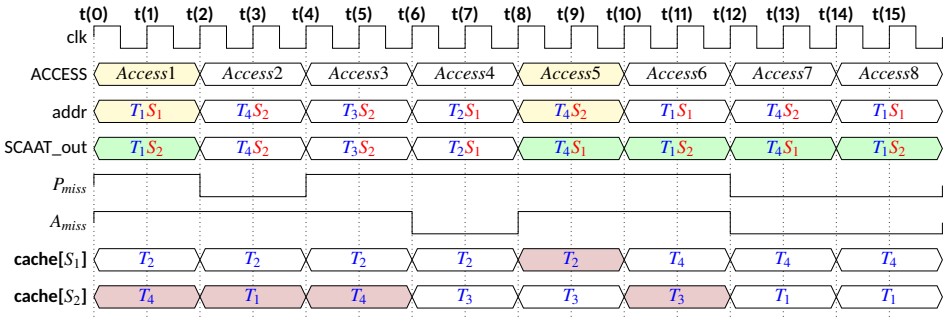


Figure 15: An example scenario demonstrating change in cache access pattern [60].

itself can still cause coherency issues due to timing delays. This must be taken into consideration when SCAAT is deployed. Since SCAAT is aimed at being integrated to any cache design, timing issues must be avoided as to not introduce delay to the cache itself. This depends on the design target of the integrated SCAAT and what type of system SCAAT is being integrated into. Incorporating SCAAT into a cache with strict timing constraints may require additional design considerations. However, an address must pass through the $SCAAT_unit$ without disrupting the timing of the cache itself.

6.1.3 Impact of Mitigation on Cache Behaviour

The timing waveform in Fig. 15 illustrates the change in the cache access pattern and the resulting cache behavior when the $SCAAT_unit$ is used. A single clock (clk) cycle is 1 unit of time u denoted by the function $t(u)$. Eight consecutive accesses, denoted using the $ACCESS$ signal, are performed within a period of 16 clock cycles $t(0)$ to $t(15)$ in a given access pattern. The cache is accessed using an address divided into a block tag and a cache set. Block tags denoted by T_x (colored blue) and cache set denoted by S_y (colored red) where x and y are the tag and set identifiers, respectively. Accesses under attack are highlighted in yellow. Cache evictions are highlighted in pink. The generated output from $SCAAT_unit$ activation is highlighted in green. Signal P_{miss} indicates the predicted miss pattern and signal A_{miss} indicates the actual miss pattern as a result of $SCAAT_unit$ activation. Signals cache[S_1] and cache[S_2] show the tag of the block stored in cache sets S_1 and S_2, respectively. Cache sets S_1 and S_2 are initially populated by tags T_2 and T_4, respectively.

In this example, the first access to the cache $Access1$ is under threat. This enables the $SCAAT_unit$ which re-maps the access from S_1 to S_2. The tag being accessed is T_1 at location S_2, a miss occurs because T_1 is not in the cache and must be retrieved from higher memory levels. During $Access1$, T_4 was evicted as T_1 was brought from higher levels of memory and replaced T_4 at location S_2. During the next access, $Access2$, T_4 is predicted to be in the cache, however due to the $SCAAT_unit$ re-mapping of the previous access, T_4 was evicted. This is an example of how the $SCAAT_unit$ actively altered the behavior pattern of the cache. T_4 should have been in location S_2 during $Access2$, but was evicted to make way for T_1, which was re-mapped by the $SCAAT_unit$. This change in behavior caused by the $SCAAT_unit$ activation can alter the behavior of future accesses. An example can be seen for $Access4$. When $Access4$ is performed, it is predicted to be a miss, but since $Access1$ was re-mapped from S_1 to S_2 by the $SCAAT_unit$, T_2 was not evicted from S_1 and stayed there until $Access4$ was performed. During $Access1$, T_2 would have been evicted from location S_1 to make way for T_4. However, as a result of $SCAAT_unit$ activation during $Access1$, T_2 was not evicted from S_1. This caused $Access4$ to be a cache hit, different than the predicted cache behavior.

An activation of the $SCAAT_unit$ has changed the predicted miss patterns, shown as P_{miss}, of the cache during the time period between $t(0)$ and $t(7)$. The resulting miss pattern, shown as A_{miss}, demonstrates the effects of a re-mapping accesses that are under threat on the overall miss hit patterns of the cache. Furthermore, the behavior changes are also reflected in future accesses as well.

As an example, consider $Access5$. A second threat was detected on $Access5$, causing the $SCAAT_unit$ to re-map T_4 from S_2 to S_1. Now, all accesses to T_4 will be re-mapped to location S_1. At this point, T_1 and T_4 are re-mapped to locations S_2 and S_1, respectively. Any future access to either of these blocks, will result of re-mapping each block to its respective new location.

The previous example shows that using the $SCAAT_unit$ can reduce the predictabil-

ity of the cache behavior, making it harder to collect accurate information regarding the cache operation conditions. This strongly implies that attackers who aim to collect the aforementioned information can no longer successfully correlate the information with the behavior of the cache.

6.1.4 Requirements for $SCAAT_mem$

The $SCAAT_mem$ is a compact Content Addressable Memory (CAM) that allows storing T_x that are under threat. This allows reduction in area overhead compared to other approaches that use look-up tables. For example, the baseline LLC used for this work, and described in Table 3, has a 4096 sets with 16 ways in each set. The baseline cache, therefore, requires 12 bits (S_{bits}) to access all sets of the cache. The number of bits required to identify each re-mapped tag (T_{bits}) is 20 bits. The depth of $SCAAT_mem$ corresponds to the number of entries stored in the CAM, with one entry available for each cache set in the baseline LLC. The width corresponds to the number of tag bits required to uniquely identify a cache block. Consequently, the storage requirement is:

$$Storage = Depth \times Width = 4096 \times 20 = 81920 \text{ bits}$$

which corresponds to 10240 bytes, or approximately 10.24 KB.

Randomization-based mitigation approaches use different versions of look-up tables to keep track of the randomized accesses to the cache. These tables can grow impractically large when used with large caches. Using the parameters of the baseline LLC an size estimate of such look-up tables can be made. Table 6 shows size estimations of randomization-based approaches when used in an 8-core system.

Table 6: Size estimation for Randomization-based approaches [60].

	Width	Depth	# of memories	total size (KB)
SCAAT/RSCAAT	20	4096	1	10.24
RP-Cache[74]	12	4096	8	49.152
NewCache[75]	15	4096	8	61.44
Re-configurable Address Mapping/translation Table[52]	32	4096	16	262.144

Although the previous size estimation for different implementations of look-up tables do not take into account the required logic to control and access such look-up tables, they are suitable estimates to evaluate the storage overhead introduced to the cache storage space by each look-up table. All in all, the compact and specialized use of the $SCAAT_mem$ allows the implementation of a low area CAM that stores all the required data to achieve the required randomization.

6.1.5 Attack model and SCAAT vulnerability

While the $SCAAT_unit$ provides a sufficient randomization of the accesses to the cache at a low area overhead, it comes with a potentially dangerous vulnerability. The baseline LLC described in Table 3 requires 12 S_{bits} to access all sets of the cache. Accordingly, the LFSR is implemented as a 12-bit primitive-polynomial register that will generate and store the pseudo-random sequences used by the $SCAAT_unit$ to scramble the access pattern to the cache. A primitive-polynomial LFSR with an *N*-bit register will generate a sequence of maximum length $2^N - 1$ unique combinations of length *N*-bit. When the LFSR reaches the maximum length sequence, the value stored in the LFSR register will return to its initial value, repeating the same sequence in the same order.

The $SCAAT_unit$ uses an LFSR to store T_x at locations generated by the LFSR in the $SCAAT_mem$ when T_x are under threat. Each T_x remains inside the $SCAAT_mem$ until the LFSR reaches its maximal sequence. Therefore, every tag T_x stays in the $SCAAT_mem$ for $2^N - 1$ activations of the $SCAAT_unit$. Nevertheless, if a threat is detected to the set S_x where tag T_x has been remapped before the LFSR completes its maximal period, the $SCAAT_unit$ is not reactivated and tag T_x is not protected. This behavior can be exploited by an adversary.

Each T_x is replaced in the $SCAAT_mem$ with a new tag after $2^N - 1$ of $SCAAT_unit$ activations. These activations must be as a result of threats on tags other than T_x. If an adversary targets the set S_x where T_x was previously re-mapped, they can deduce the LFSR sequence. Since the LFSR sequence repeats in the same order, the adversary can generate an LFSR sequence, hereinafter called $LFSR'$, where the *N*-bit representing S_x is the beginning of the $LFSR'$ sequence. Algorithm 1 shows an attack flow that exploits the SCAAT vulnerability.

Algorithm 1 Attack Flow on SCAAT_unit and SCAAT_mem [63].

```
Initialize LFSR with primitive polynomial of width N
Initialize SCAAT_mem as empty
while program executes do
    if (T_x under threat) then
        S_x ← LFSR_out
        SCAAT_mem[S_x] ← T_x
    end if
    while (Attack S_x repeatedly) do
        Record Access Trace
        Generate candidate LFSR'
        for each candidate LFSR' do
            if Trace correlates to LFSR' then
                LFSR' is true LFSR sequence
            end if
        end for
    end while
    Predict future remaps of T_x
    if LFSR completes full cycle 2^N − 1 then
        Replace oldest entries in SCAAT_mem
    end if
end while
```

Several attacks can be used to exploit this vulnerability. If an adversary is aware of the presence of the $SCAAT_unit$, targeting T_x will knowingly alert the cache monitor. Since the depth and width of the cache are available to the adversary, the width of the LFSR and depth of the $SCAAT_unit$ can also be deduced. Once the cache monitor detects an attack on the cache, it will activate the $SCAAT_unit$, generating a randomized location for the access under threat. Since the adversary is aware of the presence of the $SCAAT_unit$ and the length of the LFSR, the adversary can use this new location to create several $LFSR'$ sequences, a sequence for each primitive polynomial available for a the LFSR length. Once the $LFSR'$ sequences are generated, an eviction attack, a Prime+Probe attack variant, can be used to target locations within the different $LFSR'$ sequences and generate a new trace in the same order in the $LFSR'$ sequence. If the adversary was able to correlate one of the $LFSR'$ sequences to the random re-mapping done by the $SCAAT_unit$, the adversary can commence attacking the cache with the proper re-mapping of all the targeted data. The $SCAAT_unit$, in that case, is bypassed.

Despite this limitation, SCAAT has demonstrated that detection-triggered remapping can introduce effective disruptions to predictable cache access patterns and reduce the correlation between observable cache behavior and the cache internal state. However, the analysis presented in this section revealed a specific vulnerability: once a tag has been

remapped in the $SCAAT_mem$, subsequent threats targeting the same tag may not trigger a new remapping event until the corresponding tag is naturally replaced. An adversary exploiting this behavior may eventually predict future remapping locations and reduce the mitigation effectiveness. This observation motivates the development of RSCAAT. RSCAAT extends SCAAT by introducing recursive remapping capabilities that continuously re-randomize previously protected tags whenever new threats are detected.

6.2 RSCAAT

In this work, the vulnerability presented in the previous section is addressed. It can be overcome by re-mapping existing T_x that are already stored in the SCAAT when they fall under threat – a re-mapping redundancy is implemented by designing the Recursive Secure Cache Alternative Address Table (RSCAAT).

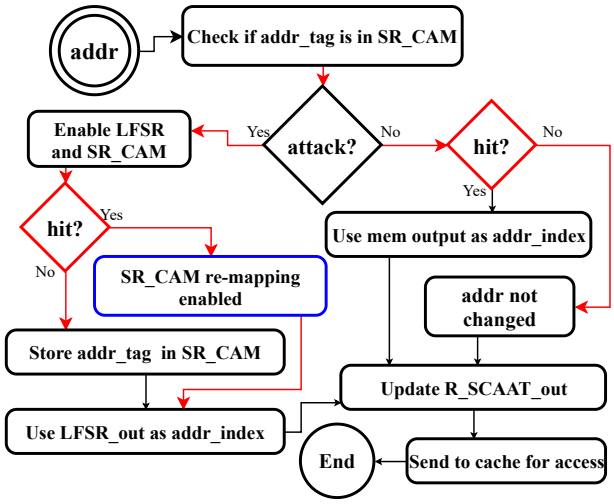


Figure 16: Updated control logic transition diagram for the RSCAAT [63].

The RSCAAT is comprised of three components: A control logic that enables the different components of the RSCAAT, a specialized content addressable memory called the Self-Remapping Content Addressable Memory (SR_CAM), and a primitive polynomial LFSR.

6.2.1 RSCAAT control logic

The control logic of the $SCAAT_unit$, shown in Fig. 14, can be reconfigured to achieve and take advantage of the SCAAT re-mapping redundancy required. Fig. 16 shows the reconfigured control logic transition diagram of the new recursive mitigation mechanism (called hereinafter R_SCAAT_unit) where the reconfigured control diagram transitions are highlighted in red. The R_SCAAT_unit has the same inputs (clk, rst, $attk$, and $addr$), and internal signals ($addr_tag$, $addr_index$, and $LFSR_out$) as the original $SCAAT_unit$ control logic. However, the added redundancy re-mapping functionality, highlighted in blue, is discussed in detail in the next section. The $found_in_SCAAT$ signal is replaced with a hit signal that is generated as an output of the SR_CAM. The output of R_SCAAT_unit is the re-mapped access.

The R_SCAAT_unit control logic leverages the signal received from the cache monitor. Once an access to the cache is received, the R_SCAAT_unit checks for the signal sent by the cache monitor. This reconfigured control logic allows the R_SCAAT_unit to acti-

vate the re-mapping redundancy, if T_x is a *hit* and the location where T_x is stored in the *SR_CAM* is under threat. If a threat signal is received from the monitor when T_x is being accessed, both the LFSR and the *SR_CAM* are activated. If the *SR_CAM* reports that T_x is not a *hit*, it means that T_x is not stored in the *SR_CAM*, then T_x is stored at location of the current value in the LFSR register (*LFSR_out*). If T_x is a *hit*, the re-mapping redundancy is activated and T_x is re-mapped to the current *LFSR_out* value. In both cases, the *LFSR_out* is used as the index bits of the *R_SCAAT_out* output until T_x is either written into the *SR_CAM* (when T_x is not a *hit*) or re-mapped into a new location in the *SR_CAM* (when T_x is a *hit*). If no threat signal is received from the monitor when T_x is being accessed, the *hit* output signal from the *SR_CAM* is checked. If T_x is a *hit*, the output of *SR_CAM* is used as the index bits of the *R_SCAAT_out* output. If T_x is not a *hit*, this means that T_x was not not previously re-mapped and the unchanged access address is used as *R_SCAAT_out*.

The *R_SCAAT_unit* control logic ensures that in the case of any threat detected by the monitor, the control logic reacts to potential threats to previously re-mapped locations. However, in the case where T_x is a *hit* and threat signal is received from the monitor, the re-mapping of T_x inside *SR_CAM* is activated internally.

6.2.2 Self-Remapping Content Addressable Memory

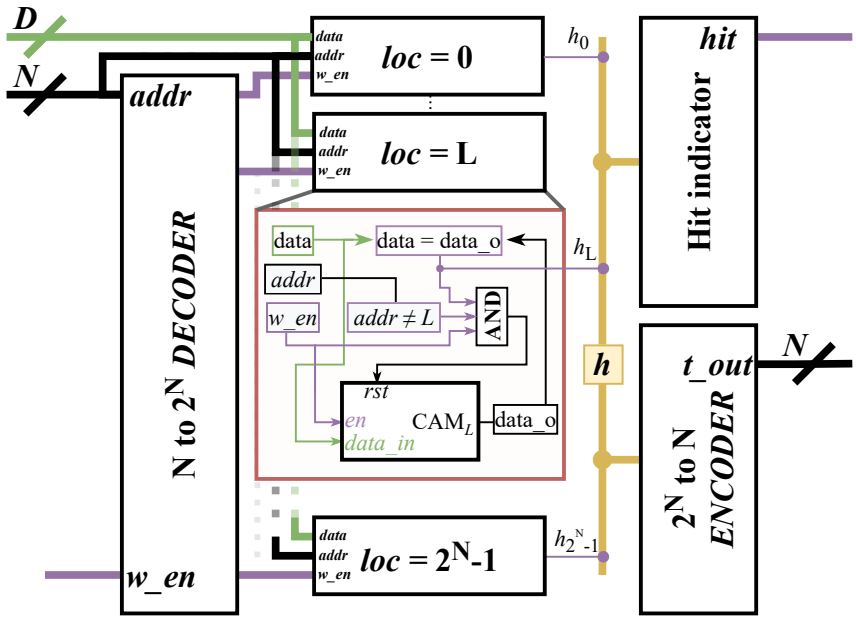


Figure 17: Components of the Self-Remapping Content Addressable Memory (SR_CAM) [63].

The *SR_CAM* is composed of four main components, illustrated in Fig. 17:

1. An N to 2^N decoder that controls the writing process of the *SR_CAM*. The decoder has two inputs: $addr$ is a bit vector of length N that specifies to which location in the *SR_CAM* the writing process is performed and a *w_en* bit that enables the writing process. Both $addr$ and *w_en* are received from the control logic of the *RSCAAT_unit*. The output of the decoder is a single bit signal (*w_en*) connected to the location (loc) between 0 and $2^N - 1$, corresponding to the decoded $addr$ vector.

2. The self-remapping mechanism that ensures that no two locations in the SR_CAM contain duplicate data and allows the re-mapping of data internally. The mechanism has three inputs: $data$ is a bit vector of length D that contains the T_x input data of the SR_CAM, the w_en bit received from the decoder, and the $addr$ bit vector. The output is a single *hit* bit (h_L) that indicates if $data$ is equal to the contents stored at $loc = L$. All the output *hit* bits from each of the locations in the SR_CAM coalesce into a single hit vector h of length 2^N.

3. A 2^N to N encoder that performs the read process of the SR_CAM. Its input is the hit vector h. The encoder returns the location t_out where $data$ was found in the SR_CAM as output.

4. The hit indicator is an array of OR gates that resembles a full binary tree. It has one input and one output. The hit indicator takes the hit vector h and returns a single bit signal hit, which indicates if $data$ was at all found in the SR_CAM. Fig. 18 shows the structure of the hit indicator. Each OR gate in the hit indicator is connected to two "child" gates. At $depth = 0$, the OR gates are connected to their respective bits of hit vector h as shown in the illustration. The OR gate at $depth = 0$ will return the result of OR the entire hit vector h resulting in either a '1' if there is an SR_CAM hit or '0' if no SR_CAM hits found.

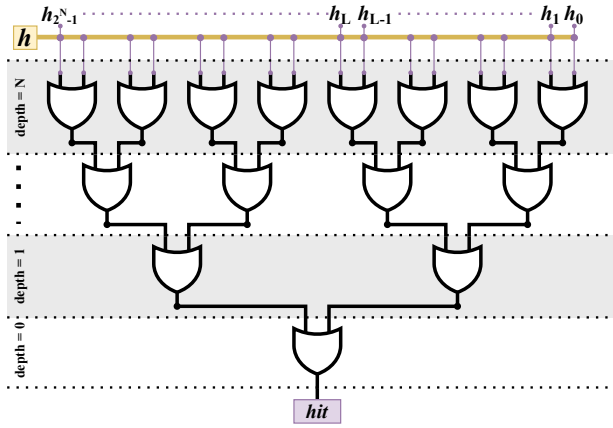


Figure 18: The structure of the hit indicator [63].

The w_en must be a '1' in order to facilitate the writing of $data$ into location $addr$ in the SR_CAM. When w_en is enabled, the decoder sends the enable signal to the loc_L corresponding to $addr$. The data stored at loc_L ($data_o$) is compared to the input $data$. The $addr$ is compared to the corresponding L. The output signal h_L is connected to the h bit vector. The hit and t_out signals are produced by passing the h bit vector to the hit indicator and the encoder respectively.

Fig. 19 shows the transition diagram of the re-mapping process. The results of the two comparisons, $data_o$ with $data$ and $addr$ with L determines if T_x requires internal re-mapping inside the SR_CAM. The w_en is only used when a threat on T_x is detected. This means that T_x is only written into the SR_CAM at the event of a threat. If $data_o$ at loc_L is equal to input $data$, it means that T_x is stored in the SR_CAM at loc_L and the h_L is returned as '1'. If L is not equal to $addr$, it means that the location in which the T_x is being written into is not equal to the location were T_x was previously stored. If en is '1' (a threat is

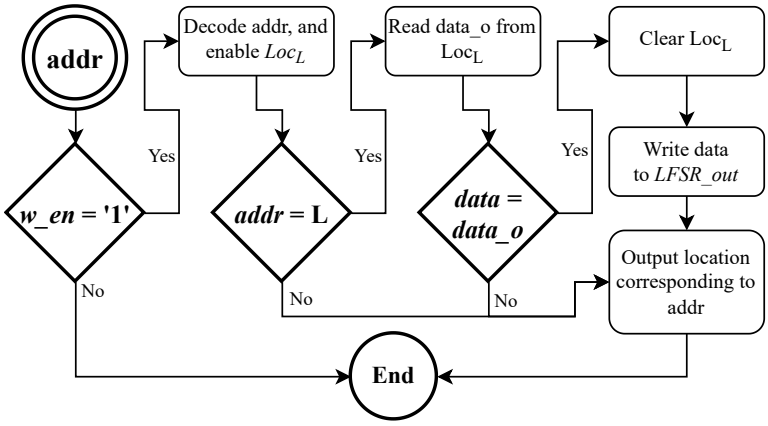


Figure 19: The re-mapping redundancy transition diagram [63].

detected) and $data = data_o$ (T_x was previously stored in the SR_CAM) and $addr \neq L$ (T_x is being written into a new location), then the previous location L where T_x was previously stored is emptied (reset) and T_x is written into a new location corresponding to the input $addr$.

Once the SR_CAM is accessed, in case a threat is detected or not, the result of comparing the input $data$ with $data_o$ at all the locations from $L = 0$ to $L = 2^N - 1$ will result in a bit vector h of length 2^N. Since the re-mapping redundancy ensures that no two locations store the same data, the h vector will only have a single '1' bit (at the vector element corresponding to the location where $data$ was a hit) while all the other bits in the vector are '0'. When the vector h is passed to the encoder, the encoder will return the binary representation of the location where $data$ was a hit. Additionally, the hit indicator returns if the $data$ is in fact a hit.

6.2.3 SR_CAM LFSR

The SR_CAM derives its addresses from the LFSR, the CAM reaches full capacity when the LFSR reaches its maximal period. At the point, the LFSR has reached its maximum length sequence and will return to its initial state. The data at the SR_CAM location after the maximal period is overwritten with new data. A new tag that is not previously stored in the SR_CAM will be directly mapped to the value of the $LFSR_out$, overwriting data at that location. Therefore, a complex replacement policy is not needed and direct mapping of the new tag is used to reduce the complexity of the SR_CAM.

The bit width of the LFSR is crucial to the operation of SR_CAM. The LFSR must generate a sufficient number of combinations to cover all the sets of the cache. To achieve the pseudo-randomness expected from the LFSR, its width must be determined by the size of the cache. Specifically, the LFSR must be an *N*-bit wide primitive-polynomial capable of producing a sequence of maximum length $2^N - 1$ that is equal to the number of the available cache sets minus one. If the LFSR does not produce sufficient combinations to cover all cache sets, remapping would be restricted to only a subset of the cache. A confined remapping would leave cache sets unused during remapping, thereby increasing the likelihood of both cache and CAM misses. Such partial coverage can lead to non-uniform remapping, concentrating cache misses within specific regions and thereby undermining both performance and security.

The worst-case remapping chain length is bounded by the LFSR period $2^N - 1$ as a

maximal polynomial LFSR is being used. While this scenario is theoretically possible, it is highly improbable due to the irregularity of cache access patterns in real programs. In order for such a case scenario to occur, a single cache line would need to be accessed in an exact pattern that repeatedly triggers a remapping to a sequence of locations that traverses the entire maximal LFSR period. This further assumes that no other cache lines are determined to be vulnerable by RSCAAT during the entire observation period and that no interfering events disrupt the progression of the remapping sequence. Under these conditions, a single cache line would be continuously remapped until the maximal period of the LFSR is achieved. The occurrence of such scenario will require extraordinary circumstances.

In the experimental setup, the CLD would require three accesses to make a determination whether a given location is vulnerable, which in turn would trigger the RSCAAT to remap it. Consequently, every remapping event requires three accesses. With a 12-bit LFSR, the maximal sequence length is $2^{12} - 1 = 4095$. To traverse the entire LFSR sequence, 4095 triggering events are needed. Since a single triggering event requires 3 accesses, the total number of accesses needed to traverse the entire LFSR period is $3 \times 4095 = 12285$ consecutive accesses to the same cache line without interruptions. Achieving this uninterrupted access pattern to a single cache line is practically infeasible.

6.2.4 Randomization considerations

For this work, the LFSR is used to facilitate randomization primarily for three reasons: a) LFSRs are very cost-effective in terms of hardware. Increasing the size of the cache requires only a logarithmic increase in the LFSR's width and complexity, resulting in negligible area overhead. b) LFSRs generate near-uniform pseudo-random sequences, but ensures non-repeating addresses during it's sequence. This guarantees that all cache lines within the cache are equally likely to be candidates for mapping/remapping when a cache line is under threat. c) LFSRs introduce low latency to the data path, allowing the SR_CAM to operate within a strict timing constraints with little to no impact on critical paths.

Other randomization techniques, such as those presented in [58], [76], and [67], use strong keyed or cryptographic randomization that yield uniform distribution and unpredictable access patterns. Such techniques are far more appealing than an a simple LFSR. Nonetheless, those techniques typically serve as the primary method of modifying the cache access behavior at high hardware complexity, area overhead, and potential latency costs.

In this work, however, the change in cache behavior is largely driven by the leakage pattern detection frequency by Cache Leakage Detector (CLD). Leakage patterns can be a result of any type of contention in the shared cache, whether it is caused by an attacker or by normally occurring contention. Thus, the observable behavior post-randomization is indistinguishable from a normally-occurring cache contention, regardless of whether the original contention stemmed from an attack or not. The LFSR only changes the mapping when a threat is detected, reducing unnecessary deviation of normal cache behavior. This means that the LFSR pattern is only visible when a potential threat is detected on the shared cache line, preventing long consecutive samples of the randomization output. This prevents standard LFSR state-recovery attacks that rely on observing linear relations across long sequences. Additionally, the remapping redundancy introduced by the SR_CAM further obscures any identifiable patterns that might otherwise reveal the LFSR's initial state post randomized access.

The combination of (1) detection-triggered randomization during any leakage pattern, (2) applying the LFSR in non-continual pattern, and (3) the remapping redundancy pro-

vided by the SR_CAM, effectively conceals the LFSR sequence and prevents the discovery of its initial state. Therefore, we choose an LFSR instead of stronger randomization schemes, such as those used in GIR-Cache, ScatterCache, or MIRAGE, because it provides sufficient randomization while significantly reducing hardware complexity, area, and latency.

6.2.5 SR_CAM security

While the current implementation of the SR_CAM adds a needed level of security to the RSCAAT, the security of the SR_CAM itself is difficult to assess. An advantage that SR_CAM has is its unconventional usage in the data path. While CAM-based tag lookup is predictable by the nature of their role in the cache design, the SR_CAM operates with randomness correlated with the LFSR. Without knowledge of the LFSR's initial state, the behavior of the SR_CAM is very difficult to predict. Even in case the sequence of the LFSR is tracked, a full knowledge of the SR_CAM contents is needed to predict it's behavior. Further work is needed to determine it's vulnerability to attacks normally used to target cache CAM-based tag lookup.

6.2.6 RSCAAT integration considerations

The proposed RSCAAT introduces an additional step to the cache access pipeline. Implementing this additional stage has an impact on the cache system performance. Such performance impact is implementation dependent and will vary based on the underlying cache architecture.

The primary latency concern comes from the access time penalty introduced with the SR_CAM. Since SCAAT performs both read and write operations to the SR_CAM, achieving timing closure may require an additional access cycle for SR_CAM access. A strict constant time behavior is enforced across all cache accesses. Even when SR_CAM content was not logically required, an access to the SR_CAM was performed to maintain uniform timing across all accesses. As a result, all SR_CAM accesses have the same latency, avoiding any conditional time differences that can be exploited as a timing side-channel.

In the evaluated cache model, RSCAAT was integrated without extending the critical path. RSCAAT is active during the tag comparison stage with no modification to the existing hit and miss decision logic. The additional SR_CAM lookup was done within the existing access window, consequently ensuring that the overall cache access latency remained unchanged in the baseline cache. It is important to distinguish access latency from cache effectiveness metrics such as hit rate and misses per kilo instructions (MPKI). While the RSCAAT implementation did not increase the latency of an individual cache access in the evaluated baseline cache, the remapping process can alter cache occupancy and replacement behavior. Consequently, system-level performance metrics such as hit rate and MPKI may still be affected, as discussed in Chapter 7.

RSCAAT extends the original SCAAT design through recursive remapping and the introduction of the SR_CAM. The proposed architecture enables continuous re-randomization of cache locations associated with leakage patterns while preserving compatibility with the cache monitoring framework. Although RSCAAT addresses the vulnerability identified in the original SCAAT design, the security properties of the proposed SR_CAM require further investigation. In particular, future work should evaluate whether attack techniques traditionally used against CAM-based lookup structures can be adapted to target the SR_CAM under realistic threat models. The recursive remapping mechanism is designed to mitigate the identified vulnerability and substantially reduce opportunities for exploiting predictable cache behavior.

6.3 Chapter Summary

This chapter presented two run-time mitigation mechanisms for Logical Side-Channel Attacks. First, SCAAT was introduced as a detection-triggered remapping mechanism that alters cache access behavior by dynamically changing cache set mappings. The analysis of SCAAT revealed a vulnerability associated with repeated attacks on previously remapped entries. To address this limitation, RSCAAT was proposed. RSCAAT extends SCAAT through recursive remapping and the introduction of the Self-Remapping Content Addressable Memory (SR_CAM), enabling continuous re-randomization of vulnerable cache locations. The design, operation, storage overhead, security considerations, and integration implications of both mitigation mechanisms were presented. While RSCAAT addresses the vulnerability identified in SCAAT, assessing the security of the proposed SR_CAM against dedicated CAM-oriented attacks remains an important direction for future work.

