

Scramble Cache: An Efficient Cache Architecture for Randomized Set Permutation

Amine Jaamoun, Thomas Hiscock, Giorgio Di Natale

▶ **To cite this version:**

Amine Jaamoun, Thomas Hiscock, Giorgio Di Natale. Scramble Cache: An Efficient Cache Architecture for Randomized Set Permutation. Design, Automation & Test in Europe Conference & Exhibition (DATE 2021), Feb 2021, Grenoble, France. 10.23919/DATE51398.2021.9473919 . hal-03351626

HAL Id: hal-03351626
https://hal.science/hal-03351626

Submitted on 22 Sep 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons Attribution - NonCommercial 4.0 International License

Scramble Cache: An Efficient Cache Architecture for Randomized Set Permutation

Amine Jaamoum, Thomas Hiscock
Univ. Grenoble Alpes, CEA, LETI
MINATEC Campus, F-38054 Grenoble, France
Email: firstname.name@cea.fr

Giorgio Di Natale
Univ. Grenoble Alpes, CNRS,
Grenoble INP*, TIMA, 38000 Grenoble, France
Email: giorgio.di-natale@univ-grenoble-alpes.fr

***Abstract*—Driven by the need of performance-efficient computations, a large number of systems resort to cache memories. In this context, cache side-channel attacks have been proven to be a serious threat for many applications. Many solutions and countermeasures exist in literature. Nevertheless, the majority of them do not cope with the constraints and limitations imposed by embedded systems. In this paper, we introduce a novel cache architecture that leverages randomized set placement to defeat cache side-channel analysis. A key property of this architecture is its low impact on performance and its small area overhead. We demonstrate that this countermeasure allows protecting the system against known cache side-channel attacks, while guaranteeing small overheads, making this solution suitable also for embedded systems.**

I. Introduction

Nowadays, advanced micro-architectures become common in embedded systems. Furthermore, the development of the Internet of Things tends to make these processors accessible internet-wide. While physical attacks (such as side-channel analysis [1]) are considered for decades in the design of embedded systems, remote micro-architectural attacks only became relevant threats recently on these systems. Moreover, compared to side-channel attacks which usually require a physical access to the system, micro-architectural vulnerabilities may be exploited remotely. This makes these threats even more dangerous since they may allow a distributed attack on a large number of systems at the same time.

Cache memories are among the greatest source of micro-architectural leaks. Indeed, these components are shared among different processes and security domains. Furthermore, the state of a given cache is easily modifiable (by doing memory accesses or using flush instructions) and is also easily observable (by measuring the time of memory accesses). This is why so many micro-architectural attacks target caches. Many solutions have been proposed in the literature to cope with this type of attack. They mainly rely on cache partitioning (where a process cannot access a partition dedicated to another process), or on randomization of the content of the cache (in order to add noise to the measurements of the attacker and increase the time required to extract useful information from the leakage). Partitioning solutions are very robust against cache attacks, but they require important hardware redesign and can have a significant impact on performance. On the other hand, randomization generally has a much lower overhead, but it is more difficult to prove its efficiency in terms of security.

This work proposes a new secure cache architecture that leverages randomized set permutation, while allowing the designer to adapt the security level of the system, through a configurable refresh period of the random permutation. We demonstrate in this paper that this technique can be implemented efficiently and it also throws-out most cache side-channel attacks. The cache architecture is modeled in the Gem5 simulator [2], to evaluate its performance and its security. We complement the architecture with a security analysis and performance evaluation. The Scramble Cache architecture with a 8-way 32kB, a history table of 8 elements, and 8192 accesses to refresh the seed, achieves in the worst case 0.49% degradation of the hit-rate.

The rest of this paper is structured as follows. Section II describes our assumptions and gives an overview of cache attacks. Existing solutions are then described and compared in section III. The Scramble Cache architecture is presented in section IV. Finally, the performances and the security are discussed in section V.

