A Security Model for Randomization-based Protected Caches

Jordi Ribes-González[1], Oriol Farràs[1], Carles Hernández[2], Vatistas Kostalabros[3], and Miquel Moretó[3]

[1] Universitat Rovira i Virgili, Tarragona, Spain,
jordi.ribes,oriol.farras@urv.cat
[2] Universitat Politècnica de València, València, Spain, carherlu@upv.es
[3] Barcelona Supercomputing Center, Barcelona, Spain,
vatistas.kostalabros,miquel.moreto@bsc.es

Abstract. Cache side-channel attacks allow adversaries to learn sensitive information about co-running processes by using only access latency measures and cache contention. This vulnerability has been shown to lead to several microarchitectural attacks. As a promising solution, recent work proposes Randomization-based Protected Caches (RPCs). RPCs randomize cache addresses, changing keys periodically so as to avoid long-term leakage. Unfortunately, recent attacks have called the security of state-of-the-art RPCs into question.
In this work, we tackle the problem of formally defining and analyzing the security properties of RPCs. We first give security definitions against access-based cache side-channel attacks that capture security against known attacks such as Prime+Probe and Evict+Probe. Then, using these definitions, we obtain results that allow to guarantee security by adequately choosing the rekeying period, the key generation algorithm and the cache randomizer, thus providing security proofs for RPCs under certain assumptions.

Keywords: Cache side-channel attacks · Timing attacks · Randomization-based protected caches · Randomly-mapped caches · Pseudo-random functions · Security definition

1 Introduction

In computer architecture, caches are hardware storage components designed to mitigate the high-latency issues of main memory accesses. They act as intermediaries between the core and the main memory, and they store accessed data so that later accesses do not need to fetch data from main memory, thus reducing the access delay.

The inherent low latency of cache accesses is vital for performance, and yet it can cause security problems. The access latency leaks to cores the information of whether or not data already resided in the cache hierarchy before the access took place. This information has been used in a variety of microarchitectural attacks,

for example targeting cryptographic algorithms [LYG+15,IGI+16], cloud and sandbox environments [ZJRR14,OKSK15], keystroke timing [Mon18,SLG+18], the kernel space ASLR [GMF+16,GRB+17], or establishing covert channels as well [GMWM16,MWS+17]. Speculative execution attacks such as the Spectre attack [BSN+19,CVS+19,KHF+19], the Meltdown attack [LSG+18] and the Foreshadow attack [VMW+18] also exploit cache side channels.

We focus on side-channel attacks that exploit cache contention. Cache contention is the property of caches by which, given their small capacity with respect to main memory, cached addresses are bound to be evicted if enough accesses are made. In these attacks, the attacker uses the same cache as the victim process, but process isolation is active and the attacker does not have flushing privileges. Among the attacks that exploit cache contention, we focus on access-based attacks, which consist in using the memory access time of the attacker to gather information on the activity of victim processes. In this work we do not consider attacks that require more privileges, like timing the execution of processes [LWML16], flushing the cache [GMWM16], or using cache collisions with victim processes [Ber05].

The most well-known access-based attack is undisputably the Prime+Probe attack [Per05,OST06,LYG+15,MWS+17,SWG+17]. This attack starts by taking an initial large set of addresses, and reduces it so that all addresses end up cached when sequentially accessed. This reduced set of addresses is used to bring the cache to a known state. These addresses are re-accessed at a later time, and the latencies are observed in order to derive information on the access pattern of the victim process in between both access batches. The efficiency of Prime+Probe is improved by an order of magnitude in recent work [Qur19,VKM19]. Another attack covered in our work is Evict+Probe [BM06,DXS20], which assumes that the first batch of accesses is carried by the victim process instead of the attacker.

A solution that aims at protecting against access-based attacks is cache randomization. Randomization-based protected caches (RPCs, see the works in [WL07,LL14,LWML16,Qur18,THAC18,Qur19,WUG+19,DKMPHL20]) shuffle cache addresses so that accesses are distributed randomly in the cache. Their lower area and computational overhead makes them the most appealing candidates to prevent contention attacks.

Generic attacks such as [BDY+20,PGGV21] showcase a weakness of RPCs: any RPC is expected to become insecure if attackers are allowed enough accesses. This is due to the small size of cache memories. To overcome this, known RPCs [Qur18,THAC18,Qur19] establish a *rekeying period*. The rekeying period determines a number of cache accesses after which the keying material is refreshed and the cache is flushed. Assuming that the rekeying procedure changes the distribution of the address randomization unpredictably enough, this restricts the number of accesses at disposal to perform an attack.

All previous research on RPCs focuses either in building solutions to thwart known attacks, or in finding weaknesses of existing constructions. During this period, the security of many RPCs has been defended with strong claims, only

to be called into question by later work. The present work aims at bringing tools to stop this break-and-repair cycle.

1.1 Our contributions

As the transition from traditional to modern cryptography shows, a good way to provide well-grounded and concrete security claims is to converge to a provable security approach. The present work strives to define notions that characterize security against access-based attacks, and which can be used to assess the security of previous RPCs.

First, we present a model for cache systems and RPCs. Our model considers a single cache-level hierarchy, as we aim to protect the last-level cache, and it takes into account the leakage of access latencies. We then provide a definition that captures security against access-based attacks. To the best of our knowledge, this is the first security definition for RPCs that follows the game-based approach of cryptographic security definitions, where an adversary engages in an experiment and tries to maximize its success advantage. Our security definition applies to every published RPC up-to-date, in particular to [WL07,LL14,LWML16,Qur18,THAC18,Qur19,WUG+19,DKMPHL20].