7 Experimental Results

In this section, the experiments conducted to evaluate the CLD, SCAAT, and RSCAAT are discussed. Several experiments were used to determine the effectiveness and security of both the detection and mitigation used during the experiments.

This chapter aims to answer four evaluation questions. First, how accurately can CLD detect cache leakage patterns and known logical side-channel attacks compared to existing approaches? Second, what hardware overhead is introduced by the proposed detection mechanism? Third, does RSCAAT successfully mitigate the vulnerability identified in the original SCAAT design? Finally, what are the performance and area trade-offs associated with integrating SCAAT and RSCAAT into the cache hierarchy?

7.1 Detection Analysis and Results

The detection capability and accuracy of CLD are evaluated against state-of-the-art online detection methods. Furthermore, the hardware area overhead is quantified and discussed to demonstrate CLD's efficiency.

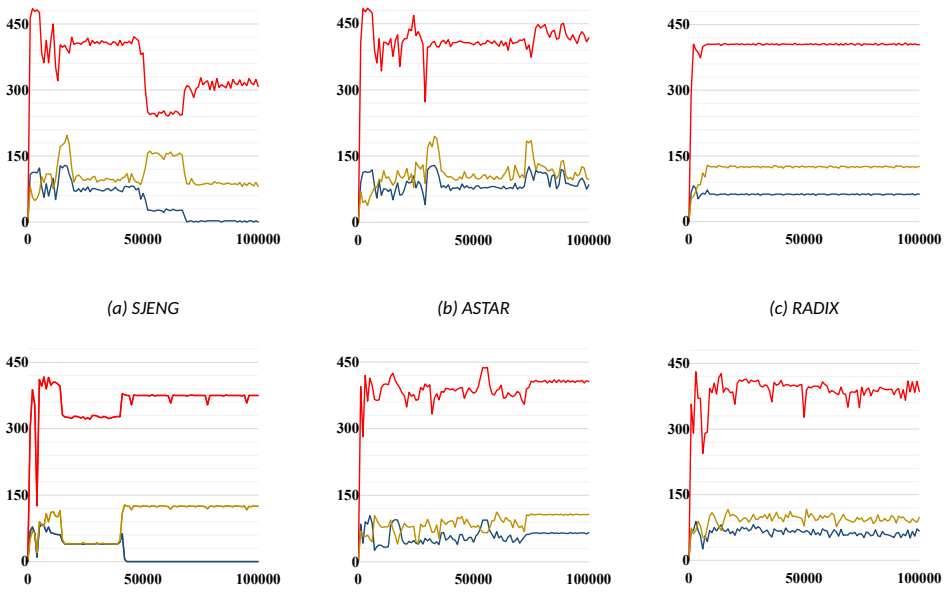
Leakage Detection

Fig. 20 shows the leakage detection histogram for all the benchmarks used for this experiment. The figures show the number of leakages detected (Y axis) by the SVA baseline, CLD, and LiD-CAT for every 1000 accesses (X axis). The SVA implementation is used as a reference baseline as it directly encodes the leakage templates derived in Chapter 5. Detections reported above the SVA baseline are treated as potential over-detections. CLD reported, on average, 26% more leakage events than the SVA baseline. LiD-CAT reported 71% more leakage events compared to the SVA baseline. Under the assumption that detections exceeding the SVA reference correspond to potential false positives, CLD exhibits a substantially lower estimated false-positive tendency than LiD-CAT. Therefore, for every ten detections produced by CLD, approximately three detections may correspond to false positives, compared to more than seven detections for LiD-CAT.

Detection Accuracy

The CLD detection accuracy varies depending on the access pattern of the benchmark. On one hand, applications such as ASTAR and FACESIM experience a much lower increase in leakage detection at 11% and 23% respectively. This can be attributed to the random nature of access to data for these benchmarks, reducing the frequent re-usage of the same data blocks. RADIX and VOLREND, on the other hand, experience up to 46% and 44% increase in leakage detection respectively. Those benchmarks are sort and image rendering algorithms that frequently re-use the same data blocks. The reuse of data increases the possibility of detecting a higher number of leakages with formula ID SSS. This can potentially lead to flagging these access patterns as leakages, while they are, in fact, normal access patterns.

Fig. 21 shows the correlation between accesses patterns of a secure process and CLD as well as accesses patterns of a secure process and LiD-CAT. The figure demonstrates that CLD has higher correlation with the secure process accesses patterns for almost all the benchmarks. This shows a direct relationship between the number of accesses by a secure process and the detected leakages by CLD, indicating higher accuracy in leakage detection. All in all, detection made by CLD have an average of 0.7964 positive correlation to the number of accesses performed by a secure process, more than double that of LiD-CAT at 0.3195 positive correlation.



(a) SJENG *(b) ASTAR* *(c) RADIX*

(d) VOLREND *(e) FACESIM* *(f) VIPS*

*Blue line shows SVA baseline leakage detections, yellow line shows CLD leakage detections, and red line shows LiD-CAT leakage detections.

Figure 20: Leakage detection histogram for all benchmarks.

For the attack-injection experiments, the same attack-injection schedule was applied to both CLD and LiD-CAT to ensure a fair comparison. Attack locations were generated using a fixed pseudo-random seed and injected into identical execution traces of the AES benchmark. The pseudo-random seed was fixed across all experiments to ensure reproducibility and to guarantee that CLD and LiD-CAT were evaluated under identical attack conditions. Unless otherwise stated, the reported results correspond to the average obtained across multiple experimental runs using the same set of attack positions. The same attack-injection schedule and pseudo-random seed were used for both CLD and LiD-CAT to ensure a fair comparison. Statistical variation across repeated runs was not recorded and is therefore not reported in this work.

CLD is demonstrated to detect well-known attacks at higher accuracy than LiD-CAT. 250 attacks per 10000 accesses were injected into the AES algorithm during run time at random. The results were obtained by comparing the detections made by CLD and LiD-CAT before and after attack injections. Table 7 shows the number of attack detections made by CLD and LiD-CAT per attack type. On average, CLD has a much better accuracy rate of 87% of injected attacks detection compared to only 48% for LiD-CAT.

Area Overhead

In order to evaluate and analyze the area overhead of the proposed CLD, several configurations of common shared Last Level Cache sizes were synthesized with CLD. The area estimation were made using the Cadence Genus tool [14] and a $40nm$ CMOS technology commercial library. The OpenRAM memory compiler [32] is used to estimate the memory area of the cache. Due to the very large SRAM area of the LLC caches used, the cell

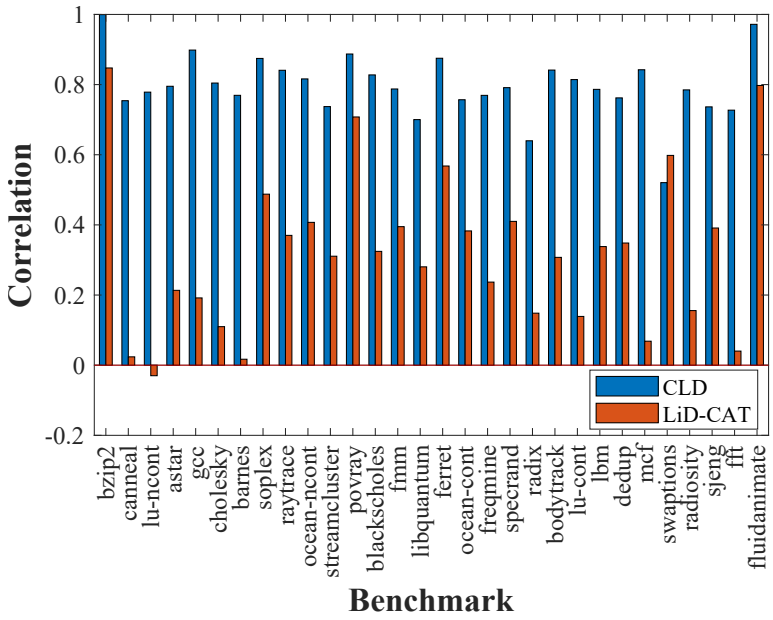


Figure 21: Access pattern correlation with CLD and LiD-CAT detections [62].

area overhead of CLD is compared to the cell area of the cache excluding the SRAM memory area and its associated controllers. All synthesized caches are configured as 16-way associative with 64 bytes per cache line.

Table 8 shows the sequential (**SEQU**), combinational (**COMB**), and total (**TOTAL**) area overhead (**AR**) and cell count overhead (**CL**) introduced by CLD to the control portion of the cache. The CLD SEQU area overhead slightly increases as the number of CLD SEQU cells doubles when the cache size doubles. As the cache size doubles, double the number of CLD SEQU cells are required to store the CLD states for each cache set. However, since the control logic (FSM) of the CLD does not change, the CLD COMB area overhead slightly decreases as cache size doubles. Yet, an increase in the number of COMB cells can be seen. This increase is because additional logic was added for read/write operation from/to the register file. As the size of the register file increases, so does its associated read and write control logic. All in all, on average, CLD adds 8.79% to the total control portion of the cache. Compared to the overall area of the cache, including both control logic and SRAM memory area of the cache, CLD introduces a negligible 0.002% increase to the total cache area.

7.2 Mitigation Analysis and Results

In order to evaluate the results from each experiment, three metrics were adapted and explored. First, the effect that introducing any of the detection and mitigation mechanisms on the cache behavior is explored. This metric can show what type of effect can be expected when the access patterns to the cache are changed. Second, the security layer introduced by each of the detection and mitigation is discussed. Lastly, the introduction of additional logic to the cache will certainly affect its long term performance.

Table 7: CLD's number and accuracy of attack detections injected into AES algorithm.

Attack type	Leakage Formula	CLD		Lid-CAT	
		dt	ac	dt	ac
Bernstein's attack	$P_S \rightarrow P_S \rightarrow P_S$	301	83%	602	35%
Flush + Reload	$P_S \rightarrow P_S \rightarrow P_I$	256	98%	516	48%
Evict + Time	$P_S \rightarrow P_I \rightarrow P_S$	201	76%	602	42%
Collision Attack	$P_I \rightarrow P_S \rightarrow P_S$	246	98%	252	99%
Flush+Flush	$P_I \rightarrow P_S \rightarrow P_I$	263	95%	603	41%
Prime+Probe	$P_I \rightarrow P_S \rightarrow P_I$	301	83%	512	49%
Attack formula 3	$P_I \rightarrow P_I \rightarrow P_S$	259	97%	602	42%
Attack formula 2	$P_S \rightarrow P_S \rightarrow P_S$	301	83%	769	33%
Attack formula 17*	$P_S \rightarrow P_S \rightarrow P_S$	301	83%	558	45%
Attack formula 18*	$P_S \rightarrow P_S \rightarrow P_S$	301	83%	557	45%

*Formulas 17 and 18 are similar to Flush+Flush attacks. **dt**: number of detections. **ac**: accuracy of detections.

Table 8: CLD's area overhead for all cache configurations [62].

cache		CLD Control overhead						CLD
size	# sets	SEQU		COMB		TOTAL		total
		AR	CL	AR	CL	AR	CL	overhead
2 MB	2048	6.36%	4160	16.51%	11578	8.70%	15891	0.0019%
4 MB	4096	6.61%	8256	15.46%	18836	8.33%	27176	0.0018%
8 MB	8192	7.67%	16448	13.53%	51906	9.35%	68701	0.0020%

* Including buffers and inverter cells

Effect on cache access behavior

Figure 22 shows the distribution of accesses to each LLC location for the *mcf* benchmark. This figure only shows accesses that occurred as a result of either re-mapping a cache block due to SCAAT or RSCAAT activation or as a result of an access to a cache block previously re-mapped due to SCAAT or RSCAAT activation. The baseline shows the number of accesses to each cache location before SCAAT or RSCAAT were used.

The effect of using the RSCAAT on the behavior of the LLC cache is very clear. The baseline exhibits a distinct access behavior that can be observed by potential leakages. This can be used to distinguish workload accesses to certain locations of the LLC as certain locations have distinct access patterns. Using the RSCAAT not only substantially reduces the observability of those leakage-like patterns, but also introduces an additional layer of security. The randomness in which the locations are remapped is compounded each time a cache location exhibits a leakage pattern. For every remapping done, the behavior of the cache is altered for both the locations where the access was supposed to happen and the location where the RSCAAT remapped the access. While using SCAAT does the same in terms of eliminating patterns, an attack using the attack model described in 6.1.5 can bypass the pattern elimination and help an adversary discover the new location where a cache block will be remapped by SCAAT. This reduces the effectiveness of SCAAT. As shown below, such vulnerability is not observed in RSCAAT remapping mechanism.

Security

The design principle of RSCAAT is devised to address the vulnerability identified in the original SCAAT design described in 6.1.5. An adversary can generate an $LFSR'$ sequence to predict the next output of the SCAAT LFSR by targeting data blocks with tags T_0 to T_n at locations S_0 to S_n respectively. Predictably, this will intentionally alert the monitor and

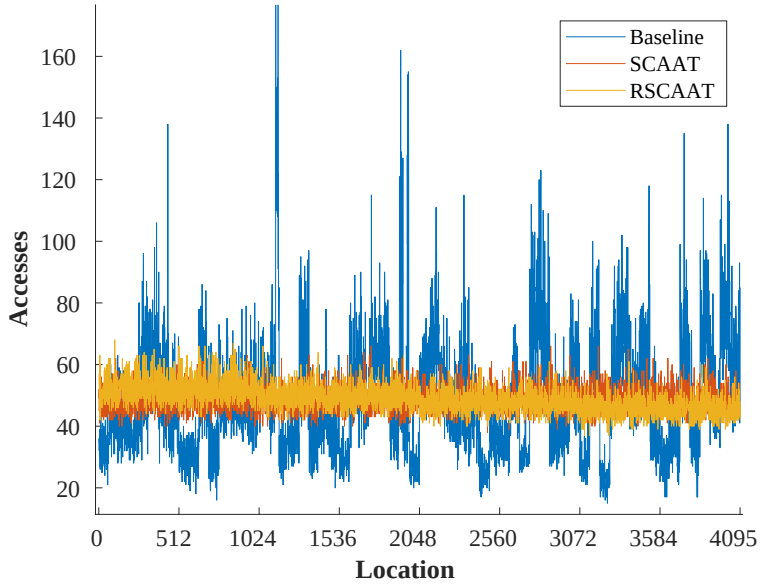


Figure 22: The distribution of number accesses per location in the LLC for the mcf benchmark [63].

activate the SCAAT to re-map the data blocks to new sets. The adversary can now target sets at random and check if the data block (T_x) in the targeted set (S_y) is re-mapped to a new location by the SCAAT. If T_x is remapped, it means that S_y bit combination has not yet been reached by the SCAAT LFSR and is a combination that will appear after the current value of the SCAAT LFSR. If T_x is not remapped, this means that T_x has been re-mapped before by SCAAT and the S_y bit combination has already appeared in the SCAAT LFSR sequence and is a combination that appeared before the current value of the SCAAT LFSR. Due to the vulnerability of SCAAT, continuously checking which tags are re-mapped when being targeted helps narrowing down where each S_y bit combination lays in respect to the current SCAAT LFSR value. This information allows an adversary to generate a reliable list of $LFSR'$ sequences and can predict where each next re-mapping by SCAAT will be done. This type of exploit is substantially more difficult against RSCAAT and was not observed during the conducted experiments.

To test the security of RSCAAT versus SCAAT, 100 unique tags, called T_{set} hereinafter, with at least one occurrence of vulnerability were traced for three benchmarks with different SCAAT/RSCAAT activation rates: *libquantum* (high activation rate), *streamcluster* (medium activation rate), and *swaptions* (low activation rate). The choice of 100 tags was made to ensure coverage across a wide range of remapping events while maintaining tractable trace analysis. Preliminary inspection of the benchmark traces showed that increasing the number of tracked tags beyond 100 did not significantly alter the observed activation trends for either SCAAT or RSCAAT. Consequently, the selected tag count was deemed sufficient to characterize the behavior of the mitigation mechanisms. The initial value of the LFSR is set to be the same for all the benchmarks. This will ensure that the same LFSR sequence is used for all the traces. Table 9 shows that RSCAAT was activated for all instances of vulnerability that T_{set} have experienced. The table also shows that SCAAT was activated only a few times after the initial re-mapping. Overall SCAAT was activated at 53% for *libquantum*, 75% for *streamcluster* and 93% for *swaptions*. In the case of *libquantum*, SCAAT is expected to be activated once for each vulnerability, however it was activated 110 times. The 10 additional activation occurrences by SCAAT happened due

to 10 tags being removed from the $SCAAT_mem$ after it reached full capacity. While the same LFSR initial value is used for all the benchmark re-mapping, T_{set} experienced different number of vulnerabilities for each benchmark. This can be attributed to the continuous change to the cache behavior, especially for RSCAAT.

Table 9: RSCAAT Security evaluation [63].

	SCAAT		RSCAAT	
	Vulnerabilities	Activations	Vulnerabilities	Activations
libquantum	211	110	189	189
streamcluster	142	106	139	139
swaptions	107	100	112	112

The elimination of this exploit is largely due to the unique design of the SR_CAM. While SCAAT is not activated for tags that have been previously re-mapped, the re-mapping function in the SR_CAM is activated regardless of a tag being re-mapped before or not. In order to generate a reliable list of $LFSR'$ sequences, the adversary must be able to distinguish which tag T_x were re-mapped previously to predict where the location S_y bit combination lays in respect to the current value of the LFSR. However, unlike SCAAT, RSCAAT continuously re-maps all tags that are targeted. This prevents an adversary of knowing how many times a tag was re-mapped. Making information regarding the location of a certain re-mapped tag in relation to the current value of the RSCAAT LFSR unattainable.

Impact on Performance

Figure 23 shows the hit rate of the benchmarks under test before (baseline) and after each benchmark is ran as a secure process. It is clear that using the SCAAT and RSCAAT contributes to performance degradation. For example, *libquantum* initially boasted a 95.29% hit rate. However, once it was executed as a secure process, the mitigation of detected leakage patterns reduced the hit rate to 70.44% for SCAAT and 61.06% when the RSCAAT was used.

Furthermore, while performance degradation is introduced to the system at different degrees for different benchmarks, all benchmarks suffered performance degradation. This can be largely attributed to the remapping of cache blocks that exhibited access patterns that may indicate cache information leakage. Each re-map will cause a miss either by remapping the cache block to an empty location or forcing eviction of another block, which in turn contributes to the performance degradation. Benchmarks with higher activation of the SCAAT and RSCAAT mechanisms suffered from higher degrees of performance degradation. The benchmark *gcc* had an average of two hundred thousand SCAAT/RSCAAT activation per run. Each activation is more likely to have caused a miss than a hit since *gcc* showed a reduction from a hit rate of 65.48% for baseline to 55.33% for SCAAT and 47.81% for RSCAAT.

Figure 24 shows the normalized average hit rate (where baseline hit rate = 100%) of each of the benchmarks after using the SCAAT and RSCAAT mitigation mechanisms.

Figure 25 shows Misses Per Kilo Instructions (MKPI) of the benchmarks under evaluation, comparing the baseline average MKPI to the average MKPI of each benchmark after the use of SCAAT and RSCAAT.

The figure 25 demonstrates an increase in the MKPI across all benchmarks, a trend corresponding to the decrease in hit rates. For instance, benchmarks such as *fft* exhibited a significant increase from an average MKPI of 10.84 at a hit rate of 69.91% to an average MKPI of 341.18 with hit rate reduction of 22.58% when SCAAT is used and an average MKPI

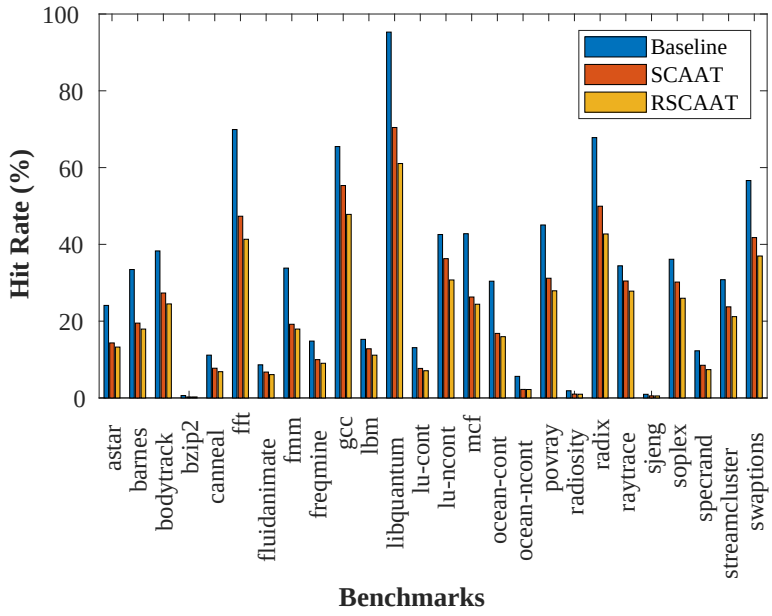


Figure 23: Resulting hit rate of each benchmark executed as a secure process before and after using the SCAAT and RSCAAT mitigation mechanisms [63].

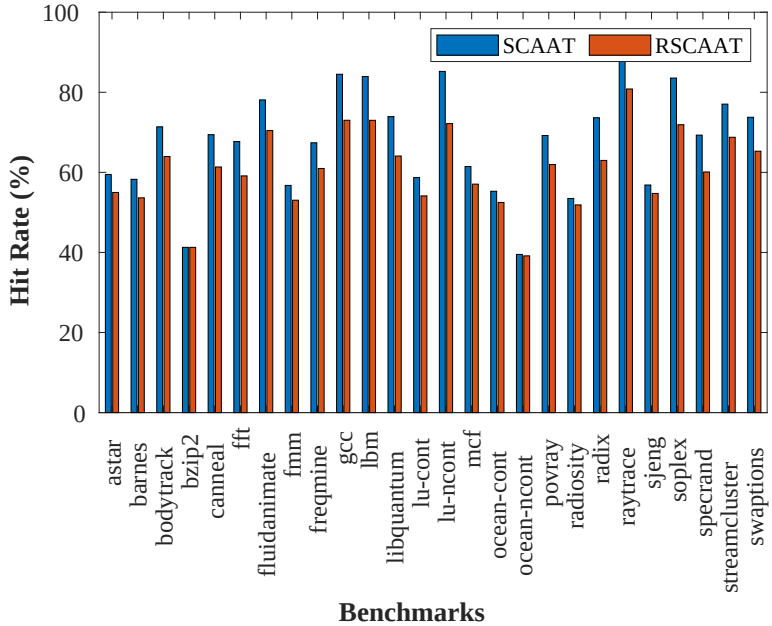


Figure 24: Normalized average hit rate of each benchmark executed after using the SCAAT and RSCAAT mitigation mechanisms. Baseline as 100% [63].

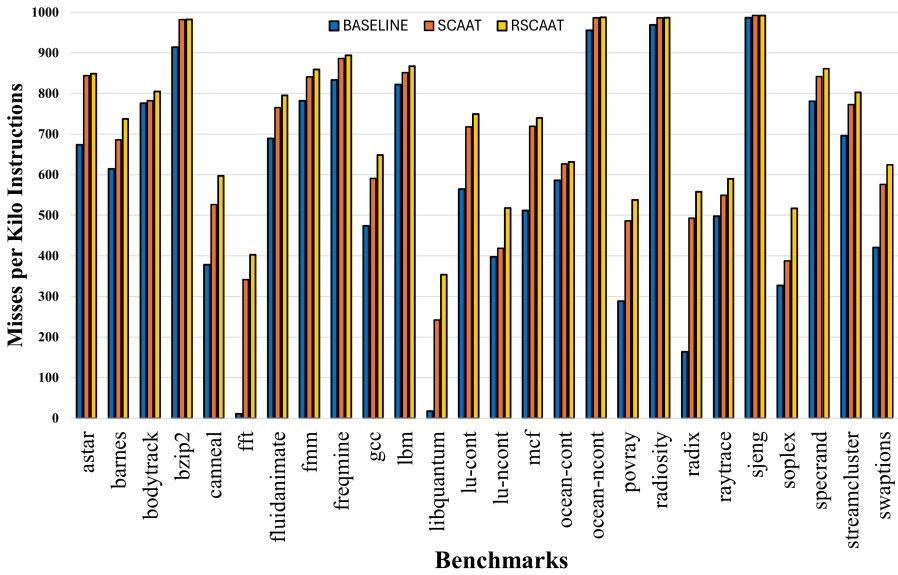


Figure 25: Average Misses per Kilo Instruction per benchmark before and after the integration of SCAAT and RSCAAT [63].

of 402.81 with hit rate decrease of 28.58% when RSCAAT is used. A similar trend is observed for other benchmarks such as *radix*. Other benchmarks such as *sjeng* and *radiosity* experienced a modest increase in the average MKPI. It is noteworthy that those benchmarks already demonstrated high MKPI prior to the implementation of either SCAAT or RSCAAT, which may explain the comparatively smaller impact observed.

The increase of MKPI across all benchmarks indicates a corresponding rise of Average Memory Access Time (AMAT). Since the LLC is shared in an inclusive cache hierarchy, an LLC miss incurs a main memory access and guarantees a higher AMAT for the executing workloads. An example of such effect can be deduced for the *fft* benchmark. The integration of RSCAAT into the cache system contributed a substantial increase in MKPI, which translated directly into increased memory latency. The increase of MKPI from 10.84 for baseline to 402.81 after RSCAAT implementation is approximately a 37$\times$ increase in the LLC MKPI. This increase indicates a transition from a relatively low memory-sensitive workload to a highly memory-bound workload. Given that LLC misses cause main memory access penalties on the order of hundreds of cycles, such an increase significantly amplifies the memory AMAT and, as a result, degrades the number of Instructions Per Cycle (IPC) that the cores can achieve. In practical terms, whereas the baseline configuration allows computation and memory access to overlap with limited stall impact, the post-RSCAAT system is dominated by long-latency DRAM accesses, causing execution throughput to collapse. These results demonstrate that RSCAAT significantly alters cache behavior, shifting performance from being core-limited to memory-limited, with LLC miss amplification as the primary driver of IPC degradation.

Area overhead

In order to evaluate and analyze the area overhead and the critical path delay overhead of the proposed RSCAAT in comparison to SCAAT and the baseline cache, several configurations were synthesized using the Synopsys synthesis suite [3] and AMS 0.18μm CMOS

technology library [2].

Table 10 indicate the area overhead incurred by both SCAAT and RSCAAT for several cache configurations. The percentages shown in the table are the area overhead incurred by SCAAT and RSCAAT compared to a baseline cache of the same configuration. The table shows that the area overhead decreases as the size of the cache increases. SCAAT added 4.3% overhead when integrated to a baseline 4-way, 256 cache line baseline cache. RSCAAT shows slightly higher overhead of 4.95% for the same cache configuration. Integrating SCAAT and RSCAAT to a slightly larger 8-way, 1024 cache line cache showed a decrease in the overhead percentage at 3.81% and 4.27% respectively. At the largest configuration of 16-way, 4096 cacheline, SCAAT overhead further decreased to 3.57% and RSCAAT decreased to 4.03% overhead.

Table 10: Area increase per cache configuration [63].

		# Cache Lines				
	# Ways	256	512	1024	2048	4096
SCAAT	4	4.3%	4.24%	4.05%		
	8	4.44%	4.10%	3.81%	3.78%	
	16				3.76%	3.57%
RSCAAT	4	4.94%	4.79%	4.60%		
	8	5.00%	4.77%	4.27%	4.27%	
	16				4.24%	4.03%

On average, SCAAT incurred a 4% area overhead while RSCAAT incurred 4.55% area increase. A steady decrease in area overhead is observed as the baseline cache capacity increases, indicating scalability at higher cache capacities. It is important to note that a single outlier can be seen for small size caches. Cache configurations with 256 cache lines showed an increase in overhead as the number of cache ways increase. This slight increase is most likely a synthesis artifact and does not have an effect on the overall scalability of SCAAT and RSCAAT.

The SR_CAM accounts for 93% of the overall size of the RSCAAT. Since the SR_CAM size correlates to the size of the baseline cache, as the size of the cache increases, the size of the SR_CAM would proportionately increase as well. A similar effect is observed in the SCAAT. However, in both RSCAAT and SCAAT, this proportionate increase only applies to their memories (SR_CAM and $SCAAT_mem$) and doesn't apply to the control logic of the size of the LFSR. For every additional power of 2 increase in the number of the cache lines, the LFSR width only increases by a single bit. While the control logic only increase relative to the width of the LFSR as it determines the dimensions of the decoder, the encoder, and hit indicator used in the SR_CAM and discussed in section 6.2.2.

7.3 Chapter Summary

This chapter evaluated the proposed CLD, SCAAT, and RSCAAT mechanisms in terms of detection capability, security, performance impact, and hardware overhead. The results showed that CLD achieved higher detection accuracy and substantially lower false positive rate than LiD-CAT while introducing negligible area overhead. The mitigation experiments showed that RSCAAT successfully addresses the vulnerability identified in the original SCAAT design by ensuring that vulnerable cache locations continue to be remapped when new threats are detected. Although both mitigation mechanisms affect cache hit rate and increase MPKI due to remapping-induced cache perturbations, the associated

hardware overhead remains modest and scales favorably with cache size. Overall, the results support the effectiveness of the proposed detection and mitigation framework for reducing exploitable cache leakage patterns at run time.

8 Conclusion

8.1 Summary and Conclusions

Logical Side-Channel Attacks (LSCAs) pose a growing threat to the security of modern cache systems. The design and implementation of cache systems can be improved to reduce the levels of this risk. This thesis examines how cache contention can be used both to identify exploitable cache leakage patterns and to activate mitigation mechanisms capable of reducing the risk of information leakage at run time. This thesis proposed a cache leakage detection mechanism and two mitigation mechanisms designed for shared, multi-level cache systems.

This thesis investigated the relationship between cache contention and logical side-channel leakage and addressed the following research questions:

1. Can cache-contention information be used to identify exploitable cache leakage patterns at run time?
2. How accurately can such leakage patterns be detected using readily available cache information?
3. Can run-time mitigation mechanisms be used to reduce the exploitability of cache leakage patterns?
4. What are the hardware and performance costs associated with such detection and mitigation mechanisms?