II. Background

A. Security Model

In this work, we consider an embedded system with a single-core processor, which comprises a first-level cache. We assume this system can run multiple tasks due to the presence of a small operating system (such as FreeRTOS, VxWorks, or any other equivalent). An important assumption we make is that all tasks share the same address space. In other words, we assume that there is no Memory Management Unit (MMU) and, instead, memory isolation between processes is done through a static partitioning of the address space.

We assume that an attacker may gain privilege on the system by corrupting a task running on the operating system: either remotely by exploiting software vulnerabilities or through a physical attack (e.g., debug port or a memory corruption).

Being an embedded system, we also assume an attacker can measure the global execution time of applications through side-channel measurements [1]. In this model, most cache attacks described in subsection II-B can be performed. Even trace-driven attacks that usually require either hyperthreading or multiple-core architecture become relevant due to the time

*Institute of Engineering Univ. Grenoble Alpes

sharing within the same core. Since there is no MMU, the cache remains valid between context switches.

B. Cache Attacks

Cache attacks leak information on the addresses accessed based from various measurable elements on the cache: memory access time or performance counters (e.g., cache hit or miss counts). These micro-architectural attacks appeared very shortly [3] after physical side-channel attacks [1]. Since these works, several practical exploits were developed, leading to the discovery of Common Vulnerabilities and Exposures (CVEs) on widely used cryptographic libraries. A common classification, proposed by Acıçmez et. al [4], identifies three categories of cache attacks:

- *Timing-driven* attacks, which use the overall execution time of a process to deduce the behavior of the cache. They are classified as *passive* attacks (such as the Bernstein's attack [5]) when no modification of the victim cache are required and only address conflicts of the victim algorithm are exploited; or *active* attacks, when the attacker forces the eviction of specific lines of the cache, as in the EVICT+TIME attack of Osvik et. al [6].
- *Access-driven* attacks, which use the addresses accessed by the victim process to perform the attack. In these attacks, the adversary will usually put the cache in a known state and perform a differential analysis after the victim's execution. This category includes the PRIME+PROBE attacks [6], [7] as well as cache template attacks [8].
- *Trace-driven* attacks, which extend the timing driven categories by observing a very precise sequence of events: cache hit or cache misses. These attacks are very powerful as shown with the FLUSH+RELOAD [9] or FLUSH+FLUSH attacks [10]. However, they require a high-rate monitoring of the events, which is usually only possible on high-end hardware (multi-core architectures, or hyper-threading).

A more formal way to describe cache attacks is described by Deng et. al [11]. The attack is modeled as a three-step procedure (as shown in Figure 1): (1) The cache is placed into a defined state by the attacker, (2) The victim accesses some sensitive data, thus changing the state of the accessed cache lines; (3) The attacker makes some measurements and recovers sensitive information, which are correlated with the accessed cache lines.

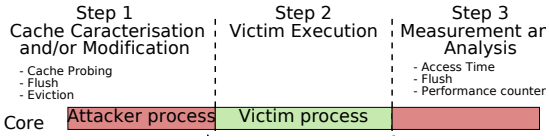


Fig. 1: Three step attack model of cache attacks on a single core considered in this work

III. RELATED WORK

The most straightforward defense against cache attacks is *cache partitioning*. The core idea is to separate different processes by isolating them into distinct regions within the cache, preventing interferences between a victim process and the attacker process. Static partitioning introduced by Page [12] creates partitions of cache at the hardware level. Intel proposes the Cache Allocation Technology (CAT) [13] on Xeon cores, whose primary goal is to get predictable performances for processes, but it can also be used as mitigation for cache attacks [14]. Nevertheless, CAT performs a way-partitioning of the cache, and it can isolate a limited number of domains (4 or 8). The NoMo cache of [15] uses way-partitioning of one or more ways of a set for each hardware thread and therefore, it suffers from the same limitations of CAT. The Partition Locked cache [16] performs finer-grained dynamic partitioning, where each process can lock a specific line of the cache and prevent it from being evicted by another process. This solution requires an Instruction Set Architecture (ISA) extension and the authors measured a 12% overhead on execution time.