Due to specific constraints of RPCs, standard cryptographic security definitions do not fit into this context. Namely, the size of caches is fixed and small, and cannot grow arbitrarily according to a security parameter. Instead, we adjust the security of the cache with the number of accesses the adversary is allowed to make, and we do so by establishing a rekeying period. We view the rekeying period as the security parameter of RPCs.

Our next step is to study the impact of rekeying and cache parameters on security. To this end, given a certain probability p and assuming that the cache randomizer behaves as a random oracle, we provide rekeying periods that guarantee all attacks succeed in breaking security during one epoch with advantage at most p. This result is ported to a scenario with noise conditions, and we give tools to extend the obtained security guarantees across various epochs. We complete our security analysis by studying rekeying periods that are known to allow efficient attacks.

Additionally, we obtain sufficient conditions for the cache randomizer to provide some security guarantees. We model cache randomizers as pseudo-random functions, and we see that requiring a certain security from an RPC imposes bounds on the security parameters of its underlying cache randomizer. This result allows to generalize our results to cache randomizers that are modeled as pseudo-random functions, instead of as random oracles.

Finally, we demonstrate our results through an application case. We illustrate the achieved results in the case of a commodity processor, for which we compute rekeying periods that provide concrete security guarantees. Our case of study shows that security can be extended across several epochs, providing significant security gains and increasing the time window where security is known to be enforced. The scripts used to perform the provided computations have been made

available at https://doi.org/10.5281/zenodo.6397296. In addition, we evaluate the impact of the rekeying period on performance using a cache simulator, showing the practicality of our work.

1.2 Related work

There exist two types of solutions to protect against access-based attacks. The first [WL07,DJL+12,KLA+18] consists of partitioning the cache, making each partition exclusive to only one process. Unfortunately, this isolation results in a poor usage of resources that can defeat the purpose of large cache memories. The second is randomization-based protected caches (RPCs), and it is the one considered in this work.

The first RPC, RPCache [WL07], applies a permutation of set indexes that is stored as a permutation table in memory, and changes this permutation every time there is cache interference between two processes. This approach is better suited for small caches, but it becomes impractical as cache size grows (e.g. for last-level caches).

To overcome the efficiency issues of RPCache, CEASER [Qur18] takes an encryption-based approach. In CEASER, a low-latency block cipher is used to encrypt and decrypt all physical addresses that enter and leave the cache. Keys are changed periodically to thwart known contention attacks. Despite their strong security claims, later attacks were shown to reveal information on the access pattern of the victim, either by breaking the block cipher used in CEASER [BGS+20,PGGV21], or with improved access-based attacks such as [Qur19,VKM19,PGGV21]. A similar solution is Time-Secure Cache [THAC18], which additionally implements per-process domain separation.

Subsequent work [Qur19,WUG+19] attempts to further enhance the security of RPCs by working over a non-standard type of cache memories called skewed caches [Sez93,QBBC09,SSW14,YCJQ18]. Skewed caches split their memory into various partitions. Addresses are mapped to a different cache set in each partition, and on a cache miss a partition is selected at random. Skewed-CEASER [Qur19] encrypts the addresses in each partition independently and renews the key periodically, albeit using a weak block cipher [BGS+20]. ScatterCache [WUG+19] randomizes addresses in the cache, doing it independently in each partition as well, but without key refreshing. Later work [BDY+20,PGGV21] acknowledges that using partitions considerably strengthens security, but finds that some key renewal is imperative, and that largely successful access-based attacks are still possible with the parameters chosen in Skewed-CEASER and ScatterCache.

Two previous works have examined the security properties of RPCs from a probabilistic point of view. In [BDY+20], a statistical analysis of RPCs is performed, and the obtained results are implemented in a tool that is used to evaluate the security of state-of-the-art RPCs, deeming them insecure. In [PGGV21], after presenting a new access-based attack, its success probability in breaking the security of several RPCs is computed, and this is used to better understand the influence of the RPC design parameters on security.

1.3 Overview of the article

This paper is organized as follows. Section 2 provides our models for cache memories, cache randomizers, and RPCs used in the rest of this article. Section 3 presents our single-epoch and multi-epoch security definitions for RPCs, and our pseudo-randomness definitions for cache randomizers and rekeying algorithms. Section 4 studies the security of RPCs, providing sufficient conditions to achieve provable security. We study the single-epoch and multi-epoch cases, on top of a scenario with noise conditions. We also study rekeying periods that are known to allow attacks that break security with enough advantage.Section 5 portraits an application example of how one may apply the obtained results, and a simulation that shows the impact of rekeying on the overall processor performance. The article concludes in Section 6 with a brief summary and some research directions.

As additional material, in Appendix A we state several assumptions on cache memories and cache randomizers that we take to define our models. Also, in Appendix B we give some considerations that affect the threat model of our security definition.

2 Randomization-based protected caches

In this section, we present our cache and RPC models. To guide the design of these models, we start the section by giving a primer on memory management and address translation that follows from the material in [Sar19,VKM19].

2.1 Memory management

The architecture surrounding memory accesses in modern computer systems roughly follows the diagram of Figure 1. It consists of the following components:

Main Memory: The *main* (or *primary*) *memory* consists of separately-addressable memory *blocks*, organized into disjoint *frames*, each of the same size 2^p. Each block is addressed by a *physical address*, which is a concatenation of the *frame number* (representing its frame) and *page offset* (representing the position of the block inside the frame).

CPU: The CPU executes processes, which in turn need access to the main memory. However, processes may need to have exclusive access to memory blocks for security (process isolation), and it is convenient for the memory space to be linearly arranged. Hence, the CPU accesses the main memory through *virtual* (or *logical*) *addresses*, which are sequential and are ultimately mapped to physical addresses. Virtual addresses consist of a *page number* and a *page offset*.