For detection of LSCAs, a lightweight run-time leakage detector, called Cache Leakage Detector (CLD) is presented. CLD operates during run-time by taking advantage of readily available cache outputs and process IDs with no dependencies and assumptions regarding an attackers' assumed behavior. A process-oriented threat model and a cache-contention-based leakage model are used to infer leakage templates capable of identifying exploitable cache access patterns. Experimental results show that CLD achieves a considerably higher detection accuracy and lower rate of over detection compared to a state-of-the-art run-time detector. Furthermore, CLD introduces negligible hardware overhead. These results demonstrate that cache contention can be utilized as an effective indicator of potential cache information leakage and logical side-channel attacks at run time.

For mitigation of identified leakage patterns, two mitigation mechanisms were proposed. The first mechanism, Secure Cache Alternative Address Table (SCAAT), has been shown to mitigate Logical Side-Channel Attacks by dynamically remapping cache accesses under the threat of detected leakage patterns to alternative cache locations. SCAAT introduces modest hardware overhead. SCAAT can be integrated with a wide range of direct-mapped and set-associative cache architectures without needing special modifications to the cache architecture. However, analysis of SCAAT revealed a vulnerability associated with repeated attacks on previously remapped cache blocks.

To address the limitation of SCAAT, a second mitigation mechanism, Recursive Secure Cache Alternative Address Table (RSCAAT), was developed. RSCAAT extends SCAAT to include recursive remapping logic and a Self-Remapping Content Addressable Memory (SR_CAM). These additions enable continuous re-randomization of cache locations associated with detected leakage patterns. Experimental results showed that RSCAAT successfully addressed the vulnerability identified in the original SCAAT design by consistently remapping vulnerable cache locations whenever leakage-like patterns were detected. While RSCAAT introduces performance overhead through its impact on cache occupancy and miss behavior, the results clearly demonstrate its effectiveness in reducing

the predictability of cache access patterns that could otherwise be exploited by an adversary.

Overall, the work presented in this thesis establishes the effectiveness and practicality of cache-contention-based leakage detection, and that run-time remapping mechanisms can leverage it to mitigate exploitable cache leakage patterns in shared cache systems. The proposed detection and mitigation mechanisms provide a foundation for integrating run-time leakage detection and mitigation directly into existing cache architectures with modest hardware overhead.

8.2 Future Research Directions

Several avenues remain for future work. While RSCAAT has been demonstrated to address the vulnerability identified in SCAAT, a security evaluation of the proposed SR_CAM is needed to investigate security properties against attacks targeting CAM-based structures. Furthermore, future work should focus on devising strong protection against logical side-channel attacks with reduced performance impact. Future work may also explore the formalization of the proposed threat model, tighter integration with operating-system-level security policies, and adaptive activation thresholds that balance security and performance according to workload behavior. Finally, broader statistical evaluation across additional workloads and cache organizations would further strengthen the characterization of the proposed detection and mitigation mechanisms.

List of Figures

1 Demonstration of a cache hit and a cache miss. 12
2 Demonstration of a possible cache contention scenario. 13
3 Problem and solution formulation. 14
4 Multi-level Cache Hierarchy. 18
5 Cache Address. 19
6 Demonstration of cache organization schemes for set-associative caches. .. 19
7 Example scenario of an Evict+Time attack. 20
8 Example scenario of a Prime+Probe attack. 21
9 Experimental methodology and data flow. 28
10 Generic cache architecture block diagram used in this work [63]. 31
11 CLD development steps [62]. 33
12 The state transition diagram of the CLD [62]. 36
13 A block diagram of the components and signals of the $SCAAT_unit$ [60]. .. 39
14 $SCAAT_unit$ control logic transition diagram [60]. 40
15 An example scenario demonstrating change in cache access pattern [60]. .. 40
16 Updated control logic transition diagram for the RSCAAT [63]. 44
17 Components of the Self-Remapping Content Addressable Memory (SR_CAM) [63]. 45
18 The structure of the hit indicator [63]. 46
19 The re-mapping redundancy transition diagram [63]. 47
20 Leakage detection histogram for all benchmarks. 52
21 Access pattern correlation with CLD and LiD-CAT detections [62]. 53
22 The distribution of number accesses per location in the LLC for the mcf benchmark [63]. 55
23 Resulting hit rate of each benchmark executed as a secure process before and after using the SCAAT and RSCAAT mitigation mechanisms [63]. 57
24 Normalized average hit rate of each benchmark executed after using the SCAAT and RSCAAT mitigation mechanisms. Baseline as 100% [63]. 57
25 Average Misses per Kilo Instruction per benchmark before and after the integration of SCAAT and RSCAAT [63]. 58

List of Tables

1 Attack formulas [21] analyzed in this work. 22
2 Comparison of representative cache attack detection and mitigation approaches. 26
3 System Configuration [63]. 29
4 Hit/miss combination of all leakage scenarios [62]. 35
5 The transition table of the CLD FSM [62]. 36
6 Size estimation for Randomization-based approaches [60]. 42
7 CLD's number and accuracy of attack detections injected into AES algorithm. 54
8 CLD's area overhead for all cache configurations [62]. 54
9 RSCAAT Security evaluation [63]. 56
10 Area increase per cache configuration [63]. 59

References

[1] Intel® Celeron® Processor J4125 (4M Cache, up to 2.70 GHz) - Product Specifications | Intel — intel.com. https://www.intel.com/content/www/us/en/products/sku/197305/intel-celeron-processor-j4125-4m-cache-up-to-2-70-ghz/specifications.html. [Accessed 24-Apr-2023].

[2] Ams process technology – 0.18µm cmos and other foundry technologies. https://ams.com/process-technology, 2026. Accessed: 2026-04-02.

[3] Synopsys design compiler: Timing, area, power, & test optimization. https://www.synopsys.com/Tools/Implementation/RTLSynthesis/DesignCompiler/Pages/default.aspx, 2026. Accessed: 2026-04-02.

[4] O. Acıiçmez and Ç. K. Koç. Trace-driven cache attacks on aes (short paper). In *International Conference on Information and Communications Security*, pages 112–121. Springer, 2006.

[5] M. Alam, S. Bhattacharya, D. Mukhopadhyay, and S. Bhattacharya. Performance counters to rescue: A machine learning based safeguard against micro-architectural side-channel-attacks. volume 2017, page 564, 2017.

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

[7] J.-L. Baer and W.-H. Wang. On the inclusion properties for multi-level cache hierarchies. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, ISCA '88, page 73–80, Washington, DC, USA, 1988. IEEE Computer Society Press.

[8] J.-L. Baer and W.-H. Wang. On the inclusion properties for multi-level cache hierarchies. *SIGARCH Comput. Archit. News*, 16(2):73–80, May 1988.

[9] M.-M. Bazm, T. Sautereau, M. Lacoste, M. Sudholt, and J.-M. Menaud. Cache-based side-channel attacks detection through intel cache monitoring technology and hardware performance counters. pages 7–12, 04 2018.

[10] D. J. Bernstein. Cache-timing attacks on aes. Technical report, 2005.

[11] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[12] A. Bogdanov, T. Eisenbarth, C. Paar, and M. Wienecke. *Differential Cache-Collision Timing Attacks on AES with Applications to Embedded CPUs*, pages 235–251. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[13] S. Briongos, G. Irazoqui, P. Malagón, and T. Eisenbarth. Cacheshield: Protecting legacy processes against cache attacks. *CoRR*, 2017.

[14] Cadence Design Systems, Inc., San Jose, CA, USA. *Genus Synthesis Solution*, 2026. Available at: https://www.cadence.com.

[15] Chair of VLSI Design, Diagnostics and Architecture. PoC - Pile of Cores, 2016.

[16] S. Chattopadhyay, M. Beck, A. Rezine, and A. Zeller. Quantifying the information leakage in cache attacks via symbolic execution. 18(1), 2019.

[17] S. Chattopadhyay and A. Roychoudhury. Symbolic verification of cache side-channel freedom. *TCAD*, 2018.

[18] Chi Xu, Xi Chen, R. P. Dick, and Z. M. Mao. Cache contention and application performance prediction for multi-core systems. In *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, pages 76–86, 2010.

[19] M. Chiappetta, E. Savas, and C. Yilmaz. Real time detection of cache-based side-channel attacks using hardware performance counters. volume 49, pages 1162 – 1174, 2016.

[20] V. Costan, I. Lebedev, and S. Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 857–874, 2016.

[21] S. Deng, W. Xiong, and J. Szefer. Cache timing side-channel vulnerability checking with computation tree logic. In *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '18, New York, NY, USA, 2018. Association for Computing Machinery.

[22] S. Deng, W. Xiong, and J. Szefer. Analysis of secure caches using a three-step model for timing-based attacks. *Journal of Hardware and Systems Security*, 3(4):397–425, 2019.

[23] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM TACO*, Jan. 2012.

[24] J.-F. Gallais, I. Kizhvatov, and M. Tunstall. Improved trace-driven cache-collision attacks against embedded aes implementations. In *International Workshop on Information Security Applications*, pages 243–257. Springer, 2010.

[25] C. P. García and B. B. Brumley. {Constant-Time} callees with {Variable-Time} callers. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 83–98, 2017.

[26] D. Genkin, R. Poussier, R. Q. Sim, Y. Yarom, and Y. Zhao. Cache vs. key-dependency: Side channeling an implementation of pilsung. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 231–255, 2020.

[27] T. Ghasempouri, J. Raik, K. Paul, C. Reinbrecht, S. Hamdioui, and M. Taouil. A security verification template to assess cache architecture vulnerabilities. In *2020 23rd International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pages 1–6, 2020.

[28] L. Giner, S. Steinegger, A. Purnal, M. Eichlseder, T. Unterluggauer, S. Mangard, and D. Gruss. Sasscache: A secure skewed-associative cache with keyed index mapping. In *44th IEEE Symposium on Security and Privacy (S&P 2023)*, 2023.

[29] B. Gras, K. Razavi, H. Bos, C. Giuffrida, et al. Translation leak-aside buffer: Defeating cache side-channel protections with tlb attacks. In *USENIX Security Symposium*, volume 216, 2018.

[30] D. Gruss, C. Maurice, K. Wagner, and S. Mangard. Flush+flush: A fast and stealthy cache attack. In J. Caballero, U. Zurutuza, and R. J. Rodríguez, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299, Cham, 2016. Springer International Publishing.

[31] D. Gruss, R. Spreitzer, and S. Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, pages 897–912, 2015.

[32] M. R. Guthaus, J. E. Stine, S. Das, G. Chen, N. Thaluru, and K. Agrawal. Openram: An open-source memory compiler. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–6, 2016.

[33] D. M. Harris and S. L. Harris. Chapter 8 - memory systems. In D. M. Harris and S. L. Harris, editors, *Digital Design and Computer Architecture*, pages 463–512. Morgan Kaufmann, Burlington, 2007.

[34] Z. He and R. Lee. How secure is your cache against side-channel attacks? In *MICRO 2017 - 50th Annual IEEE/ACM International Symposium on Microarchitecture Proceedings*, Proceedings of the Annual International Symposium on Microarchitecture, MICRO, pages 341–353, United States, Oct. 2017. IEEE Computer Society. 50th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2017 ; Conference date: 14-10-2017 Through 18-10-2017.

[35] Z. He and R. B. Lee. How secure is your cache against side-channel attacks? In *MICRO*, 2017.

[36] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, Amsterdam, 5 edition, 2012.

[37] M. Hill. A case for direct-mapped caches. *Computer*, 21(12):25–40, 1988.

[38] Intel Corporation. Intel486 DX Microprocessor. `http://datasheets.chipdb.org/Intel/x86/486/datashts/240440-006.pdf`.

[39] International Business Machines Corporation. IBM System/360 Model 85 Functional Characteristics. `http://www.bitsavers.org/pdf/ibm/360/functional_characteristics/A22-69161_360-//85_funcChar_Jun68.pdf`.

[40] G. Irazoqui, T. Eisenbarth, and B. Sunar. S $ a: A shared cache attack that works across cores and defies vm sandboxing–and its application to aes. In *2015 IEEE Symposium on Security and Privacy*, pages 591–604. IEEE, 2015.

[41] G. Keramidas, A. Antonopoulos, D. N. Serpanos, and S. Kaxiras. Non deterministic caches: A simple and effective defense against side channel attacks. *Design Automation for Embedded Systems*, 12(3):221–230, 2008.

[42] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer. Dawg: A defense against cache timing attacks in speculative execution processors. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 974–987. IEEE, 2018.

[43] 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 *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[44] B. Köpf, L. Mauborgne, and M. Ochoa. Automatic quantification of cache side-channels. In P. Madhusudan and S. A. Seshia, editors, *Computer Aided Verification*, pages 564–580, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[45] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[46] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *2016 IEEE international symposium on high performance computer architecture (HPCA)*, pages 406–418. IEEE, 2016.

[47] 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.

[48] S. Mittal. A survey of techniques for designing and managing cpu register file. *Concurrency and Computation: Practice and Experience*, 29(4):e3906, 2017.

[49] F. Mueller. Timing predictions for multi-level caches. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, pages 29–36. Citeseer, 1997.

[50] R. Murphy. On the effects of memory latency and bandwidth on supercomputer application performance. In *2007 IEEE 10th International Symposium on Workload Characterization*, pages 35–43. IEEE, 2007.

[51] M. Mushtaq, A. Akram, M. K. Bhatti, M. Chaudhry, V. Lapotre, and G. Gogniat. Nightswatch: A cache-based side-channel intrusion detector using hardware performance counters. In *HASP*, 2018.

[52] B. Niazmand, S. P. Azad, G. Jervan, and J. Sepúlveda. Design and verification of secure cache wrapper against access-driven side-channel attacks. In *2019 22nd Euromicro Conference on Digital System Design (DSD)*, pages 672–676. IEEE, 2019.

[53] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of aes. In *Cryptographers' track at the RSA conference*, pages 1–20. Springer, 2006.

[54] B. Pichai, L. Hsu, and A. Bhattacharjee. Architectural support for address translation on gpus: Designing memory management units for cpu/gpus with unified address spaces. *ACM SIGARCH Computer Architecture News*, 42(1):743–758, 2014.

[55] 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.

[56] C. Reinbrecht, S. Hamdioui, M. Taouil, B. Niazmand, T. Ghasempouri, J. Raik, and J. Sepúlveda. Lid-cat: A lightweight detector for cache attacks. In *2020 IEEE European Test Symposium (ETS)*, pages 1–6, 2020.

[57] E. Ronen, R. Gillham, D. Genkin, A. Shamir, D. Wong, and Y. Yarom. The 9 lives of bleichenbacher's cat: new cache attacks on tls implementations. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 435–452. IEEE, 2019.

[58] G. Saileshwar and M. K. Qureshi. Mirage: Mitigating conflict-based cache attacks with a practical fully-associative design. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security 21)*, pages 1765–1782. USENIX Association, Aug. 2021.

[59] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros. Splash-3: A properly synchronized benchmark suite for contemporary research. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 101–111, 2016.

[60] 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.

[61] 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.

[62] A. Shalabi, T. Ghasempouri, P. Ellervee, and J. Raik. Cld: An accurate, cost-effective and scalable run-time cache leakage detector. In *2021 24th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, pages 127–132. IEEE, 2021.

[63] A. Shalabi, T. Ghasempouri, P. Ellervee, and J. Raik. Rscaat: A recursive mechanism to mitigate cache side channel attacks. *IEEE Access*, 14:52342–52359, 2026.

[64] A. Shalabi, K. Paul, T. Ghasempouri, and J. Raik. Nv-sp: A new high performance and low energy nvm-based scratch pad. In *2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 54–59. IEEE, 2020.

[65] A. Smith. A comparative study of set associative memory mapping algorithms and their use for cache and main memory. *IEEE Transactions on Software Engineering*, SE-4(2):121–130, 1978.

[66] A. J. Smith. Cache memories. *ACM Computing Surveys (CSUR)*, 14(3):473–530, 1982.

[67] W. Song, H. Ma, Z. Wang, D. Xie, and C. Ouyang. Gir-cache: Mitigating conflict-based cache side-channel attacks via global indirect replacement. In *Proceedings of the Information Security Conference (ISC) 2025*, 2025.

[68] C. D. Spradling. SPEC CPU2006 Benchmark Tools. *SIGARCH Computer Architecture News*, 35, March 2007.

[69] S. C. F. L. C. R. Thomas Unterluggauer, Austin Harris. Chameleon cache: Approximating fully associative caches with random replacement to prevent contention-based cache attacks. *arXiv preprint arXiv:2209.14673*, 2022.

[70] E. Tromer, D. A. Osvik, and A. Shamir. Efficient cache attacks on aes, and countermeasures. *Journal of Cryptology*, 2010.

[71] Y. Tsunoo. Cryptanalysis of block ciphers implemented on computers with cache. *preproceedings of ISITA 2002*, 2002.

[72] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli. Multi2Sim: A Simulation Framework for CPU-GPU Computing . In *Proc. of the 21st International Conference on Parallel Architectures and Compilation Techniques*, Sep. 2012.

[73] G. Van Rossum. *The Python Library Reference, release 3.8.2*. Python Software Foundation, 2020.

[74] Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. *ACM SIGARCH Computer Architecture News*, 35(2):494–505, 2007.

[75] Z. Wang and R. B. Lee. A novel cache architecture with enhanced performance and security. In *2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 83–93. IEEE, 2008.

[76] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard. Scattercache: Thwarting cache attacks via cache set randomization. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 675–692, Santa Clara, CA, Aug. 2019. USENIX Association.

[77] M. Yan, B. Gopireddy, T. Shull, and J. Torrellas. Secure hierarchy-aware cache replacement policy (sharp): Defending against cache-based side channel attacks. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 347–360. IEEE, 2017.

[78] 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, pages 719–732, Berkeley, CA, USA, 2014. USENIX Association.

[79] Y. Yarom, D. Genkin, and N. Heninger. Cachebleed: A timing attack on openssl constant time RSA. In *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, pages 346–367, 2016.

[80] B. B. Yilmaz, R. Callan, M. Prvulovic, and A. Zajić. Quantifying information leakage in a processor caused by the execution of instructions. In *MILCOM 2017-2017 IEEE Military Communications Conference (MILCOM)*, pages 255–260. IEEE, 2017.

[81] D. Zhang, A. Askarov, and A. C. Myers. Language-based control and mitigation of timing channels. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 99–110, 2012.

[82] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers. A hardware design language for timing-sensitive information-flow security. *Acm Sigplan Notices*, 50(4):503–516, 2015.

[83] T. Zhang and R. B. Lee. New models of cache architectures characterizing information leakage from cache side channels. New York, NY, USA, 2014. Association for Computing Machinery.

[84] T. Zhang, Y. Zhang, and R. Lee. Cloudradar: A real-time side-channel attack detection system in clouds. 2016.

[85] Q. Zhao, D. Koh, S. Raza, D. Bruening, W.-F. Wong, and S. Amarasinghe. Dynamic cache contention detection in multi-threaded applications. *SIGPLAN Not.*, 46(7):27–38, Mar. 2011.

Acknowledgements

To my family. Without them, I could have never became who I am today.

To my supervisors: your guidance was instrumental to many of my accomplishments. Your patience and persistence, for which I am deeply thankful, gave light to this work.

Abstract
On the Detection and Mitigation of Cache Logic Side Channel Attacks

Concerns about the security of memory systems have increased over the past decade due to their vulnerabilities being easily exploited. Multi-level caching is introduced to increase computing performance by alleviating the speed difference between the processors and memory. However, this has created vulnerabilities that can be exploited by logical side-channel attacks. While there exist solutions that perform run-time remapping of data under threat to randomized locations, such solutions exhibit vulnerabilities in case of repetitive attacks. This thesis aims to address this issue.

LSCAs are a significant threat to modern computer systems. They exploit normal cache behavior to leak information by taking advantage of cache contention. This cache behavior and the observation of processes' access patterns to the cache, detection of those leakages is possible during run time. To realize this goal, a detection template is needed to leverage available run time cache outputs and process access information. This thesis proposes a template and it's hardware implementation in the shape a cache hardware monitor called CLD. CLD is a highly accurate, low-cost, scalable run-time cache information leak detector. CLD utilizes cache signals and process IDs to identify information leakage patterns and warn when those patterns can be potentially exploitable cache access patterns. Accuracy of CLD is evaluated by using several benchmarks and injecting attacks into a 128-bit key AES algorithm. The experiments demonstrate that CLD has far higher detection accuracy (0.7964 vs 0.3195) and lower percentage of false positive detections (1.2% vs 30.6%) compared to a state-of-the-art hardware detector. Moreover, CLD introduces a very low area overhead of 0.002% to the total area of the cache.

While screening the cache access patterns at runtime is one promising approach to protect against LSCAs, designing an environment capable of detecting and mitigating these attacks is difficult. Current monitoring systems have largely neglected finding a mitigation mechanism to address the attacks. This thesis presents two mitigation mechanisms to address those shortcomings.

The first is SCAAT, a cache attack mitigation system with a remapping functionality that handles attacks by remapping vulnerable data to random location in the cache during run time. It does so by using signals received from a monitoring system that identifies suspicious behavior. SCAAT is a system that can be added to any direct-mapped or k-way associative cache with modest area overhead, ranging from 12.77% to 13.58% for direct-mapped caches and from 3.40% to 3.96% for k-way associative caches. It is capable of changing the cache behavior at run time, eliminating any predictable patterns that an adversary can take advantage of to perform attacks. The second mitigation mechanism is RSCAAT, a run-time mitigation mechanism with a robust control logic and high reactivity to cache behavior. It features a specialized CAM with built-in re-mapping redundancies capable of eliminating predictable pseudorandom traces. RSCAAT further strengthens the mitigation process by recursively re-mapping vulnerable data whenever leakage-like patterns are detected. Experimental results showed that RSCAAT was activated for 100% of the observed leakage-like patterns, compared to activation rates between 53% and 93% for existing approaches depending on the workload. This capability is particularly important when attacks are persistent and targeted data continues to be accessed by the system.

Both SCAAT and RSCAAT were evaluated using benchmarks from several suites to test their effectiveness with different types workload behaviors. Results show that while the overhead of SCAAT and RSCAAT in performance is comparable to state-of-the-art, they

introduce significantly higher levels of security to the cache by accurately eliminating leakage-like patterns as well as recursively remapping vulnerable accesses.

Kokkuvõte
Vahemälu kõrvalkanali rünnete avastamine ja leevendamine

Mure mälusüsteemide turvalisuse pärast on viimase kümnendi jooksul suurenenud, kuna nende haavatavusi on lihtne ära kasutada. Mitmetasandiline vahemälu on kasutusele võetud, et suurendada arvutusvõimsust, leevendades protsessorite ja mälu kiiruse erinevust. See on aga loonud haavatavusi, mida saavad ära kasutada kõrvalkanali ründed. Kuigi on olemas lahendusi, mis teostavad ohu korral andmete ümberpaigutamist käitusajal juhuslikesse asukohtadesse, on sellised lahendused haavatavad korduvate rünnakute korral. Käesolev väitekiri püüab seda probleemi lahendada. Kõrvalkanali ründed on märkimisväärne oht tänapäevastele arvutisüsteemidele. Nad kasutavad ära tavapärast vahemälu käitumist, et lekitada teavet, kasutades ära protsesside võistlust vahemälu pärast. Selline vahemälu käitumine ja protsesside juurdepääsumustrite jälgimine võimaldab lekete tuvastamist käitusajal. Eesmärgi saavutamiseks on tarvis tuvastusmalli, et kasutada olemasolevaid käitusaja vahemälu väljundeid ja protsesside juurdepääsuteavet. Käesolev lõputöö pakub välja malli ja riistvaralise realisatsiooni vahemälu riistvaramonitorile nimega CLD. CLD on ülitäpne, odav ja skaleeritav käitusaegne vahemälu infolekke detektor. CLD kasutab vahemälu signaale ja protsessi ID-sid infolekke mustrite tuvastamiseks ja hoiatamiseks, kui need mustrid võivad olla potentsiaalselt ärakasutatavad vahemälu juurdepääsumustrid. Töös on CLD täpsust hinnatud mitme võrdlusaluse rakenduse abil rünnete sisestamisega 128-bitise võtmega AES-algoritmile. Katsed näitavad, et CLD-l on palju suurem tuvastustäpsus (0,7964 vs 0,3195) ja madalam valepositiivsete tuvastuste protsent (1,2

Kuigi vahemälu juurdepääsu mustrite skriinimine käitusajal on üks paljulubav lähenemisviis kõrvalkanali rünnete eest kaitsmiseks, on keeruline luua keskkonda, mis on võimeline neid ründeid tuvastama ja leevendama. Praegused jälgimissüsteemid on suures osas unarusse jätnud rünnetega toime tulemiseks mõeldud leevendusmehhanismi leidmise. Käesolev väitekiri esitab kaks leevendusmehhanismi nende puuduste kõrvaldamiseks.

Esimene neist on SCAAT, vahemälu rünnete leevendussüsteem, millel on ümberkaardistamise funktsioon, mis tegeleb rünnetega, paigutades haavatavad andmed ümber juhuslikele asukohtadele vahemälus käitusaja jooksul. Selleks kasutatakse signaale, mis saadakse jälgimissüsteemilt, mis tuvastab kahtlase käitumise. SCAAT on süsteem, mida saab lisada mistahes otsekaardistatud või k-tee assotsiatiivsele vahemälule ning sellel on mõõdukas pindala, mis jääb vahemikku 12,77% kuni 13,58% otsekaardistatud vahemälude puhul ja 3,40% kuni 3,96% k-tee assotsiatiivsete vahemälude puhul. SCAAT on võimeline muutma vahemälu käitumist käitusaja jooksul, kõrvaldades kõik ennustatavad mustrid, mida vastane saab rünnakute sooritamiseks ära kasutada. Teine leevendusmehhanism on RSCAAT, käitusaegne leevendusmehhanism, millel on tõhus juhtimisloogika ja kõrge reageerimisvõime vahemälu käitumisele. See sisaldab spetsiaalset kontekstipõhist mälu sisseehitatud ümberpaigutamisega, mis on võimeline kõrvaldama ennustatavaid pseudojuhuslikke jadasid. RSCAAT tugevdab veelgi leevendusprotsessi, kaardistades haavatavaid andmeid rekursiivselt ümber iga kord, kui tuvastatakse lekkelaadseid mustreid. Eksperimentaalsed tulemused näitasid, et RSCAAT aktiveerus 100% täheldatud lekkelaadsetest mustritest, võrreldes olemasolevate lähenemisviiside aktiveerimismääradega 53%–93%, olenevalt töökoormusest. See võimekus on eriti oluline, eriti kui ründed on püsivad ja süsteem pääseb jätkuvalt juurde sihitud andmetele.

Nii SCAAT-i kui ka RSCAAT-i hinnati mitme komplekti rakenduste abil, et testida nende tõhusust erinevat tüüpi töökoormuse käitumisviiside korral. Tulemused näitavad, et kuigi SCAAT-i ja RSCAAT-i jõudluskulud on võrreldavad tipptaseme leevendusmeetodite omadega, lisavad nad vahemälule oluliselt kõrgema turvalisuse taseme, kõrvaldades täpselt lekkelaadsed mustrid ja kaardistades haavatavaid juurdepääsukohti rekursiivselt ümber.

Appendix 1 - SCAAT: Secure cache alternative address table for mitigating cache logical side-channel attacks.

I

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

SCAAT: Secure Cache Alternative Address Table for mitigating cache logical side-channel attacks

Ameer Shalabi, Tara Ghasempouri, Peeter Ellervee, Jaan Raik
Department of Computer Systems, Tallinn University of Technology, Tallinn, Estonia
{ameer.shalabi,tara.ghasempouri, peeter.ellervee, jaan.raik}@taltech.ee

***Abstract*—Interest in memory systems' security has increased during the last decade due to their vulnerabilities to be exploited by logical side channels attacks. A promising approach for attack detection at run-time is to monitor the cache memory's behavior. However, designing an environment capable of detecting and mitigating these attacks is very challenging. In current monitoring systems, attack mitigation has been largely neglected. To overcome these shortcomings, in this work, we present a secure cache called SCAAT. SCAAT is equipped with an attack mitigation system to handle attacks by remapping where data is stored in the cache to random locations. In addition, SCAAT uses an attack monitor that identifies suspicious behavior that indicates cache logical side-channel attacks. The effectiveness of SCAAT is analyzed and evaluated for several cache configurations in terms of area overhead and performance.**

I. Introduction

Modern digital systems contain significant information that requires high security. In recent years, a large variety of attacks against ICs and memories have been reported. The vulnerability of caches is caused by their characteristics in resource sharing because cache states affect and are affected by all processes. Therefore, one process can infer the cache usage of another process through cache contention [1].

Today, software attacks can compromise cache security through so-called Logical Side-Channel Attacks (LSCA). Adversaries can use these attacks to understand the system's secrets by simply observing its behavior. Several examples of cache attacks, reported in recent years, [2–6] represent a serious threat for the semiconductor industry. Patching defenses in the field may be very costly, degrade the performance, and sometimes even create new issues (e.g. reduced battery lifetime of an IoT device) [7]. Therefore, there is a strong need for attack monitoring and mitigating systems for the cache memories.

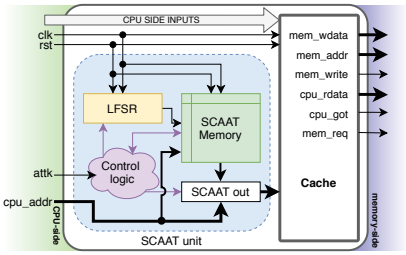


Fig. 1: Block diagram illustrating SCAAT system

Although cache monitoring has been widely studied in the context of high-performance computing, its use for mitigating cache attacks has been almost completely neglected. A limited number of publications has performed offline and online cache monitoring [8–14] for security purposes. However, none of them proposes a solution to ease and handle the attacks after the detection phase.

To this end, some works have proposed mechanisms to obtain secure caches. For instance, [15] proposes a cache-assisted secure execution system called CaSE that can protect against both software attacks and physical memory disclosure attacks on ARM-based devices. In [16], a methodology is proposed to conceal the plain text data from the L2 cache memory. It pursues to make the data unusable in the case an attacker retrieves it, and to avoid the transfer of regular patterns between CPU and cache.

On one hand, approaches in [17–19] use static partitioning where the cache is physically divided into different partitions eliminating cache interference among different applications. However, these approaches cause performance degradation. On the other hand, dynamically-partitioned caches determine the partition size dynamically at run-time. However, dynamic partitioning techniques are still vulnerable to some LSCAs. Examples of dynamically-partitioned caches can be found in [20–24].