An alternative to hardware partitioning is to *randomize* the cache behavior, which adds noise to the attacker observations. Non-intrusive randomizations include random eviction of lines [17] or random prefetching [18]. Random Permutation cache [16] (RP Cache) and NewCache architectures [19] are cache designs that use dynamic set remapping. The RP Cache maintains a permutation table that is updated as the lines are evicted from the cache. Thus, the cache requires an additional lookup when accessing memory. Still, the architecture performs better than partitioning solutions with an overhead below 1%. The ScatterCache [20] uses a keyed index derivation function (based on a strong cryptographic primitive) to obtain the locations of the ways. This architecture resolves the implementability issue of the RP Cache and manages to provide isolation without noticeable effects on performances (even some performance improvements were observed). CEASER [21] uses a dynamic remapping, periodically changing the memory-to-cache mapping on Last Level Caches (LLC). The authors measured a slowdown of 3% on a 8 MB, 16-Way LLC.

Table I compares the different existing solutions in terms of four criteria: security, overhead (considering performance and implementability), the scalability (regarding the number of threads) and whether an extension of the ISA is required. Random data placement is very effective against cache attacks and has very low effect on performances. However, a challenging aspect of these architectures is the hardware implementation of the permutation. Especially considering first-level caches where the access latency should be as low as possible. The Scramble Cache proposed in this work, uses a very simple permutation that can be implemented with only few logic gates. The security of the Scramble Cache is ensured by changing the address layout very frequently. Our architecture allows to select the security level and include mechanisms to limit the overhead on performances.

IV. THE SCRAMBLE CACHE ARCHITECTURE

A. Overview

The Scramble Cache implements an efficient randomization of the set locations. In contrast to partitioning techniques, this

TABLE I: Comparison of existing countermeasures

Solution	Security	Overhead	Scalability	ISA extension
Static Partitioning [12], [15], [16]	Good	High	No	Yes
Randomization [17], [18]	Low , Probabilistic	Low	Yes	Yes
Dynamic Remapping [16], [19]	Good, Probabilistic	Medium	No	Yes
ScatterCache [20]	Good, Probabilistic	Medium	Yes	Yes
CEASER [21]	Low , Probabilistic	Medium	Yes	No
Scramble Cache (this work)	Good, Probabilistic	Low	Yes	No

approach allows full cache sharing, which favors performance. The Scramble Cache uses a lightweight seeded permutation π_r to modify the set addresses. The permutation can be renewed at any moment by changing the seed value r. This can be done either after a fixed number of cycles, a fixed number of accesses to the cache, on interruptions or context switches.

Ideally, the Scramble Cache would change the permutation as frequently as possible in order to spread memory accesses over the whole cache, as shown in Figure 2. Interestingly, randomization tends to reduce internal cache collisions because addresses are not mapped to a fixed set anymore.

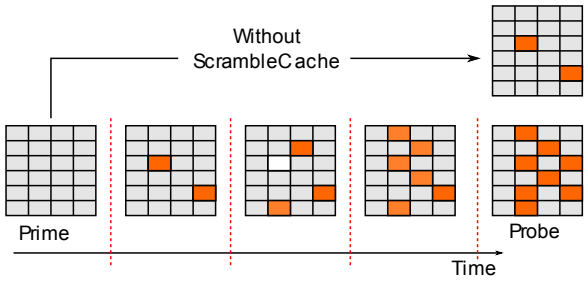


Fig. 2: Example of the Scramble Cache behavior

However, changing the cache address mapping introduces several issues:

1) *Coherency*: data writes in the cache (dirty lines) may not be properly written back to parent caches. We resolve this issue by performing write backs of all dirty lines on seed change. This cost can be reduced by keeping a history of dirty lines as we will demonstrate.
2) *Aliasing*: with permutation, unique data locations are not preserved because of the seed change, the same data may be valid at different places. We prevent this problem with a simple line generation tracking mechanism.
3) *Performance*: changing the seed frequently creates an important series of cache misses. We propose a history mechanism to move data from old locations, while preserving security properties.