MMU/TLB: The memory management unit (MMU) maps virtual addresses to physical addresses, according to a per-process *page table*. The storage and maintenance of this table is carried out by the MMU-managed translation lookaside buffer (TLB) and by the operating system (OS). The MMU implements process isolation, and it lets processes share data if and only if their page tables share a frame number.

Cache: In this work, we focus on caches that reside the farthest to the core (see Appendix B), called Last-Level Caches (LLC). Such caches take as input physical addresses, retrieve the corresponding data from main memory if necessary, and forward the data to the CPU. In the process, they save the received data in its internal storage, so that future accesses avoid the high latency of main memory.

The cache internally maps physical addresses to *cache addresses* to perform the internal storage. In the typical set-associative paradigm, the internal storage of the caches comprise 2^c *cache sets*, each indexed by a *set index*. Every cache set is divided into *cache lines*, each capable of holding the data of several blocks each. All cache sets hold the same amount a of cache lines, which is known as the *associativity* of the cache.

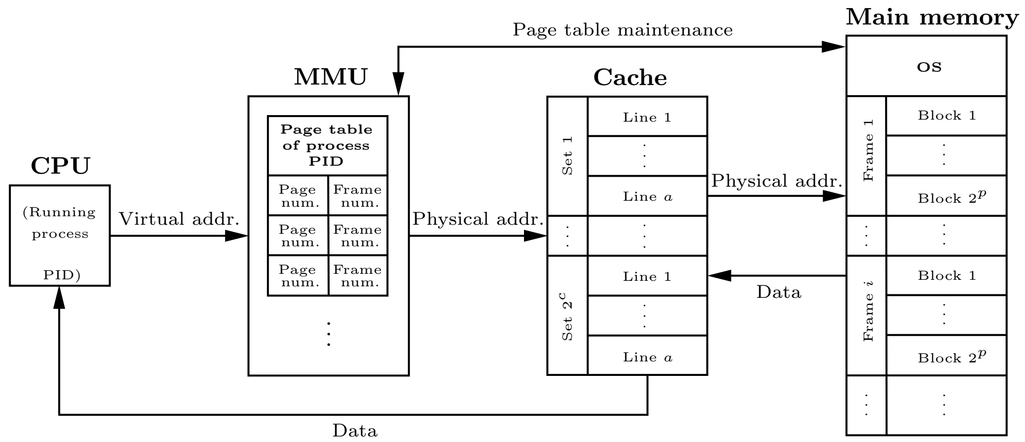

Fig. 1: Components involved in the memory management of conventional modern computer systems [And19].

2.2 Address translation

The virtual to physical address translation is carried out by the MMU. The translation consists simply on replacing the page number portion of the address by the possibly smaller frame number associated to it, according to the page table of the current process.

Cache addresses are divided into three parts: the *set index*, the *tag*, and the *offset*. Physical to cache address translation is done by assigning the most significant g bits to the tag, the next c bits to the set index, and the rest to the offset.

When data enters the cache, the set index of its address identifies the cache set where it is to be stored. The particular cache line where it will be stored

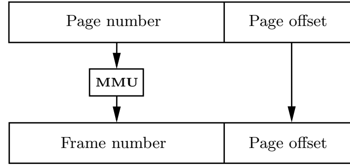

Fig. 2: Virtual-to-physical address translation.

depends on the *replacement policy* inherent to the cache. Once a line is chosen, the offset determines which position of the cache line the data will reside in. Finally, the data block is stored alongside the tag in the line. The tag uniquely identifies the line where the data resides inside the cache set.

In the described setting, one can find roughly two main types of caches depending on the form of this translation:

Virtually indexed: In this case, the set index can be generated from the virtual address and in parallel to the TLB lookup, see Figure 3. Processes are in full control of the set index part of the address. Process isolation in the cache is guaranteed by tags, since physical addresses with different frame numbers are translated to cache addresses with different tags.

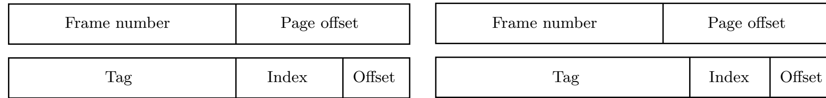

Fig. 3: Physical-to-cache address translation in virtually indexed caches.

Physically indexed: In this case, the set index is generated, at least in part, from the physical address (i.e., the frame number), see Figure 4. When different frame numbers give rise to the same tag, process isolation in the cache is enforced by the page color part of the frame number.

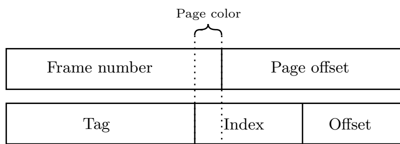

Fig. 4: Physical-to-cache address translation in physically indexed caches [VKM19].

2.3 Cache model

In this section, we describe an abstraction of caches for the purpose of defining a notion of security against access-based attacks. Find in Appendix A the set of assumptions on cache systems and cache randomizers that shapes our models.

The following definition establishes some notation for indexes, tags and cache addresses. Note that we disregard the offset part of addresses, since it is irrelevant for the considered cache timing side-channels.

Definition 1 (Set Indexes, Tags and Addresses). *Let c and g be positive integers, and let $S \subseteq \{0,1\}^c$ and $T \subseteq \{0,1\}^g$ be sets of fixed-length binary strings. We call S the* (set) index space*, and T the* tag space*. Elements of S are called* (set) indexes*, elements of T are called* tags*, and elements of $S \times T$ are called* addresses*.*

Our model for cache systems comprises an abstraction $\mathtt{C}$ of the physical cache memory, defined as an array that stores a fixed-length cache set $\mathtt{C}[s]$ for every $s \in S$. To model cache accesses, we define a function $\mathsf{access}(\mathsf{RP},(s,t))$ that models accesses to the tag t in the cache set $\mathtt{C}[s]$. To account for the leakage of the latency of cache operations, our model reveals on each access a latency bit h. If there is a cache hit ($t \in \mathtt{C}[s]$), it outputs the bit $h = 0$. Otherwise, if there is a cache miss ($t \notin \mathtt{C}[s]$), it outputs the bit $h = 1$. In the process, this function uses a replacement policy RP to compute a cache line $\ell = \mathsf{RP}((s,t))$, and it stores the tag t in $\mathtt{C}[s][\ell]$ (possibly causing an eviction if $\mathtt{C}[s][\ell]$ was not empty).