Other approaches to gain secure caches are to randomize the memory-to-cache mapping at run-time so that an attacker cannot extract useful information from observing cache contention. However, such cache is slow and power-hungry [1]. I addition, [25] presented a mitigation approach for access-driven side-channel attacks. While the approach in [25] was successful in mitigating access-driven attacks, the mitigation strategy aimed at randomizing all accesses to the cache without directly targeting attacks, which can lead to the cache being exploited using other types of LSCAs.

All in all, in the current approaches, side-channel attacks' monitoring is not coupled with an attack mitigation mechanism and they are mainly limited to alarming the users about the possibility of data leakages.

To overcome the above shortcomings, this work proposes an innovative secure cache called the Secure Cache Alternative Address Table (SCAAT). The SCAAT is a mitigation system that is connected to a monitoring system to detect an attack and mitigate it by remapping where data is stored in the cache to random locations at run time. SCAAT mitigation strategy aims to change the behavior patterns of the cache making it harder to predict and to use these patterns to leak cache behavior-related information.

The main contributions of this paper are as follows:

- Combine a monitoring system for cache memories to detect cache LSCAs with a mitigation system to handle the attacks.

978-1-7281-9535-3/20/$31.00 ©2020 IEEE
DOI 10.1109/DSD51259.2020.00043

- Introducing a Secure Cache Alternative Address Table memory called SCAAT that remaps cache addresses to mitigate cache side-channel attacks.
- Analyzing the change of behavior of three benchmarks when executed with SCAAT to observe the false positives generated by the monitoring system and the effectiveness of SCAAT mitigation strategy.
- Analyzing the performance and area of SCAAT compared to a baseline cache.

The rest of this paper is organized as follows. Section II provides the required preliminary information. Section III presents the proposed methodology. Section IV shows the experimental results. Finally, Section V concludes this paper.

II. Preliminaries

For the purpose of testing the SCAAT, the open-source IP core cache [26] was used. The reasoning behind this choice is the parameterizable and generic set-associative nature of the cache. Relevant parameters, their calculations, and constant values used to configure the cache for this work are listed in Table I. A more detailed documentation of the cache design and operations is found in [27].

Furthermore, SCAAT uses the monitor proposed in [28]. When an attack is detected by the monitor, an attack signal $attk$ is sent to the SCAAT unit for attack mitigation. Once the attack is mitigated, the SCAAT unit output is sent to the cache as a new address.

TABLE I: Cache parameters and their definitions.

Parameter	Definition
CACHE_LINES	Number of cache lines.
ASSOCIATIVITY	Number of ways per cache set.
CACHE_SETS	Number of cache sets = $CACHE_LINES/ASSOCIATIVITY$
CPU_ADDR_BITS	Bit-width of the address sent to the cache from CPU side.
MEM_ADDR_BITS	Bit-width of the memory address sent to the main memory from the cache.
CPU_DATA_BITS	Bit-width of cache to processor (CPU) data interface.
MEM_DATA_BITS	Bit-width of cache to the main memory data interface.
INDEX_BITS	Bit-width of index bits from the CPU address identifying the cache set where tag with TAG_BITS is being accessed. = $log_2(CACHE_SETS)$
TAG_BITS	Bit-width of tag bits from the CPU address identifying the cache block being accessed. = $CPU_ADDR_BITS - INDEX_BITS$
OFFSET_BITS	Bit-width of offset bits from the CPU address identifying which word within a cache line is being accessed. = $MEM_DATA_BITS/CPU_DATA_BITS$

III. Proposed Methodology

In this section, we propose a methodology to develop the SCAAT system: a secure cache coupled with a monitoring system to detect the attacks and a mitigating system to ease the attacks when they occur. First, a description of the functionality of the SCAAT system is provided. Second, we describe the SCAAT mitigation strategy for the detected attacks. Third, the SCAAT impact on the behavior of the cache is discussed.

A. Attack mitigation functionality

To mitigate the attacks detected by the monitoring system, the proposed SCAAT is divided into two main subsystems. The first subsystem is a direct-mapped cache or k-way associative cache. The second, a SCAAT unit, which employs a specialized control logic, a specialized memory, and an LFSR. Fig. 1 shows the structure of the SCAAT when connected to a generic direct-mapped or k-way associative cache. The SCAAT unit requires four inputs: clk and rst signals for synchronizing the SCAAT with the cache, the $attk$ signal received from the monitor to signal when an attack has occurred, and the cpu_addr (cache address) being accessed by the CPU. The SCAAT unit has one output: $SCAAT_out$ that is directly connected to the cache address port.

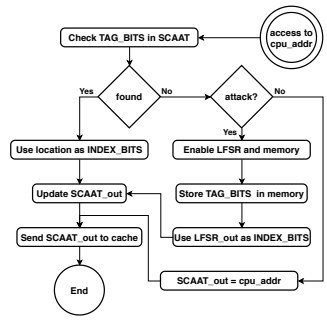


Fig. 2: SCAAT unit control logic transition diagram.

Inputs to the SCAAT unit are used to control and operate two components connected to the control logic:

- The LFSR: used to generate pseudo-random bit-vectors of length equal to the $INDEX_BITS$ parameter.
- The SCAAT memory: a memory module with a single write port and a single read port. It has a data width equal to TAG_BITS and depth equal to $INDEX_BITS$. The SCAAT memory stores the TAG_BITS of cpu_addr at location $LFSR_out$. If the input data matches any of the contents of SCAAT memory, the location of the matching data is sent as $SCAAT_mem_out$. An additional bit, added as the most significant bit, indicates if the data is found (represented by a logical '1') or not found (represented by a logical '0') in SCAAT memory. This bit is called $found_in_SCAAT$ and is used by the control logic.

Fig. 2 illustrates the control logic transition diagram of the SCAAT unit. The cpu_addr must go through the SCAAT unit regardless of the occurrence of an attack. This guarantees that all communication between the CPU and the cache is being checked by the SCAAT unit. If no attack is detected and SCAAT memory does not contain TAG_BITS of cpu_addr, the SCAAT unit is not activated and both the LFSR and the SCAAT memory are not enabled. In this case, cpu_addr is propagated through the SCAAT unit unchanged.

As shown in Fig. 2, both the LFSR and the SCAAT memory are only enabled if an attack is detected by the monitor AND the tag was not found in the SCAAT memory. When an attack occurs on tag T (i.e. TAG_BITS of cpu_addr), control logic checks if tag T has been found in SCAAT memory, if it is, its location in memory is used as $INDEX_BITS$ of $SCAAT_out$. If tag T is not in the SCAAT memory, the LFSR and the SCAAT memory are enabled. $LFSR_out$ is used as SCAAT memory location and Tag T is used as memory input data. At the end of the current clock cycle, the LFSR generates a new pseudo-random number to be used in case of future attacks.

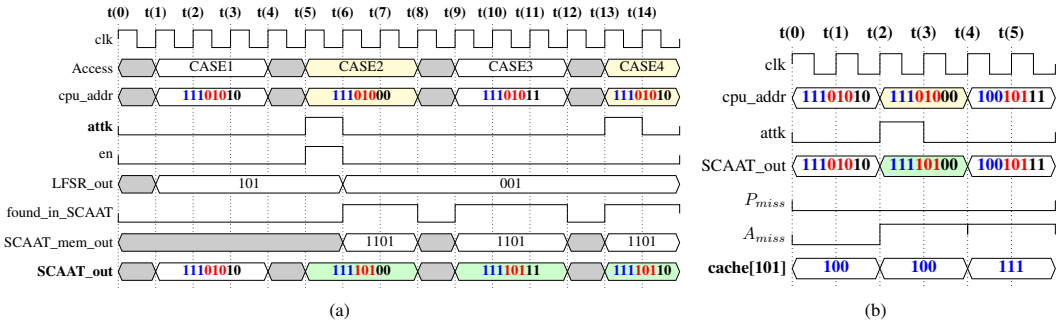


Fig. 3: Timing waveform for (a) SCAAT unit when a block is accessed four times and (b) an example scenario demonstrating change in cache access pattern.

B. Attack mitigation strategy

The timing waveform in Fig. 3a illustrates signals and data transition and propagation through the SCAAT unit. The scenario represented in Fig. 3a shows the four possible access cases to the SCAAT. Cache is accessed using 3-bit TAG_BITS (highlighted in blue), 3-bit $OFFSET_BITS$ (highlighted in red), and 2-bit $INDEX_BITS$ (highlighted in black). A block referenced by the tag 111 is accessed four times. A single clock cycle is 1 unit of time u donated by the function $t(u)$. Accesses under attack are highlighted in yellow. The generated output from SCAAT unit activation is highlighted in green. This example assumes that a write access takes 3 clock cycles (accesses $CASE1$, $CASE2$, and $CASE3$), while a read access takes 2 clock cycles (access $CASE4$).

$CASE1$ access to the block 111 is performed. No attacks were detected and tag 111 is not found in SCAAT memory, therefore cpu_addr is propagated through the SCAAT unit unchanged.

$CASE2$ is the next access to block with tag 111. This access was found to be under attack by the monitor. Since this is the first ever attack on cpu_addr with tag 111, the LFSR and SCAAT unit must be enabled. The current content of the LFSR register is used as $INDEX_BITS$ of $SCAAT_out$. This guarantee that cpu_addr is updated without the need to wait for tag 111 to be stored in SCAAT memory. The LFSR register is updated with the new value and SCAAT memory stores tag 111 at location 101. At the beginning of $t(6)$, SCAAT memory is checked for tag 111, which was stored in the previous clock cycle. At $t(6)$, the MSB of $SCAAT_mem_out$ indicates that tag 111 is found in memory at location 101. Once tag 111 is found in memory, the signal $found_in_SCAAT$ is set to '1' and location 101 is used as $INDEX_BITS$ of $SCAAT_out$.

$CASE3$ access is performed next, but this access was not under attack. Since tag 111 is stored in the SCAAT memory, $found_in_SCAAT$ is set to '1' and location 101, where tag 111 is stored, is used as $INDEX_BITS$ of $SCAAT_out$.

Finally, $CASE4$ access is performed, but this access, unlike the access made at $CASE3$, was under attack. $INDEX_BITS$ of $SCAAT_out$ are changed to reflect the location of tag 111 in SCAAT memory. However, for this access, note that even though an attack was detected by the monitor, the en signal is not set to '1' because the tag was found in SCAAT memory. This prevents generating a new pseudo-random number if an attack repeats on a tag stored in the SCAAT memory.

C. SCAAT impact on cache behavior

Introducing the SCAAT unit to the cache system aims at mitigating LSCAs. This is done by eliminating predictable patterns of cache access. By changing where data is stored in the cache for a given cache access, changes the access and timing patterns of the cache behavior.

Fig. 3b, using the same coloring scheme and address length as in Fig. 3a, shows the timing waveform of three consecutive accesses within a period of 6 clock cycles $t(0)$ to $t(5)$ in a given access pattern. Signal P_{miss} indicates the predicted miss pattern, signal A_{miss} indicates the actual miss pattern, and signal cache[101] shows the tag of the block stored at cache location 101. Assuming no attacks had ever occurred and that the cache locations 010 and 101 are populated with tags 111 and 100 respectively, all three accesses are predicted to be cache hits. However, if an attack occurs on the second access, at $t(2)$, the SCAAT unit is activated. Tag 111 is now mapped to location 101. This, of course, will cause a miss, since location 101 is populated with tag 100. Furthermore, since, after the remapping of tag 111 to location 101 occurred, the third access to tag 100, at $t(4)$, will now also cause a miss since location 101 is populated with tag 111 as a result of the previous access.

The scenario represented in Fig. 3b demonstrates the change in access pattern to the cache by invoking cache misses for accesses that are predicted to be cache hits and vise versa changes the timing pattern of cache responses to access by the CPU. Using the SCAAT unit can reduce the predictability of the cache behavior. As a result, attackers can no longer successfully correlate the information collected during an attack to the actual behavior of the cache.

IV. Experimental Results

The experimental results evaluate the proposed SCAAT in mitigating LSCAs. For the baseline cache architecture, as mentioned in Section II, the open-source IP core cache

from [26] is selected. In comparison, the proposed SCAAT will be comprised of the baseline cache with a SCAAT unit fitted to the CPU-side interface of the baseline cache. Attack detection is done using the monitor from [28]. Since the target cache is parameterized, both the baseline cache and SCAAT are evaluated using two types of cache architectures: direct-mapped and 4-way associative caches.

TABLE II: Number of Attack Occurrences (AO), instances of SCAAT activation (SA), and the number of tags stored in SCAAT memory (ST) for each of the benchmarks.

	Direct-Mapped			4-way Associative		
	AO	SA	ST	AO	SA	ST
gcc	1556	28552	83	1544	28516	80
gzip	6	268	5	6	268	5
swim	1030	12611	33	1030	12610	33

A. Performance analysis

In order to evaluate and analyze the effect of the proposed SCAAT on the performance of the cache system, three applications from SPEC2000 CPU Benchmark Suite [29] were selected. For evaluating performance, 32-bit CPU data and address busses are connected to the baseline cache and the SCAAT. To achieve this, both a direct-mapped cache and a 4-way associative cache were configured to fit the required cpu_addr length.

Table II shows the number of attack occurrences (AO), which are occurrences of false positives generated by the monitor and are treated as attacks to be mitigated by SCAAT. The number of SCAAT activation (SA) instances includes both the mitigation of the attack occurrences as well as any accesses to the tags stored (ST) in the SCAAT memory for both cache types. All applications experience a similar number of attack occurrences for both cache types resulting in a similar number of SCAAT activation instances. This shows that the SCAAT was able to mitigate all attacks detected for both cache types.

The results for cache hit rates for the benchmarks are shown in Fig. 4.

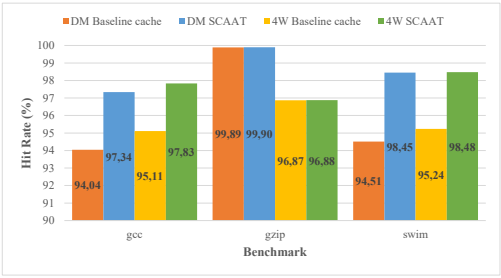


Fig. 4: Hit rate comparison of benchmarks for direct mapped (DM) and 4-way associative (4W) caches.

The results in Fig. 4 show that using the SCAAT does not have a notable effect on the overall performance of the cache. This means that using the SCAAT unit to mitigate attacks comes at no additional cost to the performance of the cache.

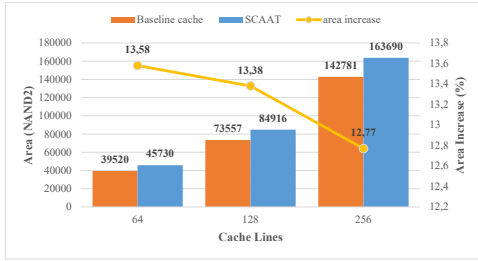
B. Area and critical path delay overheads analysis

In order to evaluate and analyze the area overhead and the critical path delay overhead of the proposed SCAAT in comparison to the baseline cache, several configurations of a 16-bit CPU_ADDR_BITS were synthesized using the Synopsys synthesis suite [30] and AMS 0.18μm CMOS technology library [31]. Table III shows the most notable configuration parameters used for area and critical path delay analysis. In addition to the parameters in Table III, all configurations have a 128-bit MEM_DATA_BITS, allowing up to 4 words per block.

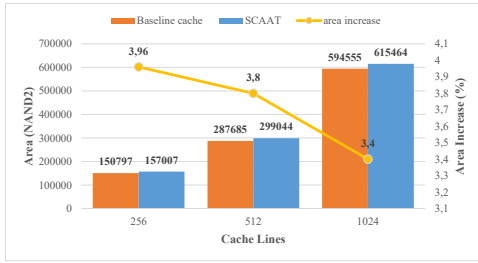
The results for area (2-input NAND (NAND2) gate area equivalent) are shown in Fig. 5. Fig. 5 (a) shows area estimation for direct-mapped baseline cache and direct-mapped SCAAT with 64, 128, and 256 cache lines each. Fig. 5 (b) shows area estimation for 4-way associative baseline cache and 4-way associative SCAAT with 256, 512, and 1024 cache lines each. Additionally, both Fig. 5 (a) and (b) include the percentage of area increase between the baseline cache and SCAAT at each respective number of cache lines. It is important to note that the area reported only includes the synthesized SCAAT unit and cache as shown in Fig 1.

TABLE III: Cache configuration for area estimation.

	Cache Type					
Parameter	Direct-Mapped			4-way associative		
Cache Lines	64	128	256	256	512	1024
Associativity	1	1	1	4	4	4
Cache (KB)	1	2	4	4	8	16
SCAAT memory (B)	64	112	192	64	112	192



(a) Area estimation for direct-mapped caches.



(b) Area estimation for 4-way associative caches.

Fig. 5: Area estimations (NAND2 equivalent) by cache type.

Fig. 5 (a) shows that an area increase between 12.77% and 13.58% occurs when the SCAAT is implemented using a direct-mapped cache architecture compared to a baseline cache of the same architecture and size. However, the percentage of area increase has been shown to decrease when the number of cache lines doubles. This can be largely attributed to the compact nature of the SCAAT memory as it only stores the TAG_BITS of cpu_addr.

Fig. 5 (b) shows that, compared to the direct-mapped caches, a much lower area increase of 3.40% to 3.96% occurs when the SCAAT is implemented using a 4-way associative cache architecture compared to a baseline cache of the same architecture and size. In addition to the SCAAT memory's compactness mentioned before, another factor for the reduction of area increase is the associativity of the cache that allows a much larger number of cache lines to be addressed using the same number of $INDEX_BITS$. Since the size of SCAAT memory is largely determined by the $INDEX_BITS$ parameter, a smaller SCAAT memory is needed relative to the size of the caches. This contributes to SCAAT memory taking a far less area percentage of the SCAAT system area. Implementing SCAAT with k-way associative cache architecture with high associativity can lead to a further decrease in area overhead.

Fig. 5 shows the advantage of using SCAAT in terms of area. While many of the current approaches require fixed architecture to increase the security of the cache at high area overhead, SCAAT can be constructed using any cache architecture with introducing low area overhead. In addition, the SCAAT does not require any change to the architecture of the cache itself.

As for the critical path delay, a clock period of $10ns$ is assumed for all the configurations. The evaluated path delay from cpu_addr input going through the SCAAT unit to the cache address input. On average, for each bit of cpu_addr, the delay is $4.83ns$ i.e. 0.48 of a clock cycle. This delay is caused by the sequential nature of the SCAAT memory i.e. half a clock cycle is needed to write a TAG_BITS of cpu_addr to the SCAAT memory. This delay, however, has no effect on the operation time of the SCAAT. As mentioned in Section III-A and demonstrated in Fig. 3a between $t(5)$ and $t(6)$, $LFSR_out$ is used as $INDEX_BITS$ of $SCAAT_out$ in the duration of time needed to store TAG_BITS in SCAAT memory. This eliminates any effect of delay caused by the sequential nature of the SCAAT memory.

V. Conclusions

In this work, a Secure Cache Alternative Address Table (SCAAT) is proposed. SCAAT, when connected to a monitoring system, has been shown to mitigate Logical Side-Channel Attacks (LSCA) by remapping address under attack to random locations. This approach shows several advantages over current approaches to cache security. On one hand, current approaches to cache security suffer from performance degradation and area overheads, however SCAAT was shown to have little effect on the cache performance as well as introducing low area overhead between 12.77% and 13.58% for direct-mapped and between 3.40% to 3.96% for k-way associative caches. On the other hand, while current approaches require special modifications or fixed cache architecture to increase security, SCAAT can be constructed using any direct-mapped or k-way associative cache architectures with no special modifications. This makes the SCAAT a feasible and scalable option for increasing the security of the cache system.

References

[1] F. Liu *et al.*, "Newcache: Secure cache architecture thwarting cache side-channel attacks," *IEEE Micro*, vol. 36, pp. 8–16, 2016.

[2] D. J. Bernstein, "Cache-timing attacks on aes," Tech. Rep., 2005.

[3] C. Percival, "Cache missing for fun and profit," in *Proc. of BSDCan 2005*, 2005.

[4] D. A. Osvik *et al.*, "Cache attacks and countermeasures: The case of AES," in *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2006, Proceedings*, 2006, pp. 1–20.

[5] D. Gruss *et al.*, "Cache template attacks: Automating attacks on inclusive last-level caches," in *Proceedings of the 24th USENIX Conference on Security Symposium*, ser. SEC'15. Berkeley, CA, USA: USENIX Association, 2015, pp. 897–912. [Online]. Available: http://dl.acm.org/citation.cfm?id=2831143.2831200

[6] Y. Yarom *et al.*, "Cachebleed: A timing attack on openssl constant time RSA," in *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, 2016, pp. 346–367. [Online]. Available: https://doi.org/10.1007/978-3-662-53140-2_17

[7] G. Doychev and B. Köpf, "Rigorous analysis of software countermeasures against cache attacks," *SIGPLAN Not.*, vol. 52, Jun. 2017.

[8] S. Deng *et al.*, "Cache timing side-channel vulnerability checking with computation tree logic," in *HASP '18*, 2018.

[9] S. Deng *et al.*, "Analysis of secure caches using a three-step model for timing-based attacks," *HaSS*, 2019.

[10] Z. He and R. B. Lee, "How secure is your cache against side-channel attacks?" in *MICRO*, 2017.

[11] M. Chiappetta *et al.*, "Real time detection of cache-based side-channel attacks using hardware performance counters," *ASC*, 2016.

[12] S. Briongos *et al.*, "Cacheshield: Protecting legacy processes against cache attacks," *CoRR*, 2017.

[13] M. Mushtaq *et al.*, "Nights-watch: A cache-based side-channel intrusion detector using hardware performance counters," in *HASP*, 2018.

[14] T. Zhang *et al.*, "Cloudradar: A real-time side-channel attack detection system in clouds," 2016.

[15] N. Zhang *et al.*, "Case: Cache-assisted secure execution on arm processors," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 72–90.

[16] M. Neagu *et al.*, "Interleaved scrambling technique: A novel low-power security layer for cache memories," in *2014 19th IEEE European Test Symposium (ETS)*. IEEE, 2014, pp. 1–2.

[17] R. B. Lee *et al.*, "Architecture for protecting critical secrets in microprocessors," in *ISCA'05*, June 2005.

[18] D. Zhang *et al.*, "Language-based control and mitigation of timing channels," *SIGPLAN Not.*, vol. 47, Jun. 2012.

[19] D. Zhang *et al.*, "A hardware design language for timing-sensitive information-flow security," *SIGPLAN Not.*, vol. 50, Mar. 2015.

[20] Y. Wang *et al.*, "Secdcp: Secure dynamic cache partitioning for efficient timing channel protection," in *53nd ACM/EDAC/IEEE DAC*, June 2016.

[21] L. Domnitser *et al.*, "Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks," *ACM TACO*, Jan. 2012.

[22] V. Kiriansky *et al.*, "Dawg: A defense against cache timing attacks in speculative execution processors," in *51st IEEE/ACM MICRO*, 2018.

[23] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," *SIGARCH Comp. Arch. News*, 2007.

[24] M. Yan *et al.*, "Secure hierarchy-aware cache replacement policy (sharp): Defending against cache-based side channel atacks," in *ISCA '17*, 2017.

[25] B. Niazmand *et al.*, "Design and verification of secure cache wrapper against access-driven side-channel attacks," in *2019 22nd Euromicro Conference on Digital System Design (DSD)*, 2019, pp. 672–676.

[26] Chair of VLSI Design, Diagnostics and Architecture. (2016) PoC - Pile of Cores. Technische Universität Dresden. [Online]. Available: https://github.com/VLSI-EDA/PoC

[27] T. B. Preußer *et al.*, "The portable open-source ip core and utility library poc," in *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2016, pp. 1–6.

[28] C. Reinbrecht *et al.*, "Lid-cat: A lightweight detector for cache attacks," in *25th IEEE European Test Symposium*, Tallinn, ESTONIA, 2020.

[29] S. Sair and M. Charney, "Memory behavior of the spec2000 benchmark suite," Technical report, Tech. Rep., 2000.

[30] Synopsys design compiler. [Online]. Available: https://www.synopsys.com/

[31] AMS 0.18um CMOS process - AMS Process Technology. [Online]. Available: https://ams.com/process-technology

Appendix 2 - CLD: An accurate, cost-effective and scalable run-time cache leakage detector

II

A. Shalabi, T. Ghasempouri, P. Ellervee, and J. Raik. Cld: An accurate, cost-effective and scalable run-time cache leakage detector. In *2021 24th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, pages 127–132. IEEE, 2021

CLD: An Accurate, Cost-Effective and Scalable Run-Time Cache Leakage Detector

Ameer Shalabi, Tara Ghasempouri, Peeter Ellervee, Jaan Raik
Department of Computer Systems, Tallinn University of Technology, Tallinn, Estonia
{ameer.shalabi, tara.ghasempouri, peeter.ellervee, jaan.raik}@taltech.ee

***Abstract*—Cache logical side channel attacks pose a significant threat to the security of modern computer systems. This is a result of exploitation of cache information leakages arising from cache contention. Detection of such leakages can be inferred from cache behavior and processes' access patterns during run time. To achieve this, a detection template that uses available information on cache outputs and process accesses at run-time is required. In this work, such template is proposed and implemented as a hardware monitor called Cache Leakage Detector (CLD). CLD is a high-accuracy, cost-effective and scalable run-time cache information leakage detector. CLD uses cache signals and process IDs to detect exploitable cache access patterns. It does so by identifying potential information leakage patterns. Accuracy of CLD is evaluated by using several benchmarks and injecting attacks into a 128-bit key AES algorithm. The experiments demonstrate that CLD has far higher detection accuracy (0.7964 vs 0.3195) and lower percentage of false positive detections (1.2% vs 30.6%) compared to a state-of-the-art hardware detector. Moreover, CLD introduces a very low area overhead of 0.002% to the total area of the cache. Experimental result section reports the above claims in detail.**

I. Introduction

The design of secure digital systems is becoming increasingly difficult. In the past, a wide variety of attacks against ICs and memories have been reported, many of them exploiting weaknesses in modern cache architectures [1]. More specifically, the design targets of modern caches and cache systems arise from their characteristics in resource sharing. As a consequence, when several processes are executed at the same time, the usage of the cache by one process can be inferred by another process through cache contention [1]. Since modern processors execute several processes at a time, it is possible to leak sensitive information regarding critical processes through Logical Side-Channel Attacks (LSCA) [2].

In recent years, a wide range of cache attacks [2–5] have been presented. The greater challenge in detecting cache attacks is characterizing the behavior of the system that leads to information leakage. To this end, several approaches for evaluating and quantifying cache's vulnerability to information leakage have been presented [6–9]. Some of those approaches, for example, used information-flow analysis and static cache analysis [7] for quantifying cache side-channels. In the case of LSCAs, the system behavior is not compromised since these attacks rely on side-channels to obtain leakage information. In fact, it is not unusual that an attack is only detected after the attacker has successfully extracted sensitive information. Therefore, a reliable and accurate run-time detection of information leakage is crucial for the security of modern digital systems.

One way to detect such attacks is by monitoring the cache behavior. A few publications have reported offline and online cache monitoring for security purposes. On one hand, offline monitoring [10–12] comes with little overhead and no hardware needed for implementation. Detection of attacks using such approaches relies heavily on the coverage of valid attack scenarios and are only effective after attacks occurred. Online monitoring approaches [13–18], on the other hand, yield higher attack detection than offline monitoring as they collect information for internal hardware monitors, such as CPU Hardware Performance Counters (HPCs) [13–15], to detect different types of LSCAs. Such approaches often only target a specific attack and its variations and require additional offline analysis to detect attacks, introducing an overhead in cost and performance. Furthermore, a run-time attack detection device, LiD-CAT [19], used models to characterize the information leakage by exploiting common cache attacks' characteristics. However, such approach was shown to yield a high percentage of false positives.

978-1-6654-3595-6/21/$31.00 ©2021 IEEE

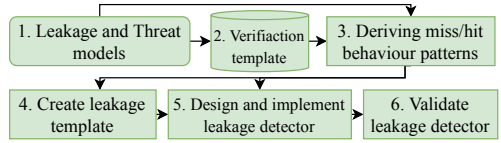


Fig. 1: CLD devolopment steps

In summary, all the previous works either operate offline, assume knowledge of the attacker to be known or target specific concrete attacks. To overcome the above drawbacks, in this work, a process-based classification is proposed to detect cache information leakage patterns regardless if the attacker is known, unknown, or present. It performs detection of possible information leakages online by checking each cache access, allowing the detection of all types of possible leakages at run time. Additionally, the proposed approach can be fitted into the cache architecture itself with little overhead.

As shown in Fig. 1, the proposed methodology is divided into six main steps. Step 1 defines the leakage and threat models related to the framework of this work. Step 2 creates a verification template from the leakage and threat models. Step 3 uses the information from step 1 and 2 to derive hit/miss behavior patterns. In step 4, a template for cache information leakage is created. In step 5, a leakage detector is designed and implemented. Finally, in step 6 the effectiveness of the detector is validated.

The contribution of this paper is proposing a novel technique for detecting cache information leakage patterns that can be exploited by LSCAs that exceeds the state of the art by proposing:

- An accurate leakage detection model providing high pos-

itive correlation between access patterns and information leakage detection with a low ratio of false positives.
- A run-time leakage detection mechanism that operates independently from any assumptions regarding the attackers' behavior.
- A cost-effective detector architecture based on readily available cache output signals, CLD introduces negligible area overhead to total area of the cache.

To the best of the authors' knowledge, this is the first run-time detector that focuses on cache leakages as opposed to specific attack types.

The rest of the paper is organized as follows: Section II outlines the background of the relevant concepts and related work on cache attacks and attack modelling, Section III presents the proposed methodology, Section IV discusses the results of experimentation, and finally, Section V concludes the paper.

II. Background

In this section, an overview of the relevant cache attacks and cache attack modeling are presented.

A. Cache Attacks

Depending on the side-channel information leakage being exploited by an attacker, cache attacks are commonly categorized into three types: time-driven, trace-driven, and access-driven. In a time-driven attack, the source of information leakage is the execution time of a process and its cache accesses' timing, which can be exploited to leak sensitive information. Such attacks are the Evict+Time [20], collision timing [21], and Bernstein's [3] attacks.