We describe the Scramble Cache starting from a set associative cache, with physical tags where S bits of the effective address (the set bits) are used to locate the cache sets. The cache contains 2^S sets, each made on N lines, for a total size of $N \times 2^S$ cache words. The Scramble Cache architecture is depicted on Figure 3. Compared to a baseline cache, the architecture adds three important elements: (1) The permutation of the input address, (2) a generation management mechanism, (3) a history table. Random seeds come from a cache-internal

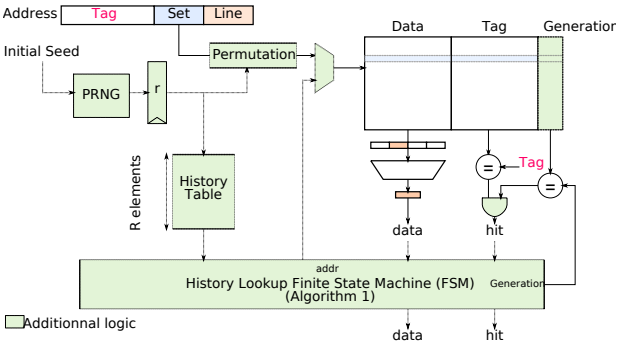


Fig. 3: The Scramble Cache Architecture

PRNG that is seeded randomly at reset. The modified cache control logic is found in the *History Lookup FSM* and is detailed on algorithm 1. The rest of this section describes the components of the architecture.

B. Address Permutation

The Scramble Cache applies a permutation on the set index extracted from the cache input address. Any address transformation (bijective or not) can be applied, because the cache always checks the physical tags before supplying the data to the CPU. We represent the permutation as a function $\pi_r(s)$, where r is a randomization parameter and s is the input set address.

A cheap way to implement a permutation in hardware is to use an "eXclusive-OR" (XOR) function, with $\pi_r(s) = s \oplus r$. However, a XOR only allows limited number of permutations. For example, when $S = 8$ (the cache has 256 sets), out of possible 256! permutations, only 256 can be reached with a XOR-based permutation. Therefore an attacker may successfully perform an exhaustive search of the permutation seed r. However, if the permutation changes many times between attacker analysis (as shown on Figure 2), all intermediate values of r have to be recovered, which makes the search more complex. But changing the permutation has a cost in terms of performance.

Thus, we suggest an alternative permutation family in the Scramble Cache, which has a wider seed input. The permutation is constructed by adding a layer of bit shuffling after the XOR, which can be viewed as a randomized barrel shifter. Formally, the π function takes the form:

$$\pi_r(s) = f(s \oplus r_0, r_1)$$

The function f is defined recursively, by dividing the binary representation of its input in two equal parts. The permutation

of a 2-bit wide input is denoted cswap, which stands for "conditional swap". Let $n = |s|$ be the bit length of $f(s, r)$. The function f first computes w of size n bits, with the i-th bit defined as:

$$w[i] = \begin{cases} \texttt{cswap}(s[i], s[i + n/2], r[i])[0] & \text{if } i < n/2 \\ \texttt{cswap}(s[i - n/2], s[i], r[i])[1] & \text{otherwise} \end{cases}$$

Then, the binary w representation is divided in two equal parts w_0, w_1 and the recursive formula $f(s, r) = f(w_0, r)||f(w_1, r)$ is applied. Figure 4 shows the permutation graph of f for the 4-bit wide inputs.