Definition 2 (Cache System). *Let S be an index space and T be a tag space. A* cache system over S, T *(or simply* cache system*) is a tuple $(\mathtt{C}, \mathsf{RP}, \mathsf{access})$ that consists of:*

$\mathtt{C}$**:** *An array indexed by the set S. For every $s \in S$, it stores an ordered array of fixed length that we denote by $\mathtt{C}[s]$. We refer to $\mathtt{C}$ as a* cache*, to the arrays $\mathtt{C}[s]$ as* cache sets*, and to each of the positions of cache sets as their* cache lines*. We assume that $\mathtt{C}$ is a global variable that can be modified both by RP and access.*

$\mathsf{RP}(x)$ *: A possibly stateful algorithm that takes as input an address $x \in S \times T$. It outputs a cache line $\ell = \mathsf{RP}(x)$. We call RP a* replacement policy.

$\mathsf{access}(\mathsf{RP}, x)$**:** *A function that takes as input a replacement policy RP and an address $x = (s,t) \in S \times T$. It modifies the cache $\mathtt{C}$ and outputs a* latency *bit h. The bit h indicates whether there has been a* cache hit *($t \in \mathtt{C}[s]$, in which case $h = 0$) or a* cache miss *($t \notin \mathtt{C}[s]$, then $h = 1$). This function proceeds as follows:*

1. *Compute the cache line $\ell = \mathsf{RP}(x)$.*
2. *If $t \in \mathtt{C}[s]$, output the bit 0.*
3. *If $t \notin \mathtt{C}[s]$, assign the tag $\mathtt{C}[s][\ell] = t$ and output the bit 1.*

When a slot $\mathtt{C}[s][\ell]$ in the cache has not yet been accessed, we denote $\mathtt{C}[s][\ell] = \mathtt{NULL}$. We say that a cache set $\mathtt{C}[s]$ is *empty* if $\mathtt{C}[s][\ell] = \mathtt{NULL}$ for every cache line

ℓ. Similarly, a cache $\mathtt{C}$ is *empty* if all its cache sets $\mathtt{C}[s]$ are empty. A cache set $\mathtt{C}[s]$ is said to be *full* if $\mathtt{C}[s][\ell] \neq \mathtt{NULL}$ for every cache line ℓ.

If $\mathtt{C}[s][\ell] = t$ and executing $\mathsf{access}(\mathsf{RP}, (s, t'))$ for $t' \neq t$ reassigns $\mathtt{C}[s][\ell] = t'$, we say that the access *evicts* the address (s, t) from the cache.

Definition 3. *A cache system* $(\mathit{C}, \mathsf{RP}, \mathit{access})$ *is*

- fully associative *if* $|\mathit{C}| = 1$ *(i.e., if it is over an index space satisfying* $|S| = 1$*),*
- a-associative *if, for every* $s \in S$*, the cache set* $\mathit{C}[s]$ *has size* a *(i.e.,* a *lines), and*
- directly mapped *if it is* 1*-associative.*

Regarding replacement policies, conventional replacement policies include the Random (RAND) policy, the Least-Recently Used (LRU) policy, Pseudo-LRU (PLRU), First-In-First-Out, or Most-Recently Used. As stated in other work [BDY+20,PGGV21], two main options to consider are RAND and LRU:

$\mathsf{RAND}((s, t))$**:** This policy selects a random cache line when there is a cache miss and the cache set $\mathtt{C}[s]$ is full. It proceeds as follows:
- If $t \in \mathtt{C}[s]$ (cache hit case), output the cache line ℓ such that $\mathtt{C}[s][\ell] = t$.
- If $t \notin \mathtt{C}[s]$ (cache miss case):
  - If $\mathtt{C}[s]$ is not full, output the smallest cache line ℓ so that $\mathtt{C}[s][l] = \mathtt{NULL}$.
  - If $\mathtt{C}[s]$ is full, output a uniformly random cache line ℓ.

$\mathsf{LRU}((s, t))$**:** If there is a cache miss and the cache set $\mathtt{C}[s]$ is full, this policy returns the cache line of $\mathtt{C}[s]$ that has not been accessed for the longest time. For this, it logs every access in its internal state $\mathrm{st}_{\mathsf{LRU}}$. It proceeds as follows:
- If $t \in \mathtt{C}[s]$ (cache hit case), let ℓ be the cache line such that $\mathtt{C}[s][\ell] = t$. Log (s, ℓ) into $\mathrm{st}_{\mathsf{LRU}}$, and output ℓ.
- If $t \notin \mathtt{C}[s]$ (cache miss case):
  - If $\mathtt{C}[s]$ is not full, let ℓ be the smallest cache line so that $\mathtt{C}[s][\ell] = \mathtt{NULL}$. Log (s, ℓ) into $\mathrm{st}_{\mathsf{LRU}}$, and output ℓ.
  - If $\mathtt{C}[s]$ is full, retrieve from $\mathrm{st}_{\mathsf{LRU}}$ the line ℓ of this cache set that has not been accessed for the longest time. Log (s, ℓ) into $\mathrm{st}_{\mathsf{LRU}}$, and output ℓ.