In a trace-driven attack, the attacker records cache traces of a target process. Using these traces, sensitive information is deduced and collected. This can be done by generating traces of static and dynamic power [22] or system monitors [23] while a target process is being executed.

In an access-driven attack, an attacker process is used to observe and collect information regarding sensitive cache addresses accessed by the target process. By doing so, the attacker can deduce the access pattern of the target process. This can lead to secret information to be collected by the attacker. Examples of such attack include the Prime+Probe [20], Flush+Reload [24], and Flush+Flush [25] attacks.

B. Modeling Cache Attacks

Information leakages, using the previously mentioned cache attacks, can be modelled using several approaches. The most straight-forward approach is to simulate attacks on a given system to evaluate the information leakage vulnerabilities of the system [26]. However, simulation comes at considerable disadvantages. Extensive knowledge of the attack process and protection against such attacks is needed. Due to the complexity of simulating hardware and software together, this technique can prove very expensive and time consuming.

Due to the highlighted drawbacks of simulation in security verification, works apply formal verification techniques to model different cache attacks and verify cache attacks' occurrences in a target system. Work presented in [10] described an exhaustive set of 28 attack formulas in a three-step model. The formulas were further analyzed in [11]. These attack formulas are shown in Table I and are described using sequences of cache accesses, identification of an attacker or a victim that preforms these accesses, and knowledge that the attacker possesses regarding the victim. In Table I, column ID presents the id of each formula. Column $attack formula$, demonstrates attacks in three steps performed by actors. These actors perform actions defined as $V1$: victim accesses a location known to the attacker; Vx: victim accesses a location unknown to the attacker; Vr: victim flushes a location unknown to the attacker; $A1$: an access by the attacker to a cache location; Ar: a flush by the attacker of a cache location.

However, these models offer no description of the resulting behavior of the cache when such attacks are preformed, and whether such behavior is indicative of information leakage. Additionally, the model makes several assumptions that can be proven difficult to ascertain. More specifically, the assumptions that **i)** the attacker's behavior is known and/or detectable and **ii)** what knowledge the attacker possesses regarding the access behavior of the victim.

TABLE I: Attack formulas [10] analyzed in this work

ID	attack formula	$*$	ID	attack formula	$*$
1	$Vx \rightarrow Ar \rightarrow Vx$	-	15	$Vx \rightarrow Vx \rightarrow Ar$	d
2	$Vx \rightarrow Vr \rightarrow Vx$	-	16	$Ar \rightarrow Vx \rightarrow Vr$	d
3	$Ar \rightarrow A1 \rightarrow Vx$	-	17	$Vr \rightarrow Vx \rightarrow Vr$	d
4	$Vr \rightarrow A1 \rightarrow Vx$	-	18	$Vx \rightarrow Vx \rightarrow Vr$	d
5	$A1 \rightarrow A1 \rightarrow Vx$	-	19	$Ar \rightarrow Vx \rightarrow A1$	e
6	$V1 \rightarrow A1 \rightarrow Vx$	-	20	$Vr \rightarrow Vx \rightarrow A1$	e
7	$Vx \rightarrow A1 \rightarrow Vx$	a	21	$A1 \rightarrow Vx \rightarrow A1$	f
8	$Vx \rightarrow A1 \rightarrow Vx$	b	22	$V1 \rightarrow Vx \rightarrow A1$	-
9	$Vr \rightarrow V1 \rightarrow Vx$	b	23	$Vx \rightarrow Vx \rightarrow A1$	e
10	$A1 \rightarrow V1 \rightarrow Vx$	b	24	$Ar \rightarrow Vx \rightarrow V1$	b
11	$V1 \rightarrow V1 \rightarrow Vx$	b	25	$Vr \rightarrow Vx \rightarrow V1$	b
12	$Vx \rightarrow V1 \rightarrow Vx$	c	26	$A1 \rightarrow Vx \rightarrow V1$	-
13	$Ar \rightarrow Vx \rightarrow Ar$	d	27	$V1 \rightarrow Vx \rightarrow V1$	c
14	$Vr \rightarrow Vx \rightarrow Ar$	d	28	$Vx \rightarrow Vx \rightarrow V1$	b

$*$ Known attacks: **a**) Evict+Time [2]; **b**) Cache Collision [21]; **c**) Berstein [3]; **d**) Flush+Flush [25]; **e**) Flush+Reload [24]; **f**) Prime+Probe [2].

III. Proposed Methodology

In this section, the methodology used to develop the Cache Leakage Detector is presented. The leakage and threat models are presented and the verification template is explained. The method of derivation of the possible leakage patterns is presented and the process of creating the leakage template is shown. Finally, the design and implementation of the proposed CLD is discussed.

A. Threat and leakage models

To develop an accurate leakage detection mechanism, aspects of the system, the threat model, and the leakage model must be defined in an appropriate manner.

1) ***Threat Model****:* The threat model sets forth the vulnerabilities of a given system. This model will describe what can be exploited and used to leak information from the system. For this work, the target system is a classical Von Neumann architecture, multiple processors with private caches connected to a shared last level cache (LLC). An Operating System is responsible for resource management and process scheduling. In such system, several processes, each with a unique process IDs (*ProcID*), are being executed concurrently. All processes can access any location of the cache during the process execution. This causes several processes to share

the LLC, thereby causing mutual interference. In this work, a *Secure Process* (P_S) contains sensitive information and needs to be protected against possible leakage. Any process currently sharing the cache with P_S is assumed to be an *Interference Process* (P_I). P_I is a process that exhibits a cache access pattern that can lead to information leakage in the execution of P_S as a result of mutual cache interference. The emerging patterns from such mutual interference can be exploited by potential adversaries.

2) ***Leakage Model****:* In this work, to help detecting leakages, a set of leakage scenarios were devised. A leakage scenario is a sequence of three stages: S_1 (setup stage) where the first access initiates a sequence of accesses that can lead to a leakage, S_2 (trigger stage) where a second access causes a mutual interference that leads to a leakage and S_3 (observation stage) is an access where the leakage information can be observed. Each stage of the sequence describes a cache access to the same cache location. These accesses can be performed by either P_S or P_I. The sequence of accesses results in two possible leakage sequences: The first sequence happens when three accesses to same cache location are preformed by P_S with an access pattern that can be exploited to leakage information by observing the resulting cache behaviour. Such leakages can be exploited using the Bernstein's attack [3] and several variations of Cache Collision [21] attack. The second sequence happens when P_I processes interfere with process P_S's execution. Such interference leads to an access pattern by a P_I process that causes cache contention with P_S. Such leakages can be exploited using Evict+Time [2], Cache Collision [21], Flush+Flush [25], Flush+Reload [24], and Prime+Probe [2] attacks and their variations.

While previous works, such as in [10], require additional information about the attacker's knowledge and intention, the model proposed in this work removes any assumption regarding an "attacker" since information leakage from P_S can occur whether an attacker is present or not. Leakage resulting from P_I maliciously interfering with P_S's execution either requires the observation of the behavior of a single process or the resulting cache contention between a targeted process and other processes. Therefore, exploitable information leakage patterns can occur with and without the presence of malicious activity.

B. Verification template

In this step, leakage scenarios are defined based on the leakage model presented in the previous section. Due to their similarities, verification of the leakage scenarios are adopted from the idea presented in [10]. Nonetheless, [10] assumes that an attacker is present, and defines the actions that an actor can perform depending on the knowledge that attacker has. However, the leakage model presented in this work, does not make any assumption about the attacker's knowledge or intent. To overcome this, the CLD verification template will use two pieces of information that are readily available. The first is that the P_S process is known. Since P_S is the process being protected and checked for information leakage, it is considered the possible victim – it replaces actions $V1$, Vx and Vr. The second is that any processes other than P_S can cause mutual interference with it, therefore it is considered P_I process and potentially can interfere with process P_S access patterns and it replaces actions $A1$ and Ar.

C. Deriving miss/hit behaviour patterns

In addition to the issue of attacker's knowledge and intention addressed in the previous section, a significant drawback of the model presented in [10] is that it does not describe the behavior of the cache once accesses are preformed i.e. there is no classification of access patterns based on the hit/miss state of the cache.

To address this weakness, in this work, an analysis methodology, presented in [27], is adapted to analyze the resulting hit/miss behaviour of the cache when each of the leakage scenarios occurs. As shown in Table II, each leakage scenario requires three cache accesses, S_1, S_2 and S_3, and each access results in either a cache hit (H) or miss (M). The analysis aims at concluding which hit/miss combination can reveal a possible pattern of accesses that results in information leaking from the cache. As shown in Table II each leakage scenario exhibits 8 possible hit/miss combinations which can be accesses by S and I equivalent to an access by P_S and P_I respectively. For instance, SSI is equivalent to a sequence of two access by P_S and one access by P_I.

TABLE II: Hit/miss combination of all leakage scenarios.

S_1	S_2	S_3	Leakage Scenarios							
			SSS	SSI	SIS	SII	ISS	ISI	IIS	III
M	M	M								
M	M	H								
M	H	M								
M	H	H								
H	M	M								
H	M	H								
H	H	M								
H	H	H								

*"S" and "I" are equivalent to an access by P_S and P_I respectively. SSI is equivalent to a sequence of two accesses by P_S then an access by P_I.
Red square indicates $Unsafe$ group; green square indicates $Safe$ group.

All the possible hit/miss combinations of each leakage scenario are tested. Each scenario can be classified into one of the following two groups:

Definition 1. *Group $Safe$; A scenario with the resulting hit/miss combination does not lead to an exploitable leakage pattern.*

Definition 2. *Group $Unsafe$; A scenario with the resulting hit/miss combination leads to an exploitable leakage pattern.*

Table II shows which hit/miss combination of each leakage scenario belongs to the $Unsafe$ group (red square) and which hit/miss combination belongs to the $Safe$ group (green square).

D. Creating cache leakage template

By combining the results of the hit/miss combination of each leakage scenario from Table II and the verification template, a leakage template that only requires immediately available information such as the *ProcID* and the hit/miss output of the cache can be created. For example, let $event_x$ be an event of three accesses to the same cache location L by the same process $ProcID_x$ result in three consecutive misses. If $ProcID_x$ is a secure process, then the leakage scenario in question is SSS. In this case, $event_x$ indicates that a possible cache information leakage scenario at location L that

TABLE III: The transition table of the CLD FSM.

FSM transition $prev_state \rightarrow next_state$	Transition condition (FSM input)	safety_check (FSM output)
$st_0 \rightarrow st_1$	(P_S | P_I) & (M | H)	0 (safe)
$st_1 \rightarrow st_2$	P_S & (M | H)	0 (safe)
$st_1 \rightarrow st_3$	P_I & (M | H)	0 (safe)
$st_2 \rightarrow st_1$	P_I & M	0 (safe)
$st_2 \rightarrow st_0$	(P_S & (M | H)) | (P_I & H)	1 (*unsafe*)
$st_3 \rightarrow st_1$	(P_S & H) | (P_I & (M | H))	0 (safe)
$st_3 \rightarrow st_0$	P_S & M	1 (*unsafe*)

can be exploited by Berstein's or cache collision attacks (e.g attack formulas 11 and 12 in Table I). The leakage template further minimizes the required information (cache hit/miss and process ID) for leakage detection while encompassing all the possible scenarios that can lead to exploitable information leakage. Using this model, a run-time, compact, and scalable cache leakage detector can be designed.

E. CLD design and architecture

Since the *ProcID* and the hit (H) and miss (M) signals are readily available at access time during run time, those inputs can be used to construct a Finite State Machine (FSM) that checks for possible access patterns that can lead to information leakage. The leakage template described in the previous section is used to construct the transitions of the FSM. Table III shows the transition table of the FSM based on the *ProcID* and the hit (H) or miss (M) inputs. Each transition changes the state of the FSM from a $prev_state$ to a $next_state$, producing a $safety_check$ signal that indicates if an access pattern belongs to the $Unsafe$ (Definition 2) or a $Safe$ (Definition 1) scenarios. Fig. 2 shows the CLD's architecture with the corresponding FSM, register file, and cache. $Unsafe$ scenarios are shown as a red line.

Once an access is preformed, the hit (H) and miss (M) signals are received from the cache output. The *ProcID* is used to determine whether the process being accessed is P_S or P_I. The output of the register file ($prev_state[addr_index]$), is a 2-bit encoding of the previously stored CLD state for set $addr_index$. This state is used to determine the last state of the FSM. Once all these inputs are received by the FSM, a next state ($next_state[addr_index]$) is generated and stored in the register file.

All the values in the register file are initialized to $Start$ (st_0) state. Since The first access in the leakage sequence corresponds to setup stage (S_1), any access will transition the FSM from st_0 to the trigger stage (S_2) represented by state st_1. The outcome of the trigger stage is dependent on the type of access and the output of the cache and will transition to the observation stage (S_3). S_3 can be one of two possibilities: either that the current observation stage is the third access of an $Unsafe$ scenario or it is the start of a another scenario. If the observation stage is in fact third access of an $Unsafe$ scenario, then the FSM will always transition from st_2 or st_3 to st_0 and generate $safety_check$ signal to indicate the $Unsafe$ scenario. If the observation stage is in fact third access of a $safe$ scenario, then the FSM will always transition from st_2 or st_3 to st_1 since the S_3 can be the first access in another scenario. This design ensures that every access to the cache is checked for potential access patterns that can lead to an $Unsafe$ scenario.

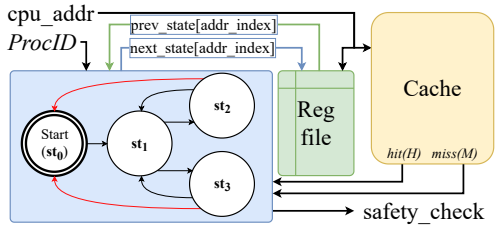


Fig. 2: The state transition diagram of the CLD.

IV. Experimental Results

In this section, the experiments conducted to evaluate the Cache Leakage Detector are explained. The experimental setup, accuracy evaluation of the CLD, and area overhead results are discussed.

A. Experiment setup

For the purpose of validating the CLD, the open-source IP core cache [28] was used. CLD's performance and accuracy were evaluated and compared to LiD-CAT [19]. This is because LiD-CAT has the most similar functionality as CLD (online detector) as well as their mutual design target (hardware, run-time detection).

Using the multi2sim simulator [29], an 8-core system with multi-level caches is simulated to execute 30 different benchmark application from SPEC CPU 2006 [30], SPLASH3 [31], and PARSEC [32] benchmark suites as well as a 128-bit key Advanced Encryption System (AES) algorithm. For every simulation run, an 8-benchmark combination is created where one unique benchmark is assigned to each core. Each simulation run is performed until the total number of accesses to a shared 4 MB, 16-way associative Last Level Cache (LLC) reach one million accesses. At the end of each run, the complete LLC access traces for each of the benchmarks are recorded and used as input for the experiments.

Two different experiments were conducted to determine the accuracy of leakage and attack detection of both CLD and LiD-CAT. In the first experiment, CLD and LiD-CAT were used to detect cache information leakage in the LLC as a result of cache contention during run-time. The number of accesses by each of the benchmarks to each cache set, starting from when all the ways in the cache set had been used at least once, are recorded. Information leakage detected at each cache set by CLD and LiD-CAT are also recorded for each benchmark. The second experiment consisted of injecting attacks into a 128-bit key Advanced Encryption Standard (AES) algorithm. Using the cache traces from multi2sim, the locations most accessed by AES are labeled. A total of 250 attacks were injected into those locations at random.

B. Leakage detection

In order for cache contention between a secure process and other processes to occur, the number of accesses by a secure process (acc_x) must be at least 1. Moreover, an increase in acc_x to a cache location S_x, would increase the possible cache contention events at location S_x by increasing the number of possible exploitable access patterns. Since the number of cache contention events are dependent on acc_x, the number

of cache information leakage occurrences as a result of cache contention events can not be greater than acc_x. Therefore, the accuracy of a cache information leakage detector can be evaluated by quantifying the direct relationship between acc_x and the number of detections made by the detector. To evaluate the leakage detection accuracy of CLD and LiD-CAT, the Pearson's product moment correlation coefficient between acc_x and the number of leakages detected by CLD is calculated. Similarly, the correlation coefficient between acc_x and the number of leakages detected by LiD-CAT at cache location S_x is calculated as well. The correlation coefficient will underscore any underlying linear relationship between acc_x and either of CLD or LiD-CAT. A high positive linear correlation indicates a high degree of association between detection of leakage scenarios and cache access patterns for a given benchmark, indicating high detection accuracy.

Fig. 3 shows the correlation between acc_x and CLD as well as acc_x and LiD-CAT. The figure demonstrates that CLD have higher correlation with acc_x for almost all the benchmarks. This shows a direct relationship between the number of accesses by a secure process and the detected leakages by CLD, indicating higher accuracy in leakage detection. All in all, detection made by CLD have an average of 0.7964 positive correlation to the number of accesses performed by a secure process, more than double that of LiD-CAT at 0.3195 positive correlation.

Fig. 4 shows the number of accesses acc_x compared to CLD and LiD-CAT for a sample of 200 cache locations (where $2000 \leq x \leq 2200$) for the *astar* benchmark [30]. CLD had a higher positive correlation coefficient of 0.7952 to the number of accesses by *astar* compared to 0.2134 for LiD-CAT. This is largely because the number of detections made by LiD-CAT at some locations in the cache are in fact higher than the number of accesses by *astar*. This indicates that a high number of false positives occur at those locations. In the sample shown, the clear spikes in LiD-CAT detections were, after analysis, directly tied to accesses performed by the *bzip2* benchmark [30]. In fact, there is 0.9361 positive correlation between the number of accesses by *bzip2* and the spikes in LiD-CAT detections (when *astar* is the secure process).

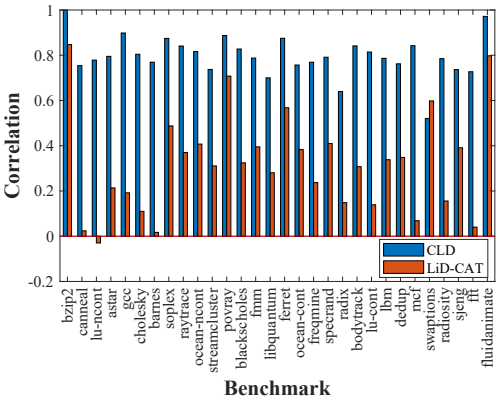


Fig. 3: Access pattern correlation with CLD and LiD-CAT detections.

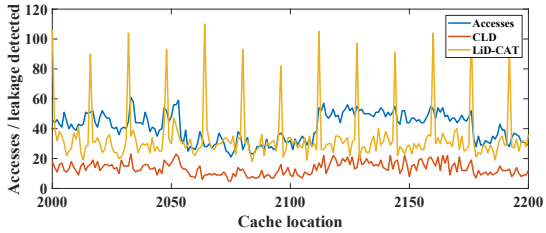


Fig. 4: Number of accesses, CLD, and LiD-CAT detections at 200 cache locations sampled from *astar* benchmark.

This finding highlights that, unlike CLD, LiD-CAT is prone to false positives generated from cache contention between benchmarks other than the secure process.

As for the second experiment, 10 attacks were injected at each of the 25 most accessed locations by AES. The results were obtained by comparing the detections made by CLD and LiD-CAT before and after attacks were injected. Table IV shows the attack types used in this experiment and the number of attack detections (D) made by both detectors and the percentage of False Positive detections (FP) made by each detector. For all the attacks used in this experiment, the number of detections made by CLD increased by almost the same number of injected attacks, with an average of only 1.2% probability of false positive. On the other hand, LiD-CAT showed an average of 30.6% probability of false positive. LiD-CAT overcame this issue with a detection threshold. However, since CLD does not require a detection threshold, in this work, no detection threshold was used for LiD-CAT in order to achieve better comparison between the two detectors. Additionally, the very large sample of access traces used in this work is responsible for the the notable increase of probability of false positive for LiD-CAT as opposed to what was reported in [19].

TABLE IV: Detections of attacks injected into AES algorithm.

		Bernstein	Flush+ Reload	Evict+ Time	Cache Collision	Flush+ Flush	Attack ID 3	Avg.
CLD	D	252	255	251	254	257	250	**253.1**
	FP	0.8%	2%	0.4%	1.6%	2.8%	0%	**1.27%**
LiD-CAT	D	278	405	354	372	300	250	**326.5**
	FP	11.2%	62%	41.6%	48.8%	20%	0%	**30.60%**

C. Area overhead

In order to evaluate and analyze the area overhead of the proposed CLD, several configurations of common shared Last Level Cache sizes were synthesized with CLD. The area estimation were made using the Cadence Genus tool [33] and a $40nm$ CMOS technology commercial library. The OpenRAM memory compiler [34] is used to estimate the memory area of the cache. Due to the very large SRAM area of the LLC caches used, the cell area overhead of CLD is compared to the cell area of the cache excluding the SRAM memory area and its associated controllers. All synthesized caches are configured as 16-way associative with 64 bytes per cache line.

Table V shows the sequential (**SEQU**), combinational (**COMB**), and total (**TOTAL**) area overhead (**AR**) and cell count overhead (**CL**) introduced by CLD to the control portion of the cache. The CLD SEQU area overhead slightly increases as the number of CLD SEQU cells doubles when the cache

TABLE V: CLD's area overhead for all cache configurations.

cache		CLD Control overhead						CLD
size	# sets	SEQU		COMB		TOTAL		total
		AR	CL	AR	CL	AR	CL	overhead
2 MB	2048	6.36%	4160	16.51%	11578	8.70%	15891	0.0019%
4 MB	4096	6.61%	8256	15.46%	18836	8.33%	27176	0.0018%
8 MB	8192	7.67%	16448	13.53%	51906	9.35%	68701	0.0020%

* Including buffers and inverter cells

size doubles. As the cache size doubles, double the number of CLD SEQU cells are required to store the CLD states for each cache set. However, since the control logic (FSM) of the CLD does not change, the CLD COMB area overhead slightly decreases as cache size doubles. Yet, an increase in the number of COMB cells can be seen. This increase is because additional logic was added for read/write operation from/to the register file. As the size of the register file increases, so does its associated read and write control logic. All in all, on average, CLD adds 8.79% to the total control portion of the cache. Compared to the overall area of the cache, including both control logic and SRAM memory area of the cache, CLD introduces a negligible 0.002% increase to the total cache area.

D. Design limitations

Inputs to CLD are highly dependent on the performance of the cache. Since CLD uses the hit and miss signals generated by the cache as an input, it inherits the delay of these signals. In both experiments, both hit and miss signals were generated combinationaly [28], introducing negligible delay to CLD hardware implementation. Nonetheless, when adapting the CLD for other types of caches, the respective timing constraints must be taken into consideration.

V. Conclusions

In this work, CLD, a high accuracy and low overhead run-time leakage detector is presented with a mechanism that operates independently from any assumptions regarding the attackers' behavior. By taking advantage of readily available cache outputs and process IDs, CLD is able to clearly outperform a state-of-the-art run-time attack detector, both in detecting cache leakages and cache attacks. The experiments in this work demonstrate that leakage detections made by CLD have, on average, high positive correlation of 0.7952 to the access patterns of the process being protected at a very low probability of false positive (of 1.2%). The results from the experiments show that CLD has much higher accuracy of leakage and attack detections than current state of the art run-time attack detector. Moreover, a cost-effective detector architecture is proposed and implemented in $40nm$ technology introducing as low as 0.002% area overhead to total cell area of the cache. All in all, this work demonstrates that CLD is capable of detecting both cache information leakages as well as cache attacks at run-time and at high accuracy with negligible area overhead.

References

[1] F. Liu *et al.*, "Newcache: Secure cache architecture thwarting cache side-channel attacks," *IEEE Micro*, vol. 36, pp. 8–16, 2016.

[2] D. A. Osvik *et al.*, "Cache attacks and countermeasures: The case of AES," in *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2006, Proceedings*, 2006, pp. 1–20.

[3] D. J. Bernstein, "Cache-timing attacks on aes," Tech. Rep., 2005.

[4] D. Gruss *et al.*, "Cache template attacks: Automating attacks on inclusive last-level caches," in *Proceedings of the 24th USENIX Conference on Security Symposium*, ser. SEC'15, 2015, pp. 897–912.

[5] Y. Yarom *et al.*, "Cachebleed: A timing attack on openssl constant time RSA," in *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, 2016, pp. 346–367.

[6] S. Chattopadhyay *et al.*, "Quantifying the information leakage in cache attacks via symbolic execution," vol. 18, 2019.

[7] B. Köpf *et al.*, "Automatic quantification of cache side-channels," in *Computer Aided Verification*, P. Madhusudan and S. A. Seshia, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 564–580.

[8] T. Zhang and R. B. Lee, "New models of cache architectures characterizing information leakage from cache side channels." New York, NY, USA: Association for Computing Machinery, 2014.

[9] B. B. Yilmaz *et al.*, "Quantifying information leakage in a processor caused by the execution of instructions," in *MILCOM 2017-2017 IEEE Military Communications Conference (MILCOM)*. IEEE, 2017, pp. 255–260.

[10] S. Deng *et al.*, "Cache timing side-channel vulnerability checking with computation tree logic," in *7th Int. WS on Hardware & Architectural Support for Security & Privacy*, ser. HASP '18, 2018, pp. 2:1–2:8.

[11] S. Deng *et al.*, "Analysis of secure caches using a three-step model for timing-based attacks," *Journal of Hardware and Systems Security*, pp. 1–29, 2019.

[12] Z. He and R. B. Lee, "How secure is your cache against side-channel attacks?" in *MICRO*, 2017.

[13] M. Chiappetta *et al.*, "Real time detection of cache-based side-channel attacks using hardware performance counters," vol. 49, 2016, pp. 1162 – 1174.

[14] M. Alam *et al.*, "Performance counters to rescue: A machine learning based safeguard against micro-architectural side-channel-attacks," vol. 2017, 2017, p. 564.

[15] M.-M. Bazm *et al.*, "Cache-based side-channel attacks detection through intel cache monitoring technology and hardware performance counters," 04 2018, pp. 7–12.

[16] S. Briongos *et al.*, "Cacheshield: Protecting legacy processes against cache attacks," *CoRR*, 2017.

[17] M. Mushtaq *et al.*, "Nights-watch: A cache-based side-channel intrusion detector using hardware performance counters," in *HASP*, 2018.

[18] T. Zhang *et al.*, "Cloudradar: A real-time side-channel attack detection system in clouds," 2016.

[19] C. Reinbrecht *et al.*, "Lid-cat: A lightweight detector for cache attacks," in *25th IEEE European Test Symposium*, Tallinn, ESTONIA, 2020.

[20] E. Tromer *et al.*, "Efficient cache attacks on aes, and countermeasures," *Journal of Cryptology*, 2010.

[21] A. Bogdanov *et al.*, "Differential cache-collision timing attacks on aes with applications to embedded cpus," in *CT-RSA 2010*.

[22] J.-F. Gallais *et al.*, "Improved trace-driven cache-collision attacks against embedded aes implementations," in *WISA*, 2011.

[23] O. Acıiçmez and Çetin K. Koç, "Trace-driven cache attacks on aes," in *ICICS*, 2006.

[24] Y. Yarom and K. Falkner, "Flush+reload: A high resolution, low noise, l3 cache side-channel attack," in *23rd USENIX*, 2014.

[25] D. Gruss *et al.*, "Flush+flush: A fast and stealthy cache attack," in *DIMVA*, 2016.

[26] S. Chattopadhyay and A. Roychoudhury, "Symbolic verification of cache side-channel freedom," *TCAD*, 2018.

[27] C. Reinbrecht *et al.*, "A security verification template to assess cache architecture vulnerabilities," in *2020 23rd International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, 2020, pp. 1–6.

[28] Chair of VLSI Design, Diagnostics and Architecture. (2016) PoC - Pile of Cores. Technische Universität Dresden. [Online]. Available: https://github.com/VLSI-EDA/PoC

[29] R. Ubal *et al.*, "Multi2sim: A simulation framework for cpu-gpu computing." New York, NY, USA: Association for Computing Machinery, 2012.

[30] J. L. Henning, "Spec cpu2006 benchmark descriptions," vol. 34, 2006.

[31] C. Sakalis *et al.*, "Splash-3: A properly synchronized benchmark suite for contemporary research," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2016, pp. 101–111.

[32] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[33] Genus Synthesis Solution. [Online]. Available: https://www.cadence.com/

[34] M. R. Guthaus *et al.*, "Openram: An open-source memory compiler," in *Proceedings of the 35th International Conference on Computer-Aided Design*, ser. ICCAD '16. New York, NY, USA: Association for Computing Machinery, 2016.

Appendix 3

III

A. Shalabi, T. Ghasempouri, P. Ellervee, and J. Raik. Rscaat: A recursive mechanism to mitigate cache side channel attacks. *IEEE Access*, 14:52342–52359, 2026

IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 2 March 2026, accepted 26 March 2026, date of publication 2 April 2026, date of current version 8 April 2026.

Digital Object Identifier 10.1109/ACCESS.2026.3680054

RESEARCH ARTICLE

RSCAAT: A Recursive Mechanism to Mitigate Cache Side Channel Attacks

AMEER SHALABI, TARA GHASEMPOURI (Member, IEEE), PEETER ELLERVEE, (Member, IEEE), AND JAAN RAIK, (Member, IEEE)
Department of Computer Systems, Tallinn University of Technology, 19086 Tallinn, Estonia

Corresponding author: Ameer Shalabi (ameer.shalabi@taltech.ee)

This work was supported in part by Estonian Research Council under Grant PSG837.

ABSTRACT Multi-level caching was introduced to increase computing performance by alleviating the speed difference between the processors and memory. However, this has introduced vulnerabilities that can be exploited by logical side-channel attacks. While there exist solutions that perform run-time remapping of data under threat to randomized locations, such solutions inhibit vulnerabilities in case of repetitive attacks. This paper goes beyond the state of the art by proposing a run-time mitigation mechanism called Recursive Secure Cache Alternative Address Table (RSCAAT) designed with a robust control logic and enhanced reactivity to cache behavior. It includes a specialized Content-Addressable Memory (CAM) with built-in re-mapping redundancies capable of eliminating predictable pseudorandom traces resulting in area overheads in the range of 4-5%. RSCAAT is evaluated using benchmarks from several suites. Results show that while the overhead of RSCAAT in performance is comparable to the state of the art, it provides significantly higher levels of security to the cache by accurately eliminating leakage-like patterns as well as recursively remapping vulnerable accesses. RSCAAT is activated for 100% of the leakage-like patterns detected while state-of-the-art solutions have been shown to only activate 53-93%.