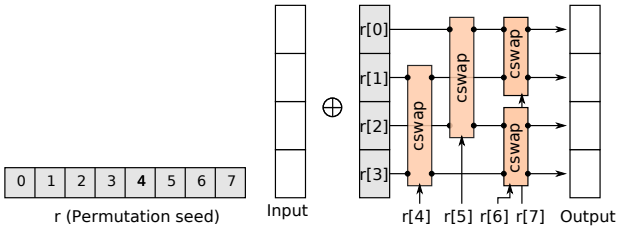


Fig. 4: Example of the f permutation for 4-bit wide addresses

C. Generation tracking for aliasing issues

Changing the permutation introduces a serious issue: the translation of an address may point towards old data. Accessing this data may discard modifications that have occurred since. To prevent the reuse of old data, we add to each line, besides the address tag, a *generation counter* to track how many generations away is each cache line, as shown in Figure 3.

The generation can be viewed as an infinite counter which is incremented each time the permutation changes. In practice, our cache can only keep track of limited generations with the history mechanism explained in subsection IV-D. Thus, for a history with depth R, the cache keeps a global generation counter c_{glob} updated as $c_{\text{glob}} \leftarrow (c_{\text{glob}} + 1) \mod R$. Each time data is brought in the cache, the line generation takes the value of the global counter. The relative age of a line can thus be computed with $(c_{\text{glob}} - c_{\text{line}}) \mod R$.

D. History Mechanism

The history mechanism is a core element of the Scramble Cache. It allows reducing the miss rate when changing the permutation by enabling older data to be moved directly to their new location. This is done by storing R previous seed values in a table (the *History Table* in Figure 3). To ensure strict process isolation, we associate a process identifier with each entry in the table.

Upon a miss, the cache yields control to an FSM, which takes care of the searching if the data is cached by using the previous R seeds, by implementing algorithm 1. This algorithm scans the history table in order, from the newest generation to the older ones, and checks if an older value of r leads to valid data. In case the data is valid, it is swapped with the current line to avoid looking in the table in the future.

On seed change, we still need to write back all lines that cannot be tracked by the history anymore. A FSM scans all sets linearly and all lines marked as dirty which have their counter $c_{\text{line}} = (c_{\text{glob}} - (R-1)) \mod R$ are removed from the cache and put into a write back queue.

Algorithm 1: History lookup algorithm

Input: Set address s, Process identifier p_{id}, isWrite

$s_{new} \leftarrow \pi_r(s)$
$c \leftarrow c_{\text{glob}}$
for *(r_{old}, pid_{old}) ∈ History Table* **do**
  $c \leftarrow (c - 1) \mod R$
  $s_{old} \leftarrow \pi_{r_{old}}(s)$
  if *generation*$[s_{old}] = c$ *and isTagValid*(s_{old}) *and* $p_{id} = p_{old}$ **then**
    swap(data$[s_{new}]$, data$[s_{old}]$)
    swap(tag$[s_{new}]$, tag$[s_{old}]$)
    generation$[s_{new}] \leftarrow c$
end

V. Evaluation

For our performance studies, we use Gem5 [2], [22], an event-driven, timing-accurate system simulator. The Gem5 simulator supports several CPU architectures, memory models, system calls emulation and different modeling granularities (functional or timing-accurate). This flexibility makes Gem5 a very powerful tool for design exploration.

A. Gem5 Modeling of the Scramble Cache

The Scramble Cache model that we developed in Gem5 handles requests coming from slave ports (CPU-side) and has a master port that interacts with the memory-side (usually a bus), as shown on Figure 5. Once a packet (Gem5's abstraction of read or write requests) is received on a CPU-side slave port, the Scramble Cache determines if the cache can accept the packet or not: the cache only handles a single request at a time. When the Scramble Cache accepts a packet, it switches into a blocked mode and delays the packet processing by a constant delay representing the hit access time. If the line is found (normal hit) the cache instantly sends the response to the CPU. Otherwise, the history table is browsed linearly to check if older data can be moved. Depending on the index of the entry, an appropriate latency is added (a multiple of the hit access latency). If the data is not present, the master port will be activated to send a read request to the main memory. The memory-side master port uses a queue of packets containing read and write back requests from the Scramble Cache.