One common property of these replacement policies is that cache sets that are not full are accessed sequentially. The results of this work assume that all replacement policies behave in this way.

From now on, when it is irrelevant or clear from context, we drop all reference to the replacement policy RP. Particularly, we do so in the declaration of cache systems and in the calls to the access function. Unless explicitly stated, we also assume that the set and tag spaces S, T are fixed and taken as input by all algorithms.

2.4 Cache randomizer and RPC models

To prevent access-based attacks to cache systems, previous work considers RPCs. Their randomization-based approach consists in restricting the access functionality of caches, so that all addresses pass first through a randomization function π that computes the actual cache set to access. In other words, given a cache system $(\mathsf{C}, \mathsf{access})$, cache accesses in RPCs are made with the function access_π defined as $\mathsf{access}_\pi((s,t)) := \mathsf{access}((\pi(s,t),t))$.

As we stated in the introduction, π should be a keyed algorithm. Moreover, we may need some pseudo-random property out of it for security (see Definition 9). For reasons also stated above, the keying material should be periodically set up using a rekeying function rekey, which could work as a key derivation function that derives new keys from previously generated keying material and randomness. The following definition formalizes these concepts.

Definition 4 (Cache Randomizer). *A* cache randomizer *is a tuple* (rekey, π) *that consists of the following:*

$\mathsf{rekey}()$**:** *A probabilistic, possibly stateful, algorithm that generates a key k. We refer to* rekey *as the* rekeying algorithm*.*

$\pi_k(x)$**:** *A probabilistic algorithm that takes as input a key k and an address $x \in S \times T$, and outputs a set index $\pi_k(x) \in S$. We refer to π_k as the* randomization function*.*

Given a cache randomizer (rekey, π) and a key k, two addresses x, y are called *congruent* if $\pi_k(x) = \pi_k(y)$.

To the end of deriving results under the assumption of the existence of ideal-case cache randomizers, we define an ideal cache randomizer as one whose randomization function behaves as an independent random oracle for every key k.

Definition 5 (Ideal Cache Randomizer).
An ideal cache randomizer *is a cache randomizer* $(\overline{\mathsf{rekey}}, \bar{\pi})$ *such that:*

1. *The* $\mathsf{rekey}()$ *algorithm samples k uniformly at random from a nonempty key space $\mathcal{K}$.*
2. *For every sampled key k, the function $\bar{\pi}_k$ is chosen uniformly at random amongst all functions from $S \times T$ to S.*

We next define the central object of study of this article.

Definition 6 (Randomization-based Protected Caches). *A* randomization-based protected cache *(or simply* RPC*) is a concatenation* $\mathcal{C} = (\mathcal{C}, \mathit{access}, \mathsf{rekey}, \pi)$ *of a cache system and a cache randomizer.*

3 Security definitions

In this section, we present our security definitions for RPCs, and we state the pseudo-randomness notions associated to cache randomizers and rekeying algorithms. This work studies the security of RPCs from a standard cryptographic viewpoint. We define a game where an attacker and a challenger interact through an RPC, and we characterize security using the outcome of the game.

All the adversaries and distinguishers involved in our security and pseudo-randomness definitions are considered to run in finite time and to have finite resources. In particular, we do not consider computationally unbounded adversaries.

In Sections 3.1 and 3.2, we present our single-epoch and multi-epoch security definitions for RPCs. Then, in Sections 3.3 and 3.4, we formalize the pseudo-randomness of cache randomization functions and of rekeying algorithms, which are the properties we need to achieve single-epoch and multi-epoch RPC security, respectively.

3.1 Single-epoch security of RPCs

In the following, we define the single-epoch security of RPCs against single-target access-based attacks. In these attacks, there is a distinguished target address $x = (\sigma, \tau) \in S \times T$. The attacker is allowed to use the access_{π_k} function on any address that does not have the same tag as x to modify the initially empty cache $\mathtt{C}$, obtaining the corresponding latency measures. The objective of the attacker is to use these latencies to learn whether or not an access to x has been carried out. Access-based attacks covered by our security definition include Prime+Probe [OST06] and Evict+Probe [BM06].

In the considered threat model, we assume that the attacker has full knowledge of x, as well as of the public parameters $(S, T, \mathsf{RP}, \mathsf{access}, \mathsf{rekey}, \pi)$. However, we disregard attacks that rely on additional attacker privileges such as cache flushing, cache collisions with victim processes, or timing the execution of victim processes [DXS20]. For a list of the assumptions that configure the threat model of our security definition, see Appendix B.

Our security definition is stated in terms of an experiment $(N_1, N_2)\text{-}\mathsf{Rand}_{\mathcal{A}}(x)$, in which a stateful adversary $\mathcal{A}$ interacts with a challenger through an RPC. This experiment takes as input a target address x and two positive integers N_1, N_2.

Initially, the cache $\mathtt{C}$ is empty, and a key k is instantiated through a call to rekey. Then, $\mathcal{A}$ performs N_1 accesses to addresses of choice, so as to prepare the cache. The address choices are made adaptively, taking into account all previous addresses and latency measures, which are seen by $\mathcal{A}$ and logged into its state $\mathrm{st}_{\mathcal{A}}$. Afterwards, an access to x is made with probability $1/2$ according to a fair coin b. Subsequently, N_2 addresses are adaptively chosen and accessed by $\mathcal{A}$. In a final guess phase, the target address x, and the state $\mathrm{st}_{\mathcal{A}}$ that contains all previous addresses and latencies, are used by $\mathcal{A}$ to make a guess b' of the bit b. The experiment outputs the bit b and the adversarial guess b'.