INDEX TERMS Hardware security, cache logical side-channel attacks, attack mitigation mechanisms, run-time remapping.

I. INTRODUCTION

The introduction of multi-level caching to processor systems is aimed at mitigating the ever-increasing speed gap between the high-frequency processing cores and the slow, high-latency Main Memory. Multiple layers of caches, increasing in size and latency, are placed between the core (smallest and fastest cache) and Main Memory (largest and slowest cache) to ensure that data is accessed at the lowest possible overall latency. By using several levels of caches, data is delivered to the core at low latency in the event of a cache hit (data found in the cache). Nonetheless, since the size of caches decreases at lower levels, data must be retrieved from the higher cache levels in the event of a cache miss (data not found in the cache) increasing the access latency.

Due to high latency caused by cache misses, the timing difference between cache hits and cache misses is significant. This timing difference remains an inherent issue in systems employing such multi-level caching schemes [1]. For multi-core processors, this timing difference problem is further compounded as it can be used as a cache side-channel by an adversary. By carefully measuring and probing this difference in timing, adversaries can deduce the behavior of the shared Last-Level Cache (LLC) and infer sensitive information and system secrets, such as in the case of Bernstein's timing attack on the Advanced Encryption Standard (AES) [2]. Regardless, cache side-channel attacks are not an issue at Level-1 (L1) and Level-2 (L2) caches as they are usually kept private to their respective cores, eliminating many possible intrusion capabilities from adversaries. The LLC, however, is commonly shared among different processing cores, considerably increasing the efficient use of available memory space. Alas, this feature of LLCs intended to improve performance, introduces the possibility for one process to observe the usage of the cache by another process through cache contention [3].

The associate editor coordinating the review of this manuscript and approving it for publication was Peter Langendoerfer.

© 2026 The Authors. This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

Cache contention occurs when processes interfere with each other's access patterns to the cache. This underlying issue is characterized by conflicting accesses to the same shared cache location (cache set/cache index) by different processes running on different cores within the multi-core system [4]. Adversaries can instigate conflicting accesses to cause cache contention between an *attacker* process and a *victim* process, targeting sensitive information by observing cache behavior patterns as it is the case with the Prime+Probe attack [5]. Cache data leakages, i.e. information generated from observing cache behavioral patterns, are then utilized as cache side-channels to extract more information regarding the behavior of the cache. Cache side-channels can be employed by adversaries to trace the cache access patterns, divulging information that adversaries are unauthorized to obtain, such as the case with Spectre [6] and Meltdown [7] attacks. Methods can be developed to take advantage of cache contention to leak cache information, whether by measuring cache access timing [8], instigating conflicting accesses [9], [10], or by simply tracing accesses to the LLC [11]. Such methods pose a looming danger to computer systems and can be powerful tools when used properly by system adversaries. Therefore, it is becoming imperative to equip shared caches with the appropriate mitigation strategies to deal with these vulnerabilities.

To this end, protection of the LLC, or any shared cache in a multi-core system, is of the utmost importance. Due to the nature of the aforementioned threats to shared caches, software solutions may not be appropriate since those solutions themselves can be victims of cache contention [12]. Therefore, the response to these threats must originate within the design and architecture of the cache itself. Consequently, several architectural solutions for mitigating cache side-channel attacks were proposed. Previous solutions aimed at mitigating cache side-channel attacks use **i)** partition-based mitigation, **ii)** randomization-based mitigation, or **iii)** classification-based mitigation (differentiating sensitive data) [13].

First, *partition-based* approaches virtually divide the cache into partitions where a secure partition is used to store sensitive data and can only be accessed by secure processes. However, dividing the cache and reserving partitions for specific processes essentially truncates the available cache space for other processes, effectively causing performance degradation [14], [15], [16], [17].

Second, *randomization-based* mitigation aims at randomizing the memory-to-cache mapping at run-time [12], [18], [19], [20]. This randomization of cache accesses generates unpredictable cache behavior rendering any extraction of useful information, whenever cache contention occurs, indiscernible. Randomization often requires a re-mapping look-up table that keeps track of randomized accesses. While look-up tables implemented in hardware are a good solution for small caches (L1 and L2 caches), they can become non-viable for larger caches such as the LLC and can be vulnerable to different types of attacks. Moreover, randomization-based approaches introduce performance degradation [21].

On one hand, partition-based and randomization-based approaches perform mitigation either by targeting possible attack scenarios, preventing adversaries from exploiting cache side-channels, or by creating an unpredictable cache behavior, rendering any extraction of useful information by adversaries unfeasible [13]. On the other hand, these approaches usually target one type of attack – and its variations, if any – and could leave the cache vulnerable to other types of side-channels.

Third, *Adaptive mitigation* relies on differentiating sensitive data from non-sensitive data. Rather than focusing on changing the behavior of the cache as a whole, these approaches target changing the behavior of the cache when sensitive data is accessed. Differentiating sensitive data can be achieved using cache monitoring systems that can be used to detect possible cache information leakages or detect attacks on the cache. An attack monitoring system can activate the mitigation mechanisms of the cache whenever a threat presents itself, without the need to interfere with accesses that are not under threat.

Using a monitoring system such as Cache Leakage Detector (CLD) [22], provides the possibility to detect patterns that are indicative of exploitable cache information leakages. While not all cache information leakages are a threat, the majority of these leakages can be used as side-channels. By detecting such leakages, the cache behavior can be altered at run-time. This adds an additional layer of security to the cache design, while providing a sufficient mitigation mechanism for whenever threats manifest.

The Secure Cache Alternative Address Table (SCAAT) [23] employs such a scheme by relying on a run-time mitigation mechanism that is only activated when a threat is detected by the monitoring system. The mitigation is performed by substituting the cache address index bits with a new randomized location. By changing where cache blocks are stored, the overall timing and access patterns of the cache change, eliminating any extraction of useful information by adversaries. Once a cache access is re-mapped, an updated cache address is sent to the cache. However, SCAAT is ineffective if a data block (cache line) falls under threat after it is re-mapped. This is a stark vulnerability as it can reveal the re-mapped locations when a cache trace is performed for a long enough period.

In this work, the design, implementation, and analysis of a vulnerability-free mitigation mechanism named Recursive Secure Cache Alternative Address Table (RSCAAT) is presented. RSCAAT employs a highly reactive and robust mitigation control logic that is able to react to a wide range of possible cache behavior scenarios. To achieve the desired functionality of the RSCAAT, a specialized Content-Addressable Memory (CAM) is designed and implemented with built-in redundancies capable of self-remapping data without the need for external signaling. This allows the CAM

to continuously randomize locations of stored data when required during run-time.

Overall, the contributions of this work are:

1) A new vulnerability in the design of state-of-the-art mitigation architectures is identified and discussed.
2) Recursive Secure Cache Alternative Address Table (RSCAAT), a new vulnerability-free mitigation mechanism with a robust control logic and higher reactivity to cache behavior is proposed.
3) A specialized Content-Addressable Memory (CAM) is introduced and implemented with built-in re-mapping redundancies capable of eliminating predictable primitive polynomial LFSR traces.
4) RSCAAT is evaluated using benchmarks chosen from several benchmark suites against the identified vulnerability showing performance comparable to the state of the art while introducing significantly higher levels of security to the cache by accurately eliminating leakage patterns as well as recursively remapping vulnerable accesses.

The remainder of this work is as follows: Section II presents the related work and discusses several solutions proposed for mitigating cache side-channel vulnerabilities, In Section III, background in terms of hardware-based detection of threats and the target cache design are presented. The SCAAT architecture, its impact on cache behavior, and the aforementioned vulnerability are discussed in detail in Section IV, Section V presents the proposed RSCAAT cache vulnerability mitigation technique, Section VI provides the experimental results, and finally Section VII concludes the paper.

II. RELATED WORK

Thwarting cache side-channels is becoming among the most pressing issues of computer security. In the past, several solutions to address cache side-channel vulnerabilities have been proposed to different degrees of success. In this section, these solutions are divided into three categories: partition-based, randomization-based, and adaptive, differentiation of data based (classification-based) mitigation approaches.

A. PARTITION-BASED MITIGATION

Partition-based mitigation aims directly at limiting, or eliminating, cache contention between a secure process and an attacker process. This is done by separating where each of these processes can access, i.e. a secure process and an attacker process are assigned predetermined available cache sets that can be accessed by either process. These cache sets can be allocated either statically [24], [25], [26] or dynamically [14], [15], [16], [17], [27], [16]. Partitioning the cache is excellent at preventing the majority of external interference, and leakages resulting from cache contention caused by flush operations and evictions, but does not prevent internal interference and leakages resulting from the process' own access patterns.

For example, the statically partitioned cache presented in [26] and the dynamically partitioned SHARP cache [16] are vulnerable to variations of Cache Internal Collision attacks [2], [13]. Chameleon Cache presented in [28] dynamically switches how the cache is organized and divided into ways, effectively partitioning the cache under certain conditions. Similarly, SassCache [29] controls which cache set belongs to which security domain, segmenting the cache and rearranging the cache ways to isolate high-security loads from interference. Both Chameleon Cache and SassCache show strong resilience to attacks using dynamic and deterministic cache partitioning. Nonetheless, cache partitioning, in general, inherently introduces cache performance degradation [30]. This is largely due to underutilization of available cache space, the restricted number of cache sets available to each process at a time, and the time that is required to dynamically re-partition the cache (for dynamically partitioned caches).

B. RANDOMIZATION-BASED MITIGATION

Randomization-based mitigation targets disassociating the cache internal state (locations, cache miss, cache hit, etc.) and the cache operations (read, evict, flush, etc.) from any observable behavior (access patterns, timing, etc.) that the cache exhibits when data is being accessed. To achieve randomization, most approaches use a random number generator [20] and a look-up table to keep track of the re-mapped location to avoid coherency issues. Randomization can also be achieved using encrypting such as the case for CEASER [31], MIRAGE Cache [32], ScatterCache [19], and GIR-cache [33].

Such designs use randomisation in different ways to protect about information leakage. CEASER employs a Low-Latency Block Cipher (LLBC) to translate the physical line address into an encrypted one, ScatterCache keyed mapping that scatters cache lines unpredictably across cache sets, while GIR-cache uses indirect replacement with randomized placement to break predictable access patterns.

While randomization-based approaches are simple in principle, they come with considerable disadvantages. Look-up tables that track the changes to the access pattern to maintain coherent access to the cache can become extremely large when used with large-sized caches such as the LLC [21] and almost always leave the cache vulnerable to some variations of side-channels attacks [30]. Caches that use cryptographic randomization can be strong against Prime+Probe attacks but do not prevent Flush+Reload and Flush+Load attacks when the cache is shared. Furthermore, cryptographic randomization requires additional hardware to facilitate mapping/remapping adding area overhead and causing performance degradation.

C. ADAPTIVE MITIGATION

Partition-based mitigation relied on the knowledge of which data requires secure locations. Determining what data needs

to be stored in secure locations is achieved at the software level. Such software approaches are vulnerable to cache contention and require offline analysis of cache access patterns. Randomization-based mitigation scrambles the access pattern to the cache but introduces performance and area overheads. However, to achieve better performance and security, mitigation approaches target sensitive data at run-time (similar to partition-based mitigation) and interfere with the access patterns by changing the location of where a cache is being accessed (similar to randomization-based mitigation).

The Catalyst cache [27], for instance, classifies data as software-managed secure pages that can only be accessed from secure cache partitions, resulting in an acceptable performance overhead. While this is reminiscent of partition-based mitigation, its fundamental principle relies on the distinction between data that needs protection from data that does not. This approach still relies on software, which may prove too complex to design according to the needs of the cache and hard to implement in some circumstances [13].

Adaptive mitigation methods offer two significant advantages: First, caches that use these technique operate only to protect sensitive data from cache contention, concentrating the impact of mitigation on data that is under threat. Second, by only mitigating attacks on sensitive data, more flexible hardware solutions can be adopted, potentially targeting a wider variety of threats, while maintaining acceptable overheads.

Additionally, using hardware solutions circumvents the need for software support that can be costly in both design time and design cost. This requires architectural infrastructure to facilitate the detection of threats and mitigating them. Detecting cache information leakages at run-time manifests both advantages. A cache leakage detector, for example, [22], can be used to detect cache information leakages, preventing any exploitation of cache side channels by adversaries. Detection of such threats enables the use of mitigation mechanisms upon detection of threats. Using attack and cache information leakage detection systems (also referred to as monitoring systems) gives way to the development of secure caches without the need to allocate secure partitions or require area-expensive look-up tables. Monitoring systems can be incorporated to any cache design, making them a flexible solution for the detection of adversarial activities as opposed to secure cache architectures with fixed security features that generally come at a high area cost and performance overhead [18], [20]. SCAAT [23] employs this scheme by activating the mitigation mechanism whenever a signal from the detector is received.

SCAAT introduces the possibility of using different detectors to activate the security measures to prevent the exploitation of cache information leakages. As opposed to partition-based approaches, SCAAT does not partition the cache to any extent, allowing access to the complete memory space in the cache. While SCAAT uses a pseudo-random LFSR for re-mapping of threatened accesses, unlike randomization-based approaches, the randomization of access is only done when it is under threat. This feature allows other accesses that are not threatened – which make up the vast majority of accesses to the cache as demonstrated in Section VI – to be completed without interference from SCAAT. This avoids any performance degradation attached to randomization-based approaches.

SCAAT makes such advantages possible by identifying cache access patterns that can be used by adversaries as side channels. By targeting those leakages, SCAAT leaves no room for access-based attacks since access patterns that are used in these attacks can be avoided or interrupted before such patterns can be exploited. Any activation of SCAAT effectively changes the cache behavior and internal state (demonstrated in Section IV-C) to a point where trace-based attacks and timing attacks become unfeasible.

Unfortunately, shortcomings in the design of SCAAT itself make it vulnerable to attacks. The predictable nature of a primitive polynomial LFSR can be used to generate a full trace of the re-mapped data blocks, rendering the SCAAT ineffective and enabling attackers to bypass its mitigation efforts. In this work, RSCAAT is designed with redundancies that mitigates the possibility of such vulnerability. By adding additional control flow that relies more on the behavior of the cache itself during run time, the function of RSCAAT becomes more complex and less predictable. A specialized Self-Remapping Content Addressable Memory (*SR_CAM*) is designed with a built-in self-remapping function that foils any attempts to trace the predictable primitive polynomial LFSR. This function, along with the complex control logic, creates an unpredictable activation of the LFSR, further obfuscating the remapped data blocks within the *SR_CAM* achieving higher security for RSCAAT compared to its vulnerable SCAAT counterpart.

III. BACKGROUND

In order for RSCAAT to function properly, it must be activated when attacks are present in the system. This activation is designed to be triggered by a monitoring system that can detect possible attacks based on the behavior of the target cache. In this section, the detector used to activate the RSCAAT is discussed and the target cache design is presented.

A. DETECTION OF ATTACKS

While mitigation of threats on the LLC is essential, it must rely on accurate and reliable detection of attacks. Current solutions for attack detection are numerous with a wide range of implementations and effectiveness. However, since RSCAAT is intended for mitigation at run time, offline attack detection solutions, such as the ones presented in [13], [26], and [34], are not a viable option to work alongside RSCAAT. Therefore, purely online detection solutions are to be considered.

Approaches presented in [35], [36], [37], [38], [39], and [40] yield high attack detection and are accurate since they

collect information from system hardware monitors. For example, solutions presented in [35], [36], and [37], use the Hardware Performance Counters (HPCs) embedded into the CPU hardware. However, these solutions come with a glaring issue: additional offline analysis may be required to achieve the high detection and accuracy rates of these solutions. Thus, these approaches also are not a viable option to work alongside RSCAAT.

As RSCAAT is designed to work entirely in hardware, it is only fitting to couple it with a hardware detector. An option is LiD-CAT [41], a hardware run-time attack detection device that characterizes the information leakage by exploiting common cache attacks' characteristics. LiD-CAT showed a high rate of detection, but it was also shown to yield a high percentage of false positives. While LiD-CAT is sufficient to activate the RSCAAT, the high rate of false positives may cause a skew of results.

Therefore, in this work, the Cache Leakage Detector (CLD) [22] is used as the method of attack detection. CLD utilises a detection template that uses available information on cache outputs and process accesses at run-time to detect possible information leakages. By identifying potential information leakage patterns from cache behavior at run time, CLD is able to alert the mitigation mechanism to alter the access pattern before exploitation can occur. Here, CLD uses a threat model that defines how threats manifest in the system, a leakage model that defines how information leakages occur.

1) THREAT MODEL

The threat model sets forth the vulnerabilities of a given system. This model will describe what can be exploited and used to leak information from the system. The target system is a classical Von Neumann architecture, multiple processors with private caches connected to a shared last level cache (LLC). Each process in this system is assigned a unique process IDs (*ProcID*) and can access any location in the shared cache. In this system, several processes share the LLC causing mutual interference. A *Secure Process*(P_S) contains sensitive information and needs to be protected against possible leakage. Any process currently sharing the cache with P_S is assumed to be an *Interference Process* (P_I). P_I is a process that exhibits a cache access pattern that can lead to information leakage in the execution of P_S as a result of mutual cache interference. The emerging patterns from such mutual interference can be exploited by potential adversaries.

2) LEAKAGE MODEL

In order to facilitate leakage detection, a set of leakage scenarios were devised. A leakage scenario is a sequence of three stages: S_1 (setup stage) where the first access initiates a sequence of accesses that can lead to a leakage, S_2 (trigger stage) where a second access causes a mutual interference that leads to a leakage and S_3 (observation stage) is an access where the leakage information can be observed. Each stage of the sequence describes a cache access to the same cache location. These accesses can be performed by either P_S or P_I. The sequence of accesses results in two possible leakage sequences: The first sequence happens when three accesses to same cache location are preformed by P_S with an access pattern that can be exploited to leakage information by observing the resulting cache behaviour. The second sequence happens when P_I processes interfere with process P_S's execution. Such interference leads to an access pattern by a P_I process that causes cache contention with P_S.

TABLE 1. The transition table of the CLD FSM.

FSM transition *prev_state* → *next_state*	Transition condition (FSM input)	safety_check (FSM output)
$st_0 \rightarrow st_1$	(P_S or P_I) and (M or H)	0 (safe)
$st_1 \rightarrow st_2$	P_S and (M or H)	0 (safe)
$st_1 \rightarrow st_3$	P_I and (M or H)	0 (safe)
$st_2 \rightarrow st_1$	P_I and M	0 (safe)
$st_2 \rightarrow st_0$	(P_S and (M or H)) or (P_I and H)	1 (*unsafe*)
$st_3 \rightarrow st_1$	(P_S and H) or (P_I and (M or H))	0 (safe)
$st_3 \rightarrow st_0$	P_S and M	1 (*unsafe*)

Therefore, in both sequences, regardless of what information we have on the attacker, the cache access patterns by processes and the resulting cache contention leads to exploitable information leakage. A leakage can be directly observed whether any information on the attacker is available or not. The access and cache behavior patterns of process P can be indicative of any leakage of information. Therefore, CLD relies on obtaining cache output information, whether an access is a hit H or a miss H at the time of each access occurs.

By combining the leakage and threat models, CLD is implemented as a Finite State Machine (FSM) that keeps track of the access patterns to each cache location. Table 1 shows the transition diagram of the CLD FSM using the (*ProcID*) of either a safe process P_S or and interference process P_I from the threat model as well as the hit H or a miss H outputs from the cache.

The accuracy of CLD has been evaluated by injecting several different attacks into a 128-bit key AES algorithm. It was demonstrated that CLD has high detection accuracy, lower percentage of false positive detections, and a very low area overhead of 0.002% to the total area of the cache.

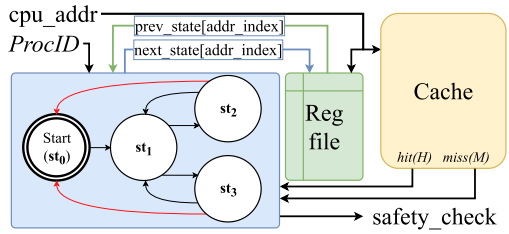


FIGURE 1. The state transition diagram of the CLD.

Fig. 1 shows the implementation of CLD's architecture with the corresponding FSM, register file, and cache. Scenarios where a leakage is detected are shown as a red line.

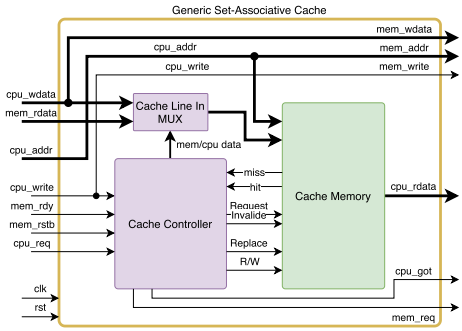


FIGURE 2. Generic cache architecture block diagram used in this work.

B. TARGET CACHE DESIGN

The RSCAAT mechanism, along with the monitoring system, aims at detecting and mitigating cache Logical Side-Channel Attacks (LSCAs). This mechanism is compatible with existing and any future direct-mapped and k-way associative cache architectures. To test the RSCAAT, the open-source IP core cache [42] was used. The reasoning behind this choice is the parameterizable and generic set-associative nature of the cache. Fig. 2 shows the block diagram of the generic cache architecture used. A generic Set-Associative cache design is adopted with a Cache Controller to facilitate the operations of the cache, a Multiplexer for reading data from the main memory and writing data from the CPU, and a Cache Memory block to store and retrieve data as needed. While this cache model is certainly simple in comparison to modern caches, it is used in this work due to its availability and ease of operation. A more complex cache can be used, however, the evaluation of the impact of incorporating the RSCAAT will be more difficult to quantify and may result in an inaccurate assessment of the RSCAAT performance.

IV. THE SECURE CACHE ALTERNATIVE ADDRESS TABLE

SCAAT [23] aims at mitigating LSCAs at run-time. LSCAs take advantage of predictable cache behavioral patterns to leak information regarding the cache's internal state. To alleviate this issue, SCAAT changes the predictable cache behavior when a threat manifests. In this section, the design, functionality, mitigation strategy, and SCAAT's impact on the behavior of the cache are presented. Additionally, a vulnerability of the SCAAT design and potential solutions to it are discussed.

A. DESIGN AND FUNCTIONALITY

SCAAT uses a real-time mitigation mechanism, called the *SCAAT_unit*, which is only activated when a threat is detected by the monitoring system. Fig. 3 shows the block diagram of the components and signals of the *SCAAT_unit*. *SCAAT_unit* has two inputs: the *attk* signal received from the cache monitors and the cache address (*addr*). When the monitoring system detects a threat, it sends the *attk* signal to a specialized logic that activates the different components of the *SCAAT_unit*. SCAAT mitigates attacks by changing the behavior of the cache at run-time. This is done by substituting the cache address index bits (*addr_index*) with a new randomized location. By changing where the cache is being accessed for threatening or suspicious behavior, the overall timing and access patterns of the cache change. This eliminates any extraction of useful information by adversaries and disassociates the internal state of the cache from the observable behavior.

Fig. 4 shows the transition diagram of the *SCAAT_unit*'s control logic.

B. CACHE COHERENCY CONSIDERATIONS

SCAAT aims at mitigating any access patterns that indicate possible information leakage or attacks on the cache. However, to guarantee that SCAAT will not interfere with the internal state of the cache, several cache coherency considerations must be taken into account.

First, The design of SCAAT must not change parts of the *addr* that point to cache data (being accessed or evicted). At the lowest level of cache (L1 instruction and data caches), the *addr_tag* (a reference to cache block) and *addr_offset* (a reference to a word being accessed) reference specific data being accessed in the cache. At higher cache levels (L2 caches and LLC), this only involves the *addr_tag* since, typically, only complete cache blocks are propagated from higher levels of cache [1]. Introducing changes to the *addr_tag* bits (higher cache levels) or *addr_offset* bits (lowest cache level) will result in invalid or incorrect data being sent to the CPU, causing cache coherency issues as a result of corrupted cache access requests.

Second, the mitigation strategy is only activated when a signal is received from the cache monitor or when any of the blocks that were re-mapped by SCAAT itself are later accessed. This limits the number of re-mapped *addr_tag*s to only those that are under threat and minimizes the possibility of coherency issues caused by SCAAT.

Third, to facilitate correct re-mapping, SCAAT must be able to remember which cache blocks were re-mapped to which location, hence the need for a specialized memory (*SCAAT_mem*). This is to make sure that future accesses to re-mapped blocks are performed properly, avoiding any coherency issues when re-mapped blocks require updating or need to be evicted from the cache. This ensures that all the index bits that were changed during the operation of mitigation are stored and retrieved as needed, minimizing any coherency issues that may result from changing those index bits.

Fourth, the mitigation of threats must be done without introducing any additional time cycles to the critical path of the cache. While coherency issues can be avoided by recording re-mapped blocks, the design of SCAAT can create coherency issues due to timing delays introduced when SCAAT changes the address being sent to the cache. This depends on the design of the cache that SCAAT is being integrated into and, naturally, must be taken into

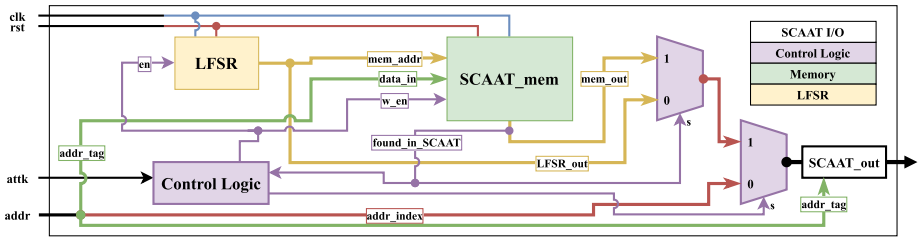


FIGURE 3. A block diagram of the components and signals of the *SCAAT_unit*.

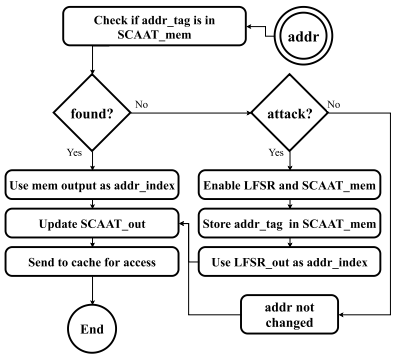


FIGURE 4. *SCAAT_unit* control logic transition diagram.

consideration when SCAAT is deployed. Since SCAAT is aimed at being integrated into any cache design, timing issues must be avoided to not introduce a delay to the cache itself. This is dependent on the design target of SCAAT. Incorporating SCAAT into a cache with strict timing constraints may require additional design considerations. However, an address must pass through the *SCAAT_unit* without disrupting the timing of the cache itself.

C. IMPACT OF MITIGATION ON CACHE BEHAVIOUR

The timing waveform in Fig. 5 illustrates the change in the cache access pattern and the resulting cache behavior when the *SCAAT_unit* is used. A single clock (*clk*) cycle is 1 unit of time u denoted by the function $t(u)$. Eight consecutive accesses, denoted using the *ACCESS* signal, are performed within a period of 16 clock cycles $t(0)$ to $t(15)$ in a given access pattern. The cache is accessed using an address divided into a block tag and a cache set. Block tags are denoted by T_x (colored blue) and cache sets are denoted by S_y (colored red) where x and y are the tag and set identifiers, respectively. Accesses under attack are highlighted in yellow. Cache evictions are highlighted in pink. The generated output from *SCAAT_unit* activation is highlighted in green. Signal P_{miss} indicates the predicted miss pattern and signal A_{miss} indicates the actual miss pattern as a result of *SCAAT_unit* activation. Signals cache[S_1] and cache[S_2] show the tag of the block stored in cache sets S_1 and S_2, respectively. Cache sets S_1 and S_2 are initially populated by tags T_2 and T_4, respectively.

In this example, the first access to the cache *Access*1 is under threat. This enables the *SCAAT_unit* which re-maps the access from S_1 to S_2. The tag being accessed is T_1 at location S_2, a miss occurs because T_1 is not in the cache and must be retrieved from higher memory levels. During *Access*1, T_4 was evicted as T_1 was brought from higher levels of memory and replaced T_4 at location S_2. During the next access, *Access*2, T_4 is predicted to be in the cache, however, due to the *SCAAT_unit* re-mapping of the previous access, T_4 was evicted. This is an example of how the *SCAAT_unit* actively altered the behavior pattern of the cache. T_4 should have been in location S_2 during *Access*2, but was evicted to make way for T_1, which was re-mapped by the *SCAAT_unit*. This change in behavior caused by the *SCAAT_unit* activation can alter the behavior of future accesses. An example can be seen for *Access*4. When *Access*4 is performed, it is predicted to be a miss, but since *Access*1 was re-mapped from S_1 to S_2 by the *SCAAT_unit*, T_2 was not evicted from S_1 and stayed there until *Access*4 was performed. During *Access*1, T_2 would have been evicted from location S_1 to make way for T_4. However, as a result of *SCAAT_unit* activation during *Access*1, T_2 was not evicted from S_1. This caused *Access*4 to be a cache hit, different than the predicted cache behavior.

An activation of the *SCAAT_unit* has changed the predicted miss patterns, shown as P_{miss}, of the cache during the time period between $t(0)$ and $t(7)$. The resulting miss pattern, shown as A_{miss}, demonstrates the effects of re-mapping accesses that are under threat on the overall miss hit patterns of the cache. Furthermore, the behavior changes are also reflected in future accesses as well.

As an example, consider *Access*5. A second threat was detected on *Access*5, causing the *SCAAT_unit* to re-map T_4 from S_2 to S_1. Now, all accesses to T_4 will be re-mapped to location S_1. At this point, T_1 and T_4 are re-mapped to locations S_2 and S_1, respectively. Any future access to either of these blocks will result in re-mapping each block to its respective new location.

The previous example shows that using the *SCAAT_unit* can reduce the predictability of the cache behavior, making it harder to collect accurate information regarding the cache operation conditions. This means that attackers who collect the aforementioned information are no longer successful in correlating the information collected with the actual behavior of the cache.