B. System Architecture

To evaluate performances, we instantiate a simple system, depicted in Figure 5, that is similar in terms of memory architecture to an embedded system. The baseline configuration uses Gem5's built-in L1 data and instruction caches of 32KB, 8-Way associative, with a random replacement policy. The L1 caches are configured with a hit latency of 2 CPU cycles. A history lookup step of the Scramble Cache has a latency defined to 1 CPU cycle. For the rest of the section, unless specified, the Scramble Cache history has a depth of 8-entries and its permutation change interval is set to 8192 accesses. We use a *Linear Feedback Shift Register* (LFSR) to generate pseudo-random numbers, with latency modeled to 1 cycle.

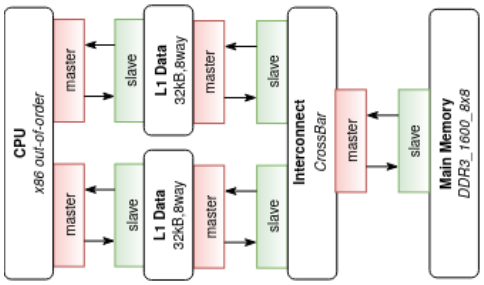


Fig. 5: System Simulated in Gem5

C. Performance Analysis

1) Benchmarking Methodology: We evaluate Scramble Cache performances by running workloads from the *Mibench* [23] benchmark suite in the small and large profiles. First, the simulations are validated on the baseline system without countermeasures, to obtain a reference time. Then, for different parameters, we replace the L1 data cache with the Scramble Cache and compare the performance results against the baseline configuration. Our analysis focuses on the following parameters: the size of the L1 cache, the depth history table, and the permutation change interval (expressed in number of cache accesses).

2) Impact of cache size: Figure 6 shows the hit rate change in a 4kB, 8kB, 16kB, and 32kB L1 data cache for different workloads. A higher hit rate overhead is better and 0% denotes no degradation. In comparison with the baseline cache, the average performance loss by the Scramble Cache using is 0.49% in the worst case (patricia). These results suggest that as the cache size increases, the Scramble Cache needs more time to write back all the dirty lines when the history table is full. With the exception of two workloads, which perform better because of the reduction in conflict misses by changing the memory-to-cache mapping.

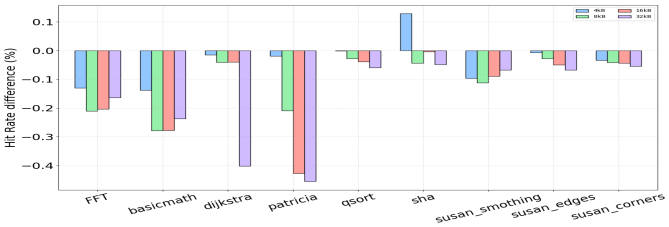


Fig. 6: Impact of L1 data capacity on Scramble Cache performance

3) Impact of the history table depth: To evaluate the effect of the history table depth, we vary this parameter on a 32 KB 8-way L1 data cache. Figure 7 shows the runtime overhead of the Scramble Cache for different history table depths. As expected, the overhead tends to decrease with a deeper history table. Indeed, increasing depth should reduce the number of misses caused by a permutation change. This is true up to a certain point. As we can observe on Figure 7, when the history table stores more than eight elements, the benefits are not always visible. At that point, the history table lookup (a linear search) is not a cheap operation anymore and it is almost as costly as reading directly from memory.