For the sake of clarity, in our security definition we denote our stateful adversary by $\mathcal{A} = (\mathcal{A}_{1,1}, \ldots, \mathcal{A}_{1,N_1}, \mathcal{A}_{2,1}, \ldots, \mathcal{A}_{2,N_2}, \mathcal{A}_3)$, where each component stands for a step in the described experiment. Moreover, for readability, we encapsulate into an algorithm called Prime the process of choosing an address, accessing it, and logging the accessed address and latency into the state $\mathrm{st}_{\mathcal{A}}$ of the adversary $\mathcal{A}$.

The next definition states the security of an RPC $\mathcal{C}$ in one epoch, in terms of the experiment Rand. It declares that $\mathcal{C}$ is secure when any adversary $\mathcal{A}$ can not guess the value of the bit b from the interaction transcript with more than a certain advantage.

Definition 7 ((N_1, N_2)-Access and N-Access Security). *Let $\mathcal{C}$ be an RPC, $\mathcal{C} = (C, \mathsf{access}, \mathsf{rekey}, \pi)$. Let N_1, N_2 be positive integers, and let $x = (\sigma, \tau) \in S \times T$ denote a target address. And let $\mathcal{A} = (\mathcal{A}_{1,1}, \ldots, \mathcal{A}_{1,N_1}, \mathcal{A}_{2,1}, \ldots, \mathcal{A}_{2,N_2}, \mathcal{A}_3)$ denote a probabilistic stateful adversary, where every $\mathcal{A}_{i,j}$ outputs in $S \times (T \backslash \{\tau\})$. Define the experiment (N_1, N_2)-$\mathsf{Rand}_{\mathcal{A}}(x)$ as follows:*

$\underline{(N_1, N_2)\text{-}\mathsf{Rand}_{\mathcal{A}}(x):}$

- *for* $s \in S: \ C[s] \leftarrow (\mathit{NULL})_j$
- $k \leftarrow \mathsf{rekey}()$
- *for* $1 \leq i \leq N_1$:
  - $\mathsf{Prime}_{\mathcal{A}_{1,i}}(x, k)$
- $b \xleftarrow{\$} \{0, 1\}$
- *if* $b = 1: \ 1 \leftarrow \mathsf{access}_{\pi_k}(x)$
- *for* $1 \leq i \leq N_2$:
  - $\mathsf{Prime}_{\mathcal{A}_{2,i}}(x, k)$
- $b' \leftarrow \mathcal{A}_3(x)$
- *output* (b, b')

$\underline{\mathsf{Prime}_{\mathcal{A}_{j,i}}(x, k):}$

- $x_{j,i} \leftarrow \mathcal{A}_{j,i}(x)$
- $h_{j,i} \leftarrow \mathsf{access}_{\pi_k}(x_{j,i})$
- $\mathrm{st}_{\mathcal{A}} \leftarrow x_{j,i}, h_{j,i}$

We say the RPC $\mathcal{C}$ is (N_1, N_2)-access secure with advantage at most p *if, for every target address x and for every probabilistic stateful adversary $\mathcal{A}$, the output (b, b') of (N_1, N_2)-$\mathsf{Rand}_{\mathcal{A}}(x)$ satisfies*

$$\mathrm{Adv}^{\mathrm{RPC}}_{\mathcal{C},\mathcal{A}}(N_1, N_2) := 2 \cdot |\Pr[b' = b] - 1/2| \leq p,$$

where the probability is taken over the randomness of the adversary and the randomness used in the RPC $\mathcal{C}$ and the experiment.

We say that $\mathcal{C}$ is N-access secure with advantage at most p *if it is $(i, N - i)$-access secure with advantage at most p for all $i = 0, \ldots, N$.*

In this definition, an adversary has advantage at most p if and only if it distinguishes whether the challenge access to x has been carried out or not (that is, the value of the bit b) with probability at most p. Hence, having advantage 0 is equivalent to being oblivious to the value of b, and having advantage 1 implies guessing b (or its complementary) almost perfectly.

3.2 Multi-epoch security of RPCs

In practice, an RPC should mask cache accesses for at least as long as critical or attacker processes can be executed, hence possibly during various epochs. However, as pointed out in previous work [BDY+20], leakage through access-based attacks can carry across epochs, as information about the underlying cache randomizer is revealed even if the key changes. This information could become useful for attackers across different epochs. This calls for a multi-epoch security definition, and for defining security properties of cache randomizers that could help enforce security across epochs.

We next extend Definition 7 to characterize the security of RPCs across $R > 0$ epochs.

Definition 8 (R-Epoch Security). *Let $\mathcal{C} = (\mathcal{C}, \mathsf{access}, \mathsf{rekey}, \pi)$ be an RPC. Let R be a positive integer, let $(N_{1,i})_{i=1}^{R}, (N_{2,i})_{i=1}^{R}$ be sequences positive integers, and let $x = (s,t) \in S \times T$ denote a target address. Let $\mathcal{A} = (\mathcal{A}_1, \ldots, \mathcal{A}_R, \mathcal{A}')_{i=1}^{R}$ be a probabilistic stateful adversary, where*

$$\mathcal{A}_i = (\mathcal{A}_{i,1,1}, \ldots, \mathcal{A}_{i,1,N_{1,i}}, \mathcal{A}_{i,2,1}, \ldots, \mathcal{A}_{i,2,N_{2,i}}),$$

and where every $\mathcal{A}_{i,j,k}$ outputs in $S \times (T \setminus \{\tau\})$.