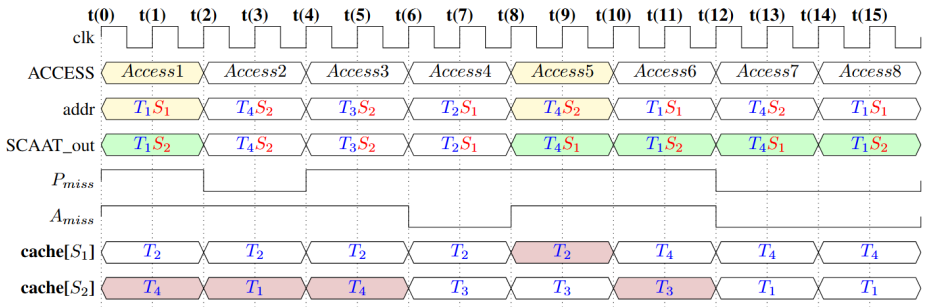


FIGURE 5. An example scenario demonstrating the change in cache access pattern.

D. REQUIREMENTS FOR SCAAT_MEM

The *SCAAT_mem* is a compact Content Addressable Memory (CAM) that allows storing T_x that are under threat. This allows a reduction in area overhead compared to other approaches that use look-up tables. For example, the baseline LLC used for this work and described in Table 3, has 4096 sets with 16 ways in each set. The baseline cache, therefore, requires 12 bits (S_{bits}) to access all sets of the cache. The number of bits required to identify each re-mapped tag (T_{bits}) is 20 bits. The *SCAAT_mem* has a depth equal to the number of cache sets (depth = 4096) and a width of T_{bits} (width = 20). In this case, the *SCAAT_mem* used with the LLC will require 10.24 KB of size. This size is comparably better than the look-up tables used by other approaches.

Randomization-based mitigation approaches use different versions of look-up tables to keep track of the randomized accesses to the cache. These tables can grow impractically large when used with large caches. Using the parameters of the baseline LLC a size estimate of such look-up tables can be made. Table 2 shows size estimations of randomization-based approaches.

TABLE 2. Size estimation for Randomization-based Look-up table approaches.

	Width	Depth	# of memories	total size (KB)
SCAAT/RSCAAT	20	4096	1	10.24
PT-Cache [12]	12	4096	8	49.152
NewCache [18]	15	4096	8	61.44
Re-configurable Address Mapping/translation Table [20]	32	4096	16	262.144

Although the previous size estimation for different implementations of look-up tables does not take into account the required logic to control and access such look-up tables, they are suitable estimates to evaluate the storage overhead introduced to the cache storage space by each look-up table. All in all, the compact and specialized use of the *SCAAT_mem* allows the implementation of a low-area CAM that stores all the required data to achieve the required randomization.

E. ATTACK MODEL AND SCAAT VULNERABILITY

While the *SCAAT_unit* provides sufficient randomization of cache accesses at a low area overhead, it also introduces a critical vulnerability. The baseline LLC described in Table 3 requires 12 S_{bits} to access all sets of the cache. As a consequence, the *SCAAT_unit* employs a 12-bit primitive-polynomial LFSR register to generate and store the pseudo-random sequences used by to scramble the access pattern to the cache. An N-bit wide primitive-polynomial LFSR will produce a maximal-length sequence of $2^N - 1$ unique N-bit wide combinations. Once this sequence is exhausted, the value stored in the LFSR register will return to its initial value, repeating the same sequence in the same order.

During operation, the *SCAAT_unit* stores threatened tags T_x in the *SCAAT_mem* at locations generated by the LFSR. Each T_x in the *SCAAT_mem* remains at the same location until the LFSR completes its maximal-length sequence. Thus, a tag T_x would only be replaced after $2^N - 1$ activations of the *SCAAT_unit*, triggered by threats on tags other than T_x. However, if a second threat on T_x is detected by the cache monitor before the LFSR sequence is completed, the *SCAAT_unit* is not reactivated, leaving T_x vulnerable.

In the worst-case scenario, an adversary can exploit this behavior. Algorithm 1 shows an attack flow that exploits the SCCAT vulnerability. By repeatedly attacking a known set S_x where T_x was previously re-mapped, the adversary can deduce the LFSR sequence. Since the LFSR sequence repeats deterministically, the adversary can generate a derived sequence $LFSR'$ beginning with the N-bit representation of S_x as it's initial state. Since the adversary can either already know or discover the cache depth and width, the length of the LFSR and possible primitive polynomials can be interfered. This information can be used to generate multiple candidate $LFSR'$ sequences. At this stage, an eviction-based attack such as Prime+Probe can be used to find the correct $LFSR'$ sequence. The adversary can target locations across the known candidate $LFSR'$ sequences and collect traces that can be compared to the randomized re-mapping performed by the *SCAAT_unit*. When a matching $LFSR'$ sequences

Algorithm 1 Attack Flow on SCAAT_unit and SCAAT_mem

```
Initialize LFSR with primitive polynomial of width N
Initialize SCAAT_mem as empty
while program executes do
    if (T_x under threat) then
        S_x ← LFSR_out
        SCAAT_mem[S_x] ← T_x
    end if
    while (Attack S_x repeatedly) do
        Record Access Trace
        Generate candidate LFSR'
        for each candidate LFSR' do
            if Trace correlates to LFSR' then
                LFSR' is true LFSR sequence
            end if
        end for
    end while
    Predict future remaps of T_x
    if LFSR completes full cycle 2^N − 1 then
        Replace oldest entries in SCAAT_mem
    end if
end while
```

is found, the adversary now has complete knowledge of the re-mapping sequence. At this point, the adversary can predict and exploit all succeeding re-mapping event, and the *SCAAT_unit* becomes ineffectual.

This vulnerability is particularly pronounced in large caches. For smaller caches, shorter LFSR widths yield shorter maximal sequences, causing frequent returns to the initial state and frequent replacement of tags in *SCAAT_mem*. In contrast, larger caches require wider LFSRs, resulting in longer maximal sequences and extended intervals before tags are replaced. This increases the exposure window for repeated attacks on the same tag.

V. RECURSIVE SECURE CACHE ALTERNATIVE ADDRESS TABLE

In this work, the vulnerability presented in the previous section is addressed. It can be overcome by adding a layer or remapping redundancy to the existing SCAAT CAM. Take a given cache tag T_x. When a threat to the cache set where T_x is stored is detected by the CLD, T_x is remapped to a new set S_x and stored in the CAM. This new set S_x can again fall under threat later on. This means that new logic is required to detect when a previously remapped T_x is still present in S_x when S_x is under threat. This logic is implemented as a re-mapping redundancy for the design of the Recursive Secure Cache Alternative Address Table (RSCAAT).

The RSCAAT is comprised of three components: A control logic that enables the different components of the RSCAAT, a specialized content addressable memory called the Self-Remapping Content Addressable Memory (*SR_CAM*), and a primitive polynomial LFSR.

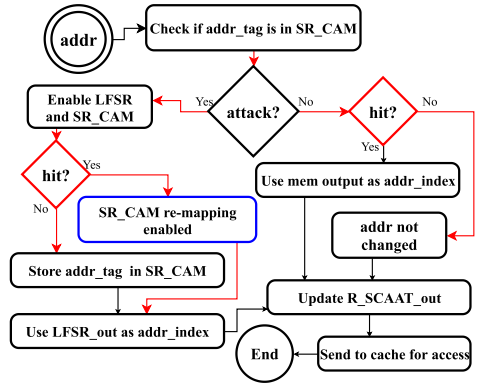


FIGURE 6. Updated control logic transition diagram for the RSCAAT.

A. RSCAAT CONTROL LOGIC

The control logic of the *SCAAT_unit*, shown in Fig. 4, can be reconfigured to achieve and take advantage of the SCAAT re-mapping redundancy required. Fig. 6 shows the reconfigured control logic transition diagram of the new recursive mitigation mechanism (called hereinafter *R_SCAAT_unit*) where the reconfigured control diagram transitions are highlighted in red. The *R_SCAAT_unit* has the same inputs (*clk*, *rst*, *attk*, and *addr*), and internal signals (*addr_tag*, *addr_index*, and *LFSR_out*) as the original *SCAAT_unit* control logic. However, the added redundancy re-mapping functionality, highlighted in blue, is discussed in detail in the next section. The *found_in_SCAAT* signal is replaced with a *hit* signal that is generated as an output of the *SR_CAM*. The output of *R_SCAAT_unit* is the re-mapped access.

The *R_SCAAT_unit* control logic leverages the signal received from the cache monitor. Once access to the cache is received, the *R_SCAAT_unit* checks for the signal sent by the cache monitor. This reconfigured control logic allows the *R_SCAAT_unit* to activate the re-mapping redundancy if T_x is a *hit* and the location where T_x is stored in the *SR_CAM* is under threat. If a threat signal is received from the monitor when T_x is being accessed, both the LFSR and the *SR_CAM* are activated. If the *SR_CAM* reports that T_x is not a *hit*, it means that T_x is not stored in the *SR_CAM*, then T_x is stored at the location of the current value in the LFSR register (*LFSR_out*). If T_x is a *hit*, the re-mapping redundancy is activated and the location of T_x in the *SR_CAM* is changed to the current *LFSR_out* value. In both cases, the *LFSR_out* is used as the index bits of the *R_SCAAT_out* output until T_x is either written into the *SR_CAM* (when T_x is not a *hit*) or changed to a new location in the *SR_CAM* (when T_x is a *hit*). If no threat signal is received from the monitor when T_x is being accessed, the *hit* output signal from the *SR_CAM* is checked. If T_x is a *hit*, the output of *SR_CAM* is used as the index bits of the *R_SCAAT_out* output. If T_x is not a *hit*, this means that T_x was not previously re-mapped and the unchanged access address is used as *R_SCAAT_out*.

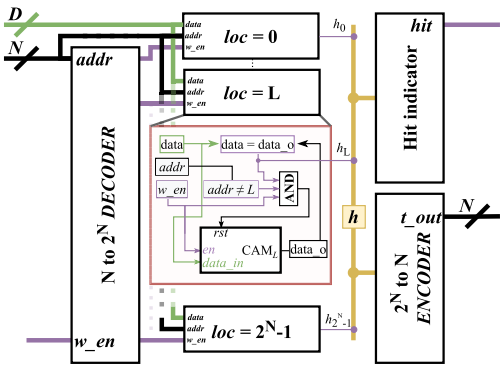


FIGURE 7. Components of the Self-Remapping Content Addressable Memory (*SR_CAM*).

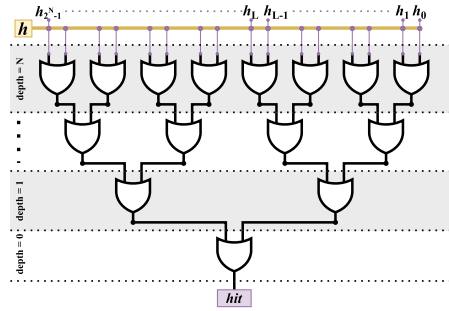


FIGURE 8. The structure of the hit indicator.

The *R_SCAAT_unit* control logic ensures that in the case of any threat detected by the monitor, the control logic reacts to potential threats to previously re-mapped locations. However, in the case where T_x is a *hit* and a threat signal is received from the monitor, the re-mapping of T_x inside *SR_CAM* is activated internally.

B. SELF-REMAPPING CONTENT ADDRESSABLE MEMORY

The *SR_CAM* is composed of four main components, illustrated in Fig. 7:

1) An N to 2^N decoder, where N is the bit width of the *SR_CAM* address, that controls the writing process of the *SR_CAM*. The decoder has two inputs: *addr* is a bit vector of length N that specifies to which location in the *SR_CAM* the writing process is performed and a *w_en* bit that enables the writing process. Both *addr* and *w_en* are received from the control logic of the *RSCAAT_unit*. The output of the decoder is a single-bit signal (*w_en*) connected to the location (*loc*) between 0 and 2^N-1, corresponding to the decoded *addr* vector.
2) The self-remapping mechanism ensures that no two locations in the *SR_CAM* contain duplicate data and allows the re-mapping of data internally. The mechanism has three inputs: *data* is a bit vector of length D that contains the T_x input data of the *SR_CAM*, the *w_en* bit received from the decoder, and the *addr* bit vector. The output is a single *hit* bit (h_L) that indicates if *data* is equal to the contents stored at $loc = L$. All the output *hit* bits from each of the locations in the *SR_CAM* coalesce into a single hit vector h of length 2^N.
3) A 2^N to N encoder that performs the read process of the *SR_CAM*. Its input is the hit vector h. The encoder returns the location *t_out* where *data* was found in the *SR_CAM* as output.
4) The hit indicator is an array of OR gates that resembles a full binary tree. It has one input and one output. The hit indicator takes the hit vector h and returns a single bit signal *hit*, which indicates if *data* was at all found in the *SR_CAM*. Fig. 8 shows the structure of the hit indicator. Each OR gate in the hit indicator is connected to two "child" gates. At $depth = 0$, the OR gates are connected to their respective bits of hit vector h as shown in the illustration. The OR gate at $depth = 0$ will return the result of OR the entire hit vector h resulting in either a '1' if there is an *SR_CAM* hit or '0' if no *SR_CAM* hits found.

The *w_en* must be a '1' in order to facilitate the writing of *data* into location *addr* in the *SR_CAM*. When *w_en* is enabled, the decoder sends the enable signal to the loc_L corresponding to *addr*. The data stored at loc_L (*data_o*) is compared to the input *data*. The *addr* is compared to the corresponding L. The output signal h_L is connected to the h bit vector. The *hit* and *t_out* signals are produced by passing the h bit vector to the hit indicator and the encoder respectively.

Fig. 9 shows the transition diagram of the re-mapping process. The results of the two comparisons, *data_o* with *data* and *addr* with L determine if T_x requires internal re-mapping inside the *SR_CAM*. The *w_en* is only used when a threat on T_x is detected. This means that T_x is only written into the *SR_CAM* in the event of a threat. If *data_o* at loc_L is equal to input *data*, it means that T_x is stored in the *SR_CAM* at loc_L and the h_L is returned as '1'. If L is not equal to *addr*, it means that the location in which the T_x is being written is not equal to the location where T_x was previously stored. If *en* is '1' (a threat is detected) and $data = data_o$ (T_x was previously stored in the *SR_CAM*) and $addr \neq L$ (T_x is being written into a new location), then the previous location L where T_x was previously stored is emptied (reset) and T_x is written into a new location corresponding to the input *addr*.

Once the *SR_CAM* is accessed, in case a threat is detected or not, the result of comparing the input *data* with *data_o* at all the locations from $L = 0$ to $L = 2^N - 1$ will result in a bit vector h of length 2^N. Since the re-mapping redundancy ensures that no two locations store the same data, the h vector will only have a single '1' bit (at the vector element corresponding to the location where *data* was a *hit*) while all the other bits in the vector are '0'. When the vector h is passed to the encoder, the encoder will return the

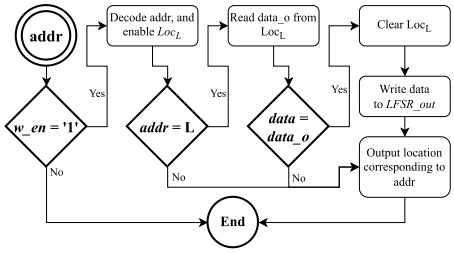


FIGURE 9. The re-mapping redundancy transition diagram.

binary representation of the location where *data* was a *hit*. Additionally, the hit indicator returns if the *data* is a *hit*.

C. SR_CAM LFSR

The *SR_CAM* derives its addresses from the LFSR, the CAM reaches full capacity when the LFSR reaches its maximal period. At the point, the LFSR has reached its maximum length sequence and will return to its initial state. The data at the *SR_CAM* location after the maximal period is overwritten with new data. A new tag that is not previously stored in the *SR_CAM* will be directly mapped to the value of the *LFSR_out*, overwriting data at that location. Therefore, a complex replacement policy is not needed and direct mapping of the new tag is used to reduce the complexity of the *SR_CAM*.

The bit width of the LFSR is crucial to the operation of *SR_CAM*. The LFSR must generate a sufficient number of combinations to cover all the sets of the cache. To achieve the pseudo-randomness expected from the LFSR, its width must be determined by the size of the cache. Specifically, the LFSR must be an N-bit wide primitive-polynomial capable of producing a sequence of maximum length $2^N - 1$ that is equal to the number of the available cache sets minus one. If the LFSR does not produce sufficient combinations to cover all cache sets, remapping would be restricted to only a subset of the cache. A confined remapping would leave cache sets unused during remapping, thereby increasing the likelihood of both cache and CAM misses. Such partial coverage can lead to non-uniform remapping, concentrating cache misses within specific regions and thereby undermining both performance and security.

The worst-case remapping chain length is bounded by the LFSR period $2^N - 1$ as a maximal polynomial LFSR is being used. While this scenario is theoretically possible, it is highly improbable due to the irregularity of cache access patterns in real programs. In order for such a case scenario to occur, a single cache line would need to be accessed in an exact pattern that repeatedly triggers a remapping to a sequence of locations that traverses the entire maximal LFSR period. This further assumes that no other cache lines are determined to be vulnerable by RSCAAT during the entire observation period and that no interfering events disrupt the progression of the remapping sequence. Under these conditions, a single cache line would be continuously remapped until the maximal period of the LFSR is achieved. The occurrence of such scenario will require extraordinary circumstances.

In the experimental setup, the CLD would require three accesses to make a determination whether a given location is vulnerable, which in turn would trigger the RSCAAT to remap it. Consequently, every remapping event requires three accesses. With a 12-bit LFSR, the maximal sequence length is $2^{12} - 1 = 4095$. To traverse the entire LFSR sequence, 4095 triggering events are needed. Since a single triggering event requires 3 accesses, the total number of accesses needed to traverse the entire LFSR period is $3 \times 4095 = 12285$ consecutive accesses to the same cache line without interruptions. Achieving this uninterrupted access pattern to a single cache line is practically infeasible.

D. RANDOMIZATION CONSIDERATIONS

For this work, the LFSR is used to facilitate randomization primarily for three reasons: a) LFSRs are very cost-effective in terms of hardware. Increasing the size of the cache requires only a logarithmic increase in the LFSR's width and complexity, resulting in negligible area overhead. b) LFSRs generate near-uniform pseudo-random sequences, but ensures non-repeating addresses during it's sequence. This guarantees that all cache lines within the cache are equally likely to be candidates for mapping/remapping when a cache line is under threat. c) LFSRs introduce low latency to the data path, allowing the *SR_CAM* to operate within a strict timing constraints with little to no impact on critical paths.

Other randomization techniques, such as those presented in [32], [19], and [33], use strong keyed or cryptographic randomization that yield uniform distribution and unpredictable access patterns. Such techniques are far more appealing than an a simple LFSR. Nonetheless, those techniques typically serve as the primary method of modifying the cache access behavior at high hardware complexity, area overhead, and potential latency costs.

In this work, however, the change in cache behavior is largely driven by the leakage pattern detection frequency by Cache Leakage Detector (CLD). Leakage patterns can be a result of any type of contention in the shared cache, whether it is caused by an attacker or by normally occurring contention. Thus, the observable behavior post-randomization is indistinguishable from a normally-occurring cache contention, regardless of whether the original contention stemmed from an attack or not. The LFSR only changes the mapping when a threat is detected, reducing unnecessary deviation of normal cache behavior. This means that the LFSR pattern is only visible when a potential threat is detected on the shared cache line, preventing long consecutive samples of the randomization output. This prevents standard LFSR state-recovery attacks that rely on observing linear relations across long sequences. Additionally, the remapping redundancy introduced by the *SR_CAM* further obscures any identifiable patterns that might otherwise reveal the LFSR's initial state post randomized access.

The combination of (1) detection-triggered randomization during any leakage pattern, (2) applying the LFSR in non-continual pattern, and (3) the remapping redundancy provided by the *SR_CAM*, effectively conceals the LFSR sequence and prevents the discovery of its initial state. Therefore, we choose an LFSR instead of stronger randomization schemes, such as those used in GIR-Cache, ScatterCache, or MIRAGE, because it provides sufficient randomization while significantly reducing hardware complexity, area, and latency.

E. SR_CAM SECURITY

While the current implementation of the *SR_CAM* adds a needed level of security to the RSCAAT, the security of the *SR_CAM* itself is difficult to assess. An advantage that *SR_CAM* has is its unconventional usage in the data path. While CAM-based tag lookup is predictable by the nature of their role in the cache design, the *SR_CAM* operates with randomness correlated with the LFSR. Without knowledge of the LFSR's initial state, the behavior of the *SR_CAM* is very difficult to predict. Even in case the sequence of the LFSR is tracked, a full knowledge of the *SR_CAM* contents is needed to predict it's behavior. Further work is needed to determine it's vulnerability to attacks normally used to target cache CAM-based tag lookup.

F. RSCAAT INTEGRATION CONSIDERATIONS

The proposed RSCAAT introduces an additional step to the cache access pipeline. Implementing this additional stage has an impact on the cache system performance. Such performance impact is implementation dependent and will vary based on the underlying cache architecture.

The primary latency concern comes from the access time penalty introduced with the *SR_CAM*. Since SCAAT performs both read and write operations to the *SR_CAM*, achieving timing closure may require an additional access cycle for *SR_CAM* access. A strict constant time behavior is enforced across all cache accesses. Even when *SR_CAM* content was not logically required, an access to the *SR_CAM* was performed to maintain uniform timing across all accesses. As a result, all *SR_CAM* accesses have the same latency, avoiding any conditional time differences that can be exploited as a timing side-channel.

In the evaluated cache model, RSCAAT was integrated without extending the critical path. RSCAAT is active during the tag comparison stage with no modification to the existing hit and miss decision logic. The additional *SR_CAM* lookup was done within the existing access window, consequently ensuring that the overall cache access latency remained unchanged in the baseline cache.

VI. EXPERIMENTAL RESULTS

In this Section, the experiments conducted to evaluate the RSCAAT are discussed. Several experiments were used to determine the effectiveness and security of the RSCAAT. Additionally, several metrics were measured to quantify the change in cache behavior when introducing the RSCAAT to the cache system.

TABLE 3. System configuration.

Processor Configuration		
Multi-core setup	4-core, inclusive multi-level caching	
Core parameters	32-bit (4B) x86 , 3.3 GHz, multi-threaded*	
Cache parameters		
Cache dimensions	64B cache block, 16 word block	
Private Caches	I/D L1	L2
	32KB direct-mapped 2 cycles latency	256KB 8-way associative 4 cycle latency
Shared Cache (LLC)	4 MB, 16-way associative, 20 cycle latency	

*Only enabled for benchmarks that support multi-threading.

A. EXPERIMENTAL METHODOLOGY

The experiments conducted for this work required several setup steps. This section discusses the setup of the experiments, the configuration of the system under test, and the combinations of workloads that are run on the multi-core system used during the experiments.

1) SYSTEM CONFIGURATION

For this work, a 4-core, 32-bit, x86, inclusive multi-level caching processor is used to simulate the workloads used in the experiments. This configuration of the processor is chosen to maximize the potential of cache contention between workloads in a 16-way, 4 Megabytes Last Level Cache (LLC). Additionally, this reflects the high performance expected from multi-core processors and their ability to handle several workloads at a time. While the LLC will be the target cache for all the experiments, it is important that the simulations reflect the expected process of data propagation throughout the multi-level cache system. To that end, two private lower-level caches are used. The L1 instruction and data caches are both a 32 Kilobyte direct-mapped cache while the L2 is a single 256 Kilobyte 8-way cache. The inclusion of those private caches allows some balance in the access rate of the workloads to the LLC. This system configuration is largely, but not exactly, modeled after the Intel®Core™i5-4570R Processor [43]. Table 3 shows the system configuration used for this work.

The RSCAAT mechanism is implemented on a traditional von Neumann architecture, chosen for its simplicity and well-characterized cache behavior. While this platform is inherently limited in scope, it provides an effective proof-of concept environment for exploring new cache-related ideas. Using such a baseline architecture demonstrates the feasibility of RSCAAT while avoiding the additional complexities introduced by modern multi-level caching systems. Moreover, the simplicity of the setup highlights potential areas for improvement and offers insights into how RSCAAT could be adapted for integration within more advanced memory hierarchies.

2) WORKLOADS

The system configuration used in this work allows 4 workloads to be executed at the same time. A benefit of such a setup is the use of different workloads that exhibit different access patterns to the LLC. To achieve this, an amalgamation of 26 different workloads from the SPEC CPU 2006 [44], SPLASH3 [45], and PARSEC [46] benchmark suites are used. To ensure a wide range of results and a broad spectrum of cache access behavior, a combination of 4 unique workloads is simulated at a time. Each combination of workloads includes at least two workloads from each benchmark suite. Furthermore, each workload is included in at least 3 combinations. A script utilizing the Python Library random generator functions is devised to implement the aforementioned scheme generating 25 unique 4-workload combinations.

3) EXPERIMENTAL SET UP

The multi2sim (m2s) simulator [47] is used to simulate the system configuration previously described. Using multi2sim, all the workload combinations are executed until each of the workloads in each combination has either finished executing or has reached at least 1 million accesses to the LLC. Through several steps of trial and error, several of the workloads that required configuration inputs, in the form of parameters or data or a combination of both, are carefully adjusted to allow each workload to execute properly without taking over the cache memory space. It is important to note here that this input adjustment was primarily done with workloads that exhibited resource-hogging behavior, i.e. largely data-driven workloads.

During system simulations, accesses to the LLC from the CPU side are recorded in the form of a trace file. To achieve accurate access timing, the Detailed Simulator of m2s is used. The m2s Detailed Simulator model simulates the entirety of the hardware structures, such as pipeline stages, registers, queues, functional units, and cache memories comprehensively [47]. This is done by using a timing-accurate x86 architecture model and simulating its behavior per simulated clock cycles. The default frequency of 3.3 GHz and the time model are used to generate the simulated system clock. A trace file is generated using the provided cache models of this architecture. The cache parameters are set according to the system configuration presented in Table 3.

Once the traces are produced, the cache accesses are fed into the cache model incorporating the SCAAT and the RSCAAT components. For every combination, the *CLD* is configured to detect cache contention and possible information leakage for one workload at a time. This allows the detection of cache contention and possible information leakage for all the workloads separately.

Once all simulation traces are fed into the cache model, new traces of cache accesses are generated. These new traces contain the location outputs by SCAAT and the RSCAAT and are used to generate the distribution of number of accesses per location for each workload after using SCAAT and RSCAAT, allowing more detailed analysis of the behavior of the cache before and after using SCAAT and RSCAAT. In addition, those new location traces are used to count and predict the behavior of the randomization mechanism in the RSCAAT system. The results of this experiment are then used to evaluate if the SCAAT vulnerability described in IV-E is present in the proposed RSCAAT.

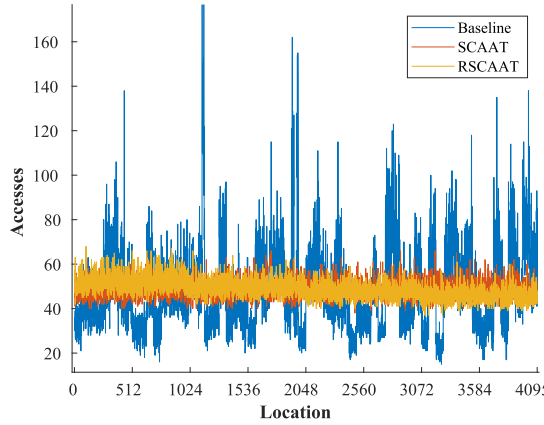


FIGURE 10. Distribution of the number of accesses per location in the LLC for the mcf benchmark.

B. ANALYSIS AND RESULTS

1) EFFECT ON CACHE ACCESS BEHAVIOR

Figure 10 shows the distribution of accesses to each LLC location for the *mcf* benchmark. This figure only shows accesses that occurred as a result of either re-mapping a cache block due to SCAAT or RSCAAT activation or as a result of access to a cache block previously re-mapped due to SCAAT or RSCAAT activation. The baseline shows the number of accesses to each cache location before SCAAT or RSCAAT were used.

The effect of using the RSCAAT on the behavior of the LLC cache is very clear. The baseline exhibits a distinct access behavior that can be observed by potential leakages. This can be used to distinguish workload accesses to certain locations of the LLC as certain locations have distinct access patterns. Using the RSCAAT not only mitigates those leakage-like patterns but also introduces an additional layer of security. The randomness in which the locations are remapped is compounded each time a cache location exhibits a leakage pattern. For every remapping done, the behavior of the cache is altered for both the locations where the access was supposed to happen and the location where the RSCAAT remapped the access. While using SCAAT does the same in terms of eliminating patterns, an attack using the attack model described in IV-E can bypass the pattern elimination and help an adversary discover the new location where a cache block will be remapped by SCAAT. This reduces the effectiveness of SCAAT. As shown below, such vulnerability is not observed in the RSCAAT remapping mechanism.

2) SECURITY

The design principle of RSCAAT is devised to mitigate the SCAAT vulnerability described in IV-E. An adversary can generate an *LFSR′* sequence to predict the next output of the SCAAT LFSR by targeting data blocks with tags T_0 to T_n at locations S_0 to S_n respectively. Predictably, this will intentionally alert the monitor and activate the SCAAT to re-map the data blocks to new sets. The adversary can now target sets at random and check if the data block (T_x) in the targeted set (S_y) is re-mapped to a new location by the SCAAT. If T_x is remapped, it means that S_y bit combination has not yet been reached by the SCAAT LFSR and is a combination that will appear after the current value of the SCAAT LFSR. If T_x is not remapped, this means that T_x has been re-mapped before by SCAAT and the S_y bit combination has already appeared in the SCAAT LFSR sequence and is a combination that appeared before the current value of the SCAAT LFSR. Due to the vulnerability of SCAAT, continuously checking which tags are re-mapped when being targeted helps narrow down where each S_y bit combination lies in respect to the current SCAAT LFSR value. This information allows an adversary to generate a reliable list of *LFSR′* sequences and can predict where each next re-mapping by SCAAT will be done. This type of exploit is effectively not possible in RSCAAT.

TABLE 4. RSCAAT security evaluation.