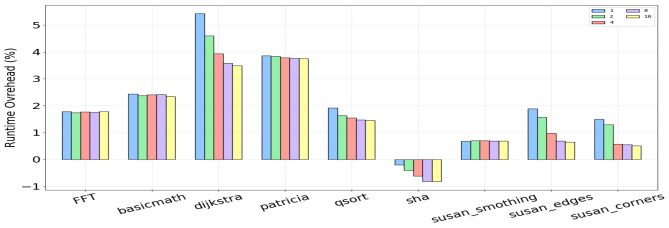


Fig. 7: Impact of history table depth on Scramble Cache performance

4) Impact of permutation change interval: The permutation change interval has a direct impact on the security of the Scramble Cache. The smallest acceptable value for this parameter should be selected in order to maximize security. However, changing the seed more frequently implies more misses, which at some point cannot be compensated with a deeper history table. Thus, for the refresh interval selection, a trade-off between security and performance is necessary. It can be seen from Figure 8 that when the seed refresh interval is long enough, the Scramble Cache approaches the performances of the baseline cache without countermeasures. However, if the seed is changing very frequently, the performances are decreasing very quickly.

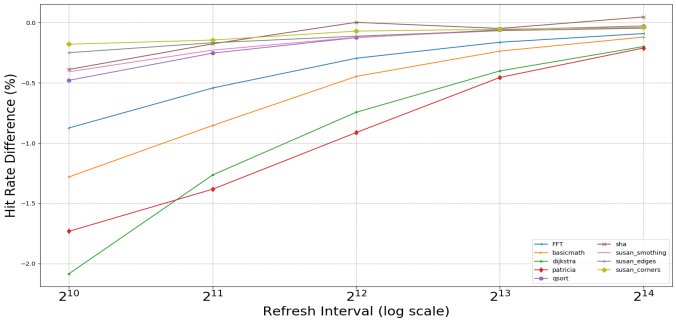
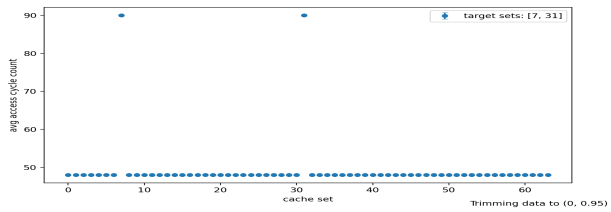


Fig. 8: Impact of remapping frequency on slowdown of Scramble Cache

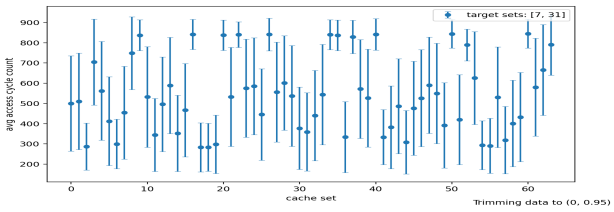
D. Security discussion

The Scramble Cache mitigates the FLUSH+RELOAD and FLUSH+FLUSH attacks thanks to the partitioning implemented in the history lookup table. If the attacker flushes a line, the next reload operation at the same address will fail with high probability. The only case for the attack to observe a hit, would be to guess the exact future location where the data will be stored after scrambling, thus allowing the attacker to perform the RELOAD operation on that location. The random nature of the permutation makes the new address of the data flushed uniformly distributed. Thus, an attacker has 2^{-S} (S is the number of bits in the set field) chance to observe the correct result. This means that a larger number of sets will decrease the efficiency of these attacks.

For other classes of attacks (address conflicts, PRIME+PROBE), the Scramble Cache adds extra noise to the victim memory access patterns to make them unexploitable. The noise level is directly related to the permutation change interval. To validate this statement, we perform a simple PRIME+PROBE attack on the Scramble Cache (data cache) using the Gem5 model. The victim and the attacker run in the same process. Between PRIME and PROBE, the victim reads two fixed memory locations once. Figure 9 shows the average cache access times measured by the attacker after the victim execution for each cache set. Times are expressed in cycles and obtained by the x86 instruction `rdtsc`. As measurements are noisy (especially with the Scramble Cache), we repeat the experiment 100 times on the unprotected cache and 10,000 times on the Scramble cache (permutation change every 8096 and a history table of 8 elements). For the latter, we plot the mean (as a circle) and the standard deviation (as a bar) of the measurement. It can be observed by comparing Figure 9a and Figure 9b that the Scramble Cache completely hides the memory access pattern, making PRIME+PROBE and address