Define the experiment $(R, (N_{1,i})_i, (N_{2,i})_i)\text{-}\mathsf{Rand}_{\mathcal{A}}(x)$ as follows:

$$\begin{array}{l} \underline{(R, (N_{1,i})_i, (N_{2,i})_i)\text{-}\mathsf{Rand}_{\mathcal{A}}(x):} \\ \quad b \xleftarrow{\$} \{0,1\} \\ \quad \textit{for } 1 \le i \le R: \\ \quad\quad (N_{1,i}, N_{2,i})\text{-}\mathsf{Rand}^{b}_{\mathcal{A}_i}(x) \\ \quad b' \leftarrow \mathcal{A}'(x) \\ \quad \textit{output } (b, b') \end{array}$$

where $\mathsf{Rand}^{b}_{\mathcal{A}_i}(x)$ denotes an experiment similar to the $\mathsf{Rand}_{\mathcal{A}_i}(x)$ experiment in Definition 7, except for the fact that the challenge bit b is fixed in advance and that there is no adversarial guess phase or output.

*We say the RPC $\mathcal{C}$ is R-*Epoch $((N_{1,i})_i, (N_{2,i})_i)$-Access Secure with advantage at most p *if, for every probabilistic stateful adversary $\mathcal{A}$, the output (b, b') of the experiment $(R, (N_{1,i})_i, (N_{2,i})_i)\text{-}\mathsf{Rand}_{\mathcal{A}}(x)$ satisfies*

$$\mathrm{Adv}^{\mathrm{ME-RPC}}_{\mathcal{C},\mathcal{A}}(R, (N_{1,i})_i, (N_{2,i})_i)) := 2\left|\Pr\left[b' = b\right] - \frac{1}{2}\right| \le p,$$

where the probability is taken over the randomness of the adversary and the randomness used in the RPC $\mathcal{C}$ and the experiment.

*We say $\mathcal{C}$ is R-*Epoch N-Access Secure with advantage at most p *if it is R-Epoch $((N_i)_i, (N - N_i)_i)$-Access Secure with advantage at most p for all $N_1, \ldots, N_n \in \{0, \ldots, N\}$.*

3.3 Pseudo-randomness of the randomization function

As for the impact of the cache randomizer on security as characterized in Definitions 7 and 8, we can consider the following cases, depending on the nature of $\pi_k : S \times T \to S$:

Unkeyed and deterministic: In this case, the attacker knows the cache set $\pi(s,t)$ every address (s,t) is assigned to. For instance, if $\mathsf{RP} = \mathsf{LRU}$ and if the cache is a-associative, a successful attacker can consider a different addresses $(s,t) \neq x$ with different tags and $\pi(s,t) = \pi(x)$, and access them sequentially before and after the challenge. The bit b equals one when, in the second accesses, some latency bit takes the value 1. Hence, security is broken in at most $2a$ adversarial accesses.

Keyed and deterministic, without rekeying: In a keyed version, the cache randomizer makes the previous task more difficult by hiding the congruence relation between addresses. However, given enough accesses, an adversary can use latencies to unveil this relation, and build various sets of at most a congruent addresses. By building enough such sets and accessing them before and after the challenge, it is possible to detect an access to x with high advantage. This claim is backed up by existing research [BDY+20,PGGV21], where such attacks are instantiated.

Keyed and deterministic, with rekeying: Setting a new key periodically is seen by previous work [BDY+20,PGGV21] as the most effective way of enforcing the security of RPCs, despite the impact on efficiency. However, knowing which rekeying periods are able to effectively hamper attacks is not trivial, and leakage across epochs must be taken into account too. We address these questions in Section 4.

A fourth option is keyed and probabilistic cache randomization, which has been realized in [Qur19,WUG+19] through the use of skewed caches [Sez93,SSW14]. While our security definition for RPCs applies to such cache randomization methods, in this work we do not consider them.

We next define a notion of pseudo-randomness for cache randomizers. This notion coincides with the standard definition of pseudo-random functions (PRFs) that can be found in [BKR98,AB00].

Definition 9 ((ν, ε)-Pseudo-Random Randomization Function). *We say that a cache randomizer* (rekey, π) *is* (ν, ε)-pseudo-random *if every distinguisher A, which is allowed ν oracle accesses to a given function, has advantage at most ε in distinguishing the oracle cache randomizer from random. That is,*

$$\mathrm{Adv}^{\mathrm{PRF}}_{(\mathsf{rekey},\pi),A}(\nu) := \left|\Pr_{k \leftarrow \mathsf{rekey}()}[A^{\pi_k}() = 1] - \Pr_{g \overset{\$}{\leftarrow} \mathcal{F}}[A^{g}() = 1]\right| \leq \varepsilon,$$

where $\mathcal{F}$ denotes the set of functions from $S \times T$ to S.

An option to instantiate a PRF with concrete security parameters in practice is given by the reduction proposed by Goldreich et al. [GGM86]. In this way, a

(ν, ε)-pseudo-random cache randomizer can be efficiently built from a $(\alpha \cdot \nu, \epsilon/\alpha)$-pseudo-random number generator (PRNG) with an α time factor increase, where $\alpha = \log|S \times T|$. The security parameters of a PRNG can be heuristically inferred from a wide array of randomness tests, such as [BRS+10]. Pseudo-random functions can also be instantiated from block ciphers [BKR98,HWKS98].

Note that the ideal cache randomizer $(\overline{\mathsf{rekey}}, \overline{\pi})$ is trivially $(\nu, 0)$-pseudo-random for every $\nu \geq 0$.

3.4 Pseudo-randomness of the rekeying algorithm

In order to extend single-epoch security to various epochs, some property must be required from the rekeying process. As shown in [AB00] for IND-CPA encryption, this property can be the pseudo-randomness of key generation. As their approach translates directly to RPCs, we adopt their definition of pseudo-randomness of stateful generators for rekeying.