	SCAAT		RSCAAT	
	Vulnerabilities	Activations	Vulnerabilities	Activations
libquantum	211	110	189	189
streamcluster	142	106	139	139
swaptions	107	100	112	112

To test the security of RSCAAT versus SCAAT, 100 unique tags, called T_{set} hereinafter, with at least one occurrence of vulnerability were traced for three benchmarks with different SCAAT/RSCAAT activation rates: *libquantum* (high activation rate), *streamcluster* (medium activation rate), and *swaptions* (low activation rate). The initial value of the LFSR is set to be the same for all the benchmarks. This will ensure that the same LFSR sequence is used for all the traces. Table 4 shows that RSCAAT was activated for all instances of vulnerability that T_{set} have experienced. The table also shows that SCAAT was activated only a few times after the initial re-mapping. Overall SCAAT was activated at 53% for *libquantum*, 75% for *streamcluster*, and 93% for *swaptions*. In the case of *libquantum*, SCAAT is expected to be activated once for each vulnerability, however, it was activated 110 times. The 10 additional activation occurrences by SCAAT happened due to 10 tags being removed from the *SCAAT_mem* after it reached full capacity. While the same LFSR initial value is used for all the benchmark re-mapping, T_{set} experienced a different number of vulnerabilities for each benchmark. This can be attributed to the continuous change to the cache behavior, especially for RSCAAT.

The elimination of this exploit is largely due to the unique design of the *SR_CAM*. While SCAAT is not activated for tags that have been previously re-mapped, the re-mapping function in the *SR_CAM* is activated regardless of whether a tag has been re-mapped before or not. In order to generate a reliable list of *LFSR′* sequences, the adversary must be able to distinguish which tag T_x was re-mapped previously to predict where the location S_y bit combination lays in respect to the current value of the LFSR. However, unlike SCAAT, RSCAAT continuously re-maps all tags that are targeted. This prevents an adversary from knowing how many times a tag was re-mapped. Making information regarding the location of a certain re-mapped tag in relation to the current value of the RSCAAT LFSR unattainable.

3) IMPACT ON PERFORMANCE

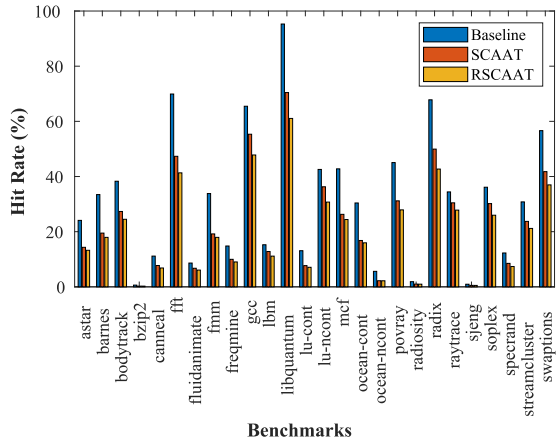


FIGURE 11. Resulting average hit rate of each benchmark executed as a secure process before and after using the SCAAT and RSCAAT mitigation mechanisms.

Due to the redundancy of RSCAAT, more cache misses are generated. In both SCAAT and RSCAAT, a miss is caused by either a true cache miss in the cache memory (cache miss) or when "remapped" location is removed from the CAM due to the CAM being full (CAM miss). The RSCAAT on the other hand, experiences far more CAM misses due to it's redundancy mechanism. When the RSCAAT detects a vulnerable location already mapped into the CAM, it remaps it again. This redundant remapping causes an additional cache miss that SCAAT does not normally experience. SCAAT would only remove a "remapped" location from it's CAM when the CAM is completely full. SCAAT maintains the "remapped" location for as long as the LFSR has not reached full round of generation, thus the "remapped" location experiences a hit for every access. This is not guaranteed during RSCAAT operation. A potential detection of vulnerability can cause RSCAAT to remap a "remapped" location again, increasing the overall miss count of the system. In future development, this can certainly be

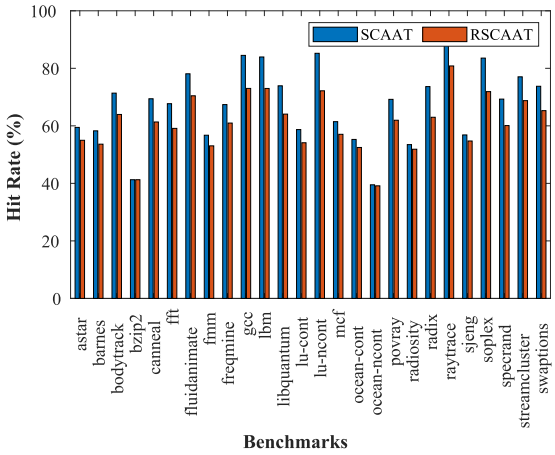


FIGURE 12. Normalized average hit rate of each benchmark executed after using the SCAAT and RSCAAT mitigation mechanisms. Baseline as 100%.

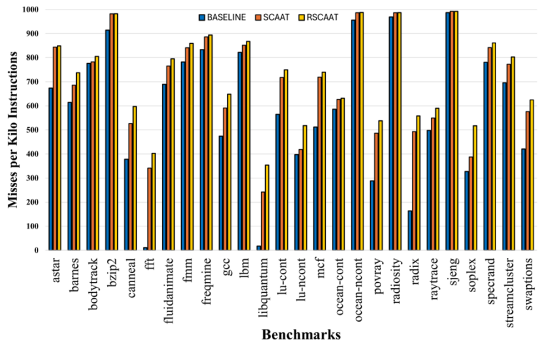


FIGURE 13. Average misses per kilo instruction per benchmark before and after the integration of SCAAT and RSCAAT.

improved by reducing the frequency of RSCAAT redundant remapping. However, such additions requires more control and monitoring of the system adding overhead and further work to determine if such methods are effective in reducing the overall miss rate.

Figure 11 shows the average hit rate of the benchmarks under test before (baseline) and after each benchmark is run as a secure process. Since each benchmark is used as a secure process with multiple combinations, the resulting hit rates from each run are averaged and shown in Figure 11. Using the SCAAT and RSCAAT contributes to performance degradation. For example, *libquantum* initially boasted a 95.29% hit rate. However, once it was executed as a secure process three times, the mitigation of detected leakage patterns reduced the average hit rate to 70.44% for SCAAT and 61.06% when the RSCAAT was used.

Furthermore, while performance degradation is introduced to the system at different degrees for different benchmarks, all benchmarks suffer performance degradation. This can be largely attributed to the remapping of cache blocks that exhibited access patterns that may indicate cache information leakage. Each re-map will cause a miss either by remapping the cache block to an empty location or forcing eviction of another block, which in turn contributes to performance degradation. Benchmarks with higher activation of the SCAAT and RSCAAT mechanisms suffered from higher degrees of performance degradation. The benchmark *gcc* had an average of two hundred thousand SCAAT/RSCAAT activation per run. Each activation is more likely to have caused a miss than a hit since *gcc* showed a reduction from a hit rate of 65.48% for baseline to 55.33% for SCAAT and 47.81% for RSCAAT.

Figure 12 shows the normalized average hit rate (where baseline hit rate = 100%) of each of the benchmarks after using the SCAAT and RSCAAT mitigation mechanisms.

Figure 13 shows Misses Per Kilo Instructions (MKPI) of the benchmarks under evaluation, comparing the baseline average MKPI to the average MKPI of each benchmark after the use of SCAAT and RSCAAT. The figure demonstrates an increase in the MKPI across all benchmarks, a trend corresponding to the decrease in hit rates. For instance, benchmarks such as *ftt* exhibited a significant increase from an average MKPI of 10.84 at a hit rate of 69.91% to an average MKPI of 341.18 with hit rate reduction of 22.58% when SCAAT is used and an average MKPI of 402.81 with hit rate decrease of 28.58% when RSCAAT is used. A similar trend is observed for other benchmarks such as *radix*. Other benchmarks such as *sjeng* and *radiocity* experienced a modest increase in the average MKPI. It is noteworthy that those benchmarks already demonstrated high MKPI prior to the implementation of either SCAAT or RSCAAT, which may explain the comparatively smaller impact observed.

The increase of MKPI across all benchmarks indicates a corresponding rise of Average Memory Access Time (AMAT). Since the LLC is shared in an inclusive cache hierarchy, an LLC miss incurs a main memory access and guarantees a higher AMAT for the executing workloads. An example of such effect can be deduced for the *ftt* benchmark. The integration of RSCAAT into the cache system contributed a substantial increase in MKPI, which translated directly into increased memory latency. The increase of MKPI from 10.84 for baseline to 402.81 after RSCAAT implementation is approximately a 37× increase in the LLC MKPI. This increase indicates a transition from a relatively low memory-sensitive workload to a highly memory-bound workload. Given that LLC misses cause main memory access penalties on the order of hundreds of cycles, such an increase significantly amplifies the memory AMAT and, as a result, degrades the the number of Instructions Per Cycle (IPC) that the cores can achieve. In practical terms, whereas the baseline configuration allows computation and memory access to overlap with limited stall impact, the post-RSCAAT system is dominated by long-latency DRAM accesses, causing execution throughput to collapse. These results demonstrate that RSCAAT significantly alters cache

behavior, shifting performance from being core-limited to memory-limited, with LLC miss amplification as the primary driver of IPC degradation.

4) AREA OVERHEAD

TABLE 5. Area increase per cache configuration.

		# Cache Lines				
	# Ways	256	512	1024	2048	4096
SCAAT	4	4.3%	4.24%	4.05%		
	8	4.44%	4.10%	3.81%	3.78%	
	16				3.76%	3.57%
RSCAAT	4	4.94%	4.79%	4.60%		
	8	5.00%	4.77%	4.27%	4.27%	
	16				4.24%	4.03%

In order to evaluate and analyze the area overhead and the critical path delay overhead of the proposed RSCAAT in comparison to SCAAT and the baseline cache, several configurations were synthesized using the Synopsys synthesis suite [48] and AMS 0.18μm CMOS technology library [49].

Table 5 indicate the area overhead incurred by both SCAAT and RSCAAT for several cache configurations. The percentages shown in the table are the area overhead incurred by SCAAT and RSCAAT compared to a baseline cache of the same configuration. The table shows that the area overhead decreases as the size of the cache increases. SCAAT added 4.3% overhead when integrated to a baseline 4-way, 256 cache line baseline cache. RSCAAT shows slightly higher overhead of 4.95% for the same cache configuration. Integrating SCAAT and RSCAAT to a slightly larger 8-way, 1024 cache line cache showed a decrease in the overhead percentage at 3.81% and 4.27% respectively. At the largest configuration of 16-way, 4096 cacheline, SCAAT overhead further decreased to 3.57% and RSCAAT decreased to 4.03% overhead. On average, SCAAT inured a 4% area overhead while RSCAAT inured 4.55% area increase. A steady decrease in area overhead is observed as the baseline cache capacity increases, indicating scalability at higher cache capacities. It is important to note that a single outlier can be seen for small size caches. Cache configurations with 256 cache lines showed an increase in overhead as the number of cache ways increase. This slight increase is most likely a synthesis artifact and does not have an effect on the overall scalability of SCAAT and RSCAAT.

The *SR_CAM* accounts for 93% of the overall size of the RSCAAT. Since the *SR_CAM* size correlates to the size of the baseline cache, as the size of the cache increases, the size of the *SR_CAM* would proportionately increase as well. A similar effect is observed in the SCAAT. However, in both RSCAAT and SCAAT, this proportionate increase only applies to their memories (*SR_CAM* and *SCAAT_mem*) and doesn't apply to the control logic of the size of the LFSR. For every additional power of 2 increase in the number of the cache lines, the LFSR width only increases by a single bit. While the control logic only increase relative to the width of the LFSR as it determines the dimensions of the decoder, the encoder, and hit indicator used in the *SR_CAM* and discussed in section V-B.

VII. CONCLUSION

This paper identified a new vulnerability in state-of-the-art mitigation architectures and proposed a Recursive Secure Cache Alternative Address Table (RSCAAT) as a highly reactive control logic coupled with a Logical Side-Channel Attacks (LSCA) mitigation mechanism designed to operate in conjunction with a cache leakage monitor. A specialized Content-Addressable Memory (SR_CAM) is introduced to continuously randomize locations of stored data during run-time using built-in redundancies capable of self-remapping data without the need for external signaling. RSCAAT alters the behavior of the cache by randomly remapping accesses to locations exhibiting leakage access patterns.

Results show that RSCAAT was activated for 100% of the leakage-like patterns detected while state-of-the-art solutions have been shown to only activate 53-93% for the observed workloads at the cost of a RSCAAT hit rate reduction of 11.62%. The remapping functionality of RSCAAT reduces leakage-like patterns that could have been used in adversarial attacks on the cache. Its recursive remapping of potentially vulnerable accesses mitigates possible exploitation of cache behavior to leak sensitive information.

RSCAAT is most effective in shared-memory architectures, where multiple caches rely on a common Last-Level Cache (LLC). In such environments, each shared cache can increase security by implementing RSCAAT independently from other shared caches, which introduces additional design overhead and can be considered a drawback of the mechanism. However, this requirement ensures that cache contention is systematically addressed across all levels of the hierarchy. By distributing the implementation, RSCAAT strengthens the robustness of the mechanism and highlights its potential for scalable deployment in complex systems.

REFERENCES

[1] J.-L. Baer and W.-H. Wang, "On the inclusion properties for multi-level cache hierarchies," *ACM SIGARCH Comput. Archit. News*, vol. 16, no. 2, pp. 73–80, May 1988, doi: 10.1145/633625.52409.

[2] D. J. Bernstein, "Cache-timing attacks on AES," Univ. Illinois at Chicago, 2005. [Online]. Available: https://cr.yp.to/antiforgery/cachetiming-20050414.pdf

[3] C. Xu, X. Chen, R. P. Dick, and Z. M. Mao, "Cache contention and application performance prediction for multi-core systems," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Mar. 2010, pp. 76–86.

[4] Q. Zhao, D. Koh, S. Raza, D. Bruening, W.-F. Wong, and S. Amarasinghe, "Dynamic cache contention detection in multi-threaded applications," *ACM SIGPLAN Notices*, vol. 46, no. 7, pp. 27–38, Mar. 2011, doi: 10.1145/2007477.1952688.

[5] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of AES," in *Proc. Cryptographers Track RSA Conf.* Cham, Switzerland: Springer, 2006, pp. 1–20.

[6] 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 *Proc. 40th IEEE Symp. Secur. Privacy*, May 2019, pp. 1–19.

[7] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *Proc. 27th USENIX Secur. Symp. (USENIX Security)*. USENIX Association, Aug. 2018, pp. 973–990.

[8] E. Tromer, D. A. Osvik, and A. Shamir, "Efficient cache attacks on AES, and countermeasures," *J. Cryptol.*, vol. 23, no. 1, pp. 37–71, Jan. 2010.

[9] Y. Yarom and K. Falkner, "Flush+reload: A high resolution, low noise, L3 cache side-channel attack," in *Proc. 23rd USENIX Conf. Secur. Symp.* Berkeley, CA, USA: USENIX Association, 2014, pp. 719–732. [Online]. Available: http://dl.acm.org/citation.cfm?id=2671225.2671271

[10] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+flush: A fast and stealthy cache attack," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, J. Caballero, U. Zurutuza, and R. J. Rodríguez, Eds., Cham, Switzerland: Springer, 2016, pp. 279–299.

[11] O. Acıiçmez and Ç. K. Koç, "Trace-driven cache attacks on aes (short paper)," in *Proc. Int. Conf. Inf. Commun. Secur.* Cham, Switzerland: Springer, 2006, pp. 112–121.

[12] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," *ACM SIGARCH Comput. Archit. News*, vol. 35, no. 2, pp. 494–505, Jun. 2007.

[13] S. Deng, W. Xiong, and J. Szefer, "Analysis of secure caches using a three-step model for timing-based attacks," *J. Hardw. Syst. Secur.*, vol. 3, no. 4, pp. 397–425, Dec. 2019.

[14] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "DAWG: A defense against cache timing attacks in speculative execution processors," in *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2018, pp. 974–987.

[15] 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. (TACO)*, vol. 8, no. 4, pp. 1–21, Jan. 2012.

[16] M. Yan, B. Gopireddy, T. Shull, and J. Torrellas, "Secure hierarchy-aware cache replacement policy (SHARP): Defending against cache-based side channel attacks," in *Proc. ACM/IEEE 44th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2017, pp. 347–360.

[17] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *Proc. 25th USENIX Secur. Symp.*, 2016, pp. 857–874.

[18] Z. Wang and R. B. Lee, "A novel cache architecture with enhanced performance and security," in *Proc. 41st IEEE/ACM Int. Symp. Microarchitecture*, Nov. 2008, pp. 83–93.

[19] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard, "Scattercache: Thwarting cache attacks via cache set randomization," in *Proc. 28th USENIX Secur. Symp.*, Aug. 2019, pp. 675–692. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/werner

[20] B. Niazmand, S. P. Azad, G. Jervan, and J. Sepúlveda, "Design and verification of secure cache wrapper against access-driven side-channel attacks," in *Proc. 22nd Euromicro Conf. Digit. Syst. Design (DSD)*, Aug. 2019, pp. 672–676.

[21] G. Keramidas, A. Antonopoulos, D. N. Serpanos, and S. Kaxiras, "Non deterministic caches: A simple and effective defense against side channel attacks," *Design Autom. Embedded Syst.*, vol. 12, no. 3, pp. 221–230, Sep. 2008.

[22] A. Shalabi, T. Ghasempouri, and J. Raik, "CLD: An accurate, cost-effective and scalable real-time cache leakage detector," in *Proc. 24th Int. Symp. Design Diagnostics (Don't Short) Electron. Circuits Syst. (DDECS)*, 2021, pp. 127–132.

[23] A. Shalabi, T. Ghasempouri, P. Ellervee, and J. Raik, "SCAAT: Secure cache alternative address table for mitigating cache logical side-channel attacks," in *Proc. 23rd Euromicro Conf. Digit. Syst. Design (DSD)*, Aug. 2020, pp. 213–217.

[24] D. Zhang, A. Askarov, and A. C. Myers, "Language-based control and mitigation of timing channels," in *Proc. 33rd ACM SIGPLAN Conf. Program. Lang. Design Implement.*, Jun. 2012, pp. 99–110.

[25] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A hardware design language for timing-sensitive information-flow security," *ACM SIGPLAN Notices*, vol. 50, no. 4, pp. 503–516, May 2015.

[26] Z. He and R. B. Lee, "How secure is your cache against side-channel attacks?" in *Proc. 50th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2017, pp. 341–353.

[27] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "CATalyst: Defeating last-level cache side channel attacks in cloud computing," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Mar. 2016, pp. 406–418.

[28] 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," 2022, *arXiv:2209.14673*.

[29] L. Giner, S. Steinegger, A. Purnal, M. Eichlseder, T. Unterluggauer, S. Mangard, and D. Gruss, "Sasscache: A secure skewed-associative cache with keyed index mapping," in *Proc. 44th IEEE Symp. Secur. Privacy (SP)*, 2023.

[30] F. Liu, H. Wu, K. Mai, and R. B. Lee, "Newcache: Secure cache architecture thwarting cache side-channel attacks," *IEEE Micro*, vol. 36, no. 5, pp. 8–16, Sep. 2016.

[31] M. K. Qureshi, "CEASER: Mitigating conflict-based cache attacks via encrypted-address and remapping," in *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2018, pp. 775–787.

[32] G. Saileshwar and M. Qureshi, "Mirage: Mitigating conflict-based cache attacks with a practical fully-associative design," in *Proc. 30th USENIX Secur. Symp. (USENIX Security)*. USENIX Association, Aug. 2021, pp. 1765–1782.

[33] W. Song, H. Ma, Z. Wang, D. Xie, and C. Ouyang, "PIR-cache: Mitigating conflict-based cache side-channel attacks via partial indirect replacement," in *Proc. 31st Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Jan. 2026, pp. 56–62.

[34] S. Deng, W. Xiong, and J. Szefer, "Cache timing side-channel vulnerability checking with computation tree logic," in *Proc. 7th Int. Workshop Hardw. Architectural Support Secur. Privacy*, New York, NY, USA, Jun. 2018, pp. 1–8, doi: 10.1145/3214292.3214294.

[35] M. Chiappetta, E. Savas, and C. Yilmaz, "Real time detection of cache-based side-channel attacks using hardware performance counters," *Appl. Soft Comput.*, vol. 49, pp. 1162–1174, Dec. 2016.

[36] M. Alam, S. Bhattacharya, D. Mukhopadhyay, and S. Bhattacharya, "Performance counters to rescue: A machine learning based safeguard against micro-architectural side-channel-attacks," *Cryptol. ePrint Archive*, p. 564, 2017.

[37] M.-M. Bazm, T. Sautereau, M. Lacoste, M. Sudholt, and J.-M. Menaud, "Cache-based side-channel attacks detection through Intel cache monitoring technology and hardware performance counters," in *Proc. 3rd Int. Conf. Fog Mobile Edge Comput. (FMEC)*, Apr. 2018, pp. 7–12.

[38] S. Briongos, G. Irazoqui, P. Malagón, and T. Eisenbarth, "Cacheshield: Protecting legacy processes against cache attacks," 2017, *arXiv:1709.01795*.

[39] M. Mushtaq, A. Akram, M. K. Bhatti, M. Chaudhry, V. Lapotre, and G. Gogniat, "NIGHTs-WATCH: A cache-based side-channel intrusion detector using hardware performance counters," in *Proc. 7th Int. Workshop Hardw. Architectural Support Secur. Privacy*, Jun. 2018, pp. 1–8.

[40] T. Zhang, Y. Zhang, and R. Lee, "Cloudradar: A real-time side-channel attack detection system in clouds," in *Proc. Int. Symp. Res. Attacks, Intrusions, Defenses*, 2016, pp. 118–140.

[41] C. Reinbrecht, S. Hamdioui, M. Taouil, B. Niazmand, T. Ghasempouri, J. Raik, and J. Sepúlveda, "LiD-CAT: A lightweight detector for cache attacks," in *Proc. IEEE Eur. Test Symp. (ETS)*, May 2020, pp. 1–6.

[42] Technische Universität Dresden. (2016). *Chair of VLSI Design, Diagnostics (Don't Short) and Architecture*. [Online]. Available: https://github.com/VLSI-EDA/PoC

[43] *Intel Core i5-4570R Processor (4M Cache, up to 3.20 GHz)–Product Specifications*. Intel Corporation. Accessed: Apr. 2, 2026. [Online]. Available: https://ark.intel.com/content/www/us/en/ark/products/76640/intel-core-i5-4570r-processor-4m-cache-up-to-3-20-ghz.html

[44] C. D. Spradling, "SPEC CPU2006 benchmark tools," *ACM SIGARCH Comput. Archit. News*, vol. 35, no. 1, pp. 130–134, Mar. 2007.

[45] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros, "Splash-3: A properly synchronized benchmark suite for contemporary research," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Apr. 2016, pp. 101–111.

[46] B. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, 2008.

[47] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2Sim: A simulation framework for CPU-GPU computing," in *Proc. 21st Int. Conf. Parallel Architectures Compilation Techn. (PACT)*, Sep. 2012, pp. 335–344.
[48] *Synopsys Design Compiler: Timing, Area, Power, & Test Optimization*. Synopsys, Inc. Accessed: Apr. 2, 2026. [Online]. Available: https://www.synopsys.com/
[49] *AMS 0.18 µm CMOS Process—AMS Process Technology*. AMS-OSRAM AG. Accessed: Apr. 2, 2026. [Online]. Available: https://ams.com/process-technology

AMEER SHALABI is currently a Junior Researcher with the Computer Systems Department, Center for Dependable Computing Systems, Tallinn University of Technology (TalTech). He is currently pursuing the Ph.D degree with TalTech, with a focus on cache memory systems, specifically in cache security design. He is interested in topics pertaining, but not limited to, computer architecture, computer memory, and hardware design and synthesis.

TARA GHASEMPOURI (Member, IEEE) received the Ph.D. degree in computer science from the University of Verona, Italy. During her Ph.D. program, she has researched the different kinds of verifications, specifically assertion-based verification on embedded systems. She is currently a Senior Researcher with the Computer Systems Department, Tallinn University of Technology. She is interested in finding innovative solutions for the verification process at different levels of abstraction. Her research interest includes security verification in hardware and software domains. She is a member of the IEEE Computer Society. She is a Program Committee Member of different conferences, such as DATE, ETS, and VLSI-SOC.

PEETER ELLERVEE (Member, IEEE) received the Dipl. Eng. degree from the Tallinn University of Technology, Estonia, in 1984, and the Ph.D. degree in electrical engineering from the Royal Institute of Technology, Sweden, in 2000. Currently, he is an Associate Professor and the Head of the Learning Centre, Department of Computer Systems, Tallinn University of Technology. He has taught or is teaching courses about digital systems and digital hardware modeling and synthesis. His research interests include behavioral and system-level syntheses, hardware-software co-design, and hardware description languages. He has published over 100 internationally reviewed articles in these areas and holds one patent.

JAAN RAIK (Member, IEEE) received the M.Sc. and Ph.D. degrees from TalTech, Estonia, in 1997 and 2001, respectively. He is currently a Professor of digital systems verification with the Department of Computer Systems and the Head of the Centre for Dependable Computing Systems, TalTech. He coordinated several European-level research and collaboration actions, including the Horizon 2020 RIA IMMORTAL, the Horizon 2020 Twinning TUTORIAL, and the FP7 Collaborative Research Project DIAMOND. He has co-authored more than 200 peer-reviewed publications. Until now, 20 Ph.D. theses have been defended under his supervision. His research interests include electrical engineering and computer science domains, including hardware test, functional verification, fault-tolerance, and security, as well as emerging computer architectures. He is a EUROPRACTICE and ARM Academic Access Representative, a member of the IEEE Computer Society, HiPEAC, and steering/program committees of numerous conferences within my field. He has been organizing top-level international conferences at the IEEE European Test Symposium, in 2020 and 2025, IFIP/IEEE VLSI-SoC, in 2016, and DDECS, in 2012, as the General Chair; the IEEE European Test Symposium, in 2019 and 2024, and DDECS, in 2013, as the Vice General Chair; and DDECS, in 2015 and 2023, and CDN-Live, in 2016, as the Program Co-Chair. He was awarded the Global Digital Governance Fellowship at Stanford University, in 2022, the HiPEAC Paper Award, in 2020, the Order of the White Star Fourth Class Medal by the President of Estonia, in 2016, and Estonian Academy of Science's Bernhard Schmidt Award for the innovation, in 2007.

Curriculum Vitae

1. Personal data

Name	Ameer Shalabi
Date and place of birth	8 November 1994 Masaha, Palestine
Nationality	Palestinian

2. Contact information

Address	Karakalliontie 14 G 50, 02620 Espoo, Finland
Phone	+358 406149988
E-mail	ameershalabi94@gmail.com

3. Education

2019–...	Tallinn University of Technology, Department of Computer systems, Information and Communication Technology, PhD studies
2017–2019	Tallinn University of Technology, Department of Computer systems, Computer and Systems Engineering, MSc *cum laude*
2008–2011	Bard College, Division of Science, Mathematics, and Computing, Computer Science, BA

4. Language competence

Arabic	native
English	fluent

5. Professional employment

2021 – ...	Nokia Solutions and Networks, SoC Design Engineer

6. Computer skills

- Operating systems: UNIX, Windows
- Programming languages: Python, C, C++, MATLAB.
- Hardware Description Languages: VHDL, Verilog.
- Scripting Languages: Bash Script, TCL, Tex.
- Tools: Synopsys, Cadence, ModelSim, MS office.

7. Defended theses

- 2019, Performance and Energy Evaluation of NVM-Based CIM and Memory Hierarchy, MSc, supervisor Prof. Jaan Raik, Tallinn University of Technology, Department of Computer Systems.
- 2016, FLD: A FIFO/LRU Decision Algorithm for Managing L1 Private Cache in Heterogeneous Processors, BA, supervisor Dr. Keith O'Hara, Bard College, Division of Science, Mathematics, and Computing.

8. Field of research

- Non-Volatile Memory.
- Cache Hierarchy.
- Computer Hardware.

11. Scientific work
Papers

1. A. Shalabi, K. Paul, T. Ghasempouri, and J. Raik. Nv-sp: A new high performance and low energy nvm-based scratch pad. In *2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 54–59. IEEE, 2020
2. 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
3. A. Shalabi, T. Ghasempouri, P. Ellervee, and J. Raik. Cld: An accurate, cost-effective and scalable run-time cache leakage detector. In *2021 24th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, pages 127–132. IEEE, 2021
4. A. Shalabi, T. Ghasempouri, P. Ellervee, and J. Raik. Rscaat: A recursive mechanism to mitigate cache side channel attacks. *IEEE Access*, 14:52342–52359, 2026

Elulookirjeldus

1. Isikuandmed

Nimi	Ameer Shalabi
Sünniaeg ja -koht	08.11.1994, Masaha, Palestiina
Kodakondsus	Palestiina

2. Kontaktandmed

Aadress	Karakalliontie 14 G 50 02620 Espoo, Soome
Telefon	+358 406149988
E-post	ameershalabi94@gmail.com

3. Haridus

2019–...	Tallinna Tehnikaülikool, Infotehnoloogia teaduskond, Info- ja kommunikatsioonitehnoloogia, doktoriõpe
2017–2019	Tallinna Tehnikaülikool, Infotehnoloogia teaduskond, Arvuti- ja süsteemitehnika, Magistrikraad *cum laude*
2008–2011	Bard College, Division of Science, Mathematics, and Computing Arvutiteadus, BA

4. Keelteoskus

araabia keel	emakeel
inglise keel	kõrgtase

5. Teenistuskäik

2021– ...	Nokia Solutions and Networks, kiibidisaini insener

6. Arvutioskused

- Operatsioonisüsteemid: UNIX, Windows.
- Programmeerimiskeeled: Python, C, C++, MATLAB.
- Riistvarakirjelduskeeled: VHDL, Verilog.
- Skriptimiskeeled: Bash Script, TCL, Tex.
- Teadustarkvara paketid: Synopsys, Cadence, ModelSim.

7. Kaitstud lõputööd

- 2019, Performance and Energy Evaluation of NVM-Based CIM and Memory Hierarchy, Magistritöö, juhendaja Prof. Jaan Raik, Tallinna Tehnikaülikool, Arvutisšteemide instituut
- 2016, FLD: A FIFO/LRU Decision Algorithm for Managing L1 Private Cache in Heterogeneous Processor, BA, juhendaja Dr. Keith O'Hara, Bard College, Division of Science, Mathematics, and Computing

8. Teadustöö põhisuunad

- Mittemuutmälu.
- Vahemälu hierarhia
- Arvutiriistvara

11. Teadustegevus
Teadusartiklite, konverentsiteeside ja konverentsiettekannete loetelu on toodud ingliskeelse elulookirjelduse juures.

ISSN 2585-6901
ISBN 978-9916-80-601-2