(a) Unprotected version



(b) Scramble Cache

Fig. 9: Results of a PRIME+PROBE attack on the Scramble Cache

VI. CONCLUSION

In this paper, we presented Scramble Cache an architecture that implements efficiently set permutation. The keystones of the Scramble Cache are its generation tracking and history mechanisms, both allowing to frequently change the permutation. Thanks to the Scramble Cache modeling in the Gem5, we observed that a PRIME+PROBE attack is made much harder. The benchmarks demonstrated that the history table depth allows a trade-off between security and performance (under the assumption that changing the seed more frequently improves security). Indeed, with the permutation changing every 8192 accesses, the overhead on the execution time is below 4%, and we already observed security improvements against conflict-based cache timing attacks. Compared to existing remapping architectures (see Table I), the Scramble Cache achieves a similar security level as best-known solutions. Furthermore, its access latency is drastically reduced thanks to a cheap permutation, making this architecture usable as a first level cache in constrained environments.

ACKNOWLEDGMENTS

The authors would like to thanks the reviewers for their helpful comments. This work was funded thanks to the French national program "Programme d'Investissement d'Avenir IRT Nanoelec" ANR-10-AIRT-05.

REFERENCES

[1] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *International Cryptology Conference*, 1996.
[2] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, *et al.*, "The Gem5 simulator," *ACM SIGARCH computer architecture news*, 2011.
[3] D. Page, "Theoretical use of cache memory as a cryptanalytic side-channel.," *IACR Cryptology ePrint Archive*, 2002.
[4] O. Acıçmez *et al.*, "Microarchitectural attacks and countermeasures," in *Cryptographic Engineering*, 2009.
[5] D. J. Bernstein, "Cache-timing attacks on aes," 2005.
[6] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of aes," in *RSA conference*, 2006.
[7] C. Percival, "Cache missing for fun and profit," 2005.
[8] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," in {*USENIX*} *Security Symposium*, 2015.
[9] Y. Yarom and K. Falkner, "Flush+ reload: a high resolution, low noise, l3 cache side-channel attack," in *USENIX Security Symposium)*, 2014.
[10] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+ flush: a fast and stealthy cache attack," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 279–299, Springer, 2016.
[11] 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*, 2019.
[12] D. Page, "Partitioned cache architecture as a side-channel defence mechanism.," *IACR Cryptology ePrint archive*, 2005.
[13] C. Intel, "Improving real-time performance by utilizing cache allocation technology," *Intel Corporation, April*, 2015.
[14] 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 *International symposium on High Performance Computer Architecture (HPCA)*, 2016.
[15] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev, "Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks," *Transactions on Architecture and Code Optimization (TACO)*, 2012.
[16] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *Proceedings of the 34th annual international symposium on Computer architecture*, 2007.
[17] J. Demme, R. Martin, A. Waksman, and S. Sethumadhavan, "Side-channel vulnerability factor: A metric for measuring information leakage," in *International Symposium on Computer Architecture (ISCA)*, 2012.
[18] F. Liu and R. B. Lee, "Random fill cache architecture," in *International Symposium on Microarchitecture*, 2014.
[19] F. Liu, H. Wu, K. Mai, and R. B. Lee, "Newcache: Secure cache architecture thwarting cache side-channel attacks," *IEEE Micro*, 2016.
[20] 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*, 2019.
[21] 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)*, pp. 775–787, IEEE, 2018.
[22] A. Butko, R. Garibotti, L. Ost, and G. Sassatelli, "Accuracy evaluation of Gem5 simulator system," in *Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, 2012.
[23] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *International workshop on workload characterization. WWC-4*, 2001.