Definition 10 ((ν, ε)-Pseudo-Random Rekeying Algorithm). *Let* rekey *be a stateful probabilistic algorithm with no input and outputs in a set* $\mathcal{K}$*. We say that* rekey *is* (ν, ε)-pseudo-random *if every distinguisher* A*, which is allowed* ν *oracle calls to a given algorithm, has advantage at most* ε *in distinguishing the oracle from random. That is,*

$$\mathrm{Adv}^{\mathrm{PRG}}_{\mathsf{rekey},A}(\nu) := \left|\Pr[A^{\mathsf{rekey}}() = 1] - \Pr[A^{K}() = 1]\right| \leq \varepsilon,$$

where K *returns* $k \xleftarrow{\$} \mathcal{K}$ *on each call.*

We refer to [AB00] for a procedure for building pseudo-random rekeying algorithms from pseudo-random functions.

4 Security analysis

In this section, we study the conditions so that an RPC can provide concrete security guarantees. More precisely, given a target advantage p, we show rekeying periods N for which an RPC is N-access secure with at most advantage p.

In a first stage, we assume the cache randomizer of the RPC in question is ideal. In the next section, we extend the obtained results to the non-ideal case, modelling the cache randomizer as a pseudo-random function. Then, we study how the given results improve under the presence of noise. Finally, we take advantage of previous research to allow the study of the multi-epoch case. We complete our security analysis by studying which rekeying periods are known to allow efficient attacks.

Most of the results presented in this section are formulated using the binomial distribution. For ease of notation, we respectively denote by $f(k; n, p)$ and $F(k; n, p)$ the probability mass function and the cumulative distribution function of the binomial random variable with parameters n and p. That is,

$$f(k; n, p) := \binom{n}{k} p^k (1-p)^{n-k} \qquad F(k; n, p) := \sum_{i=0}^{k} f(i; n, p).$$

4.1 Single-epoch security of RPCs with ideal cache randomizers

The following proposition provides rekeying periods N for which an RPC with an ideal cache randomizer is N-access secure with at most a certain advantage, as stated in Definition 7.

Proposition 1. *Let $\mathcal{C}$ be an RPC with an ideal cache randomizer and an arbitrary replacement policy. Let $|S|$ denote the number of cache sets, and suppose that the cache is a-associative. Let $p \in [0,1]$, and*

$$N = \max\left\{N' \ : \ F\left(N'-a; N', 1-1/|S|\right) \leq p\right\}.$$

Then, $\mathcal{C}$ is N-access secure with advantage at most p.

Proof. For any adversary $\mathcal{A}$ in the N-access security games, let E denote the event that $\mathcal{A}$ accesses at least a different addresses that are congruent with the target address x. By conditional probability

$$\begin{aligned}\mathrm{Adv}^{\mathrm{RPC}}_{\mathcal{C},\mathcal{A}}(N) =& 2 \cdot \left|\Pr\left[b'=b\right] - 1/2\right| \\ =& 2 \cdot \left|\Pr\left[b'=b \mid E\right] \cdot \Pr\left[E\right] + \Pr\left[b'=b \mid \bar{E}\right] \cdot \Pr\left[\bar{E}\right] - 1/2\right|.\end{aligned}$$

When using arbitrary replacement policies, it holds that the first a addresses of an initially empty cache set are inserted sequentially. Therefore, any successful attack must find at least a addresses that are congruent with x; otherwise, latency measures are independent of b. Hence $\Pr\left[b'=b \mid \bar{E}\right] = 1/2$, and so

$$\begin{aligned}&= 2 \cdot \left|\Pr\left[b'=b \mid E\right]\Pr\left[E\right] + 1/2 \cdot \Pr\left[\bar{E}\right] - 1/2\right| \\ &= 2 \cdot \left|\left(\Pr\left[b'=b \mid E\right] - 1/2\right) \cdot \Pr\left[E\right]\right| \leq \Pr\left[E\right].\end{aligned}$$

Now, we see that E follows a binomial distribution, and that $\Pr\left[E\right]$ is the probability of obtaining at least a successes in a sequence of at most N independent experiments, where successes happen with probability $1/|S|$. Hence

$$\mathrm{Adv}^{\mathrm{RPC}}_{\mathcal{C},\mathcal{A}}(N) \leq F\left(N-a; N, 1-1/|S|\right) \leq p.$$

In Figure 5 we show the rekeying period given by Proposition 1 in the ideal cache randomizer case. We arbitrarily take the advantage bound $p = 0.01$, and so these rekeying periods N guarantee that no adversary breaks N-access security with more than 1% advantage. We represent the bounds in logarithmic scale, for the typical numbers of cache sets $|S| = 2^5, 2^6, \ldots, 2^{14}$ and associativities $a = 1, 2, 4, 8, 16$.

4.2 Single-epoch security of RPCs with PRF cache randomizers

The next proposition states sufficient conditions on the cache randomizer (rekey, π) so that any RPC using π is N-access secure with a certain advantage. This proposition allows us to choose cache randomizers when aiming at a particular security level.

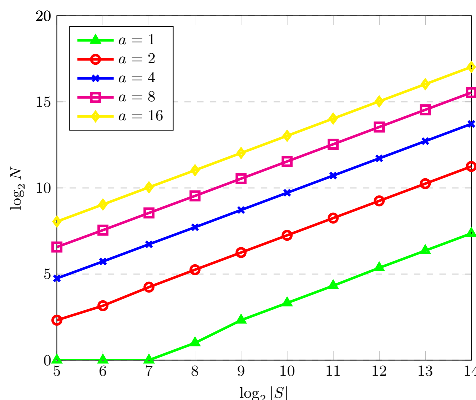

Fig. 5: Rekeying periods N so that $\mathcal{C}$ is N-access secure with advantage at most 0.01, where $|S|$ denotes the number of cache sets and a the associativity.

Proposition 2. *Let N, θ be positive integers, and let $p, \varepsilon \in [0, 1]$. Let (rekey, π) be a cache randomizer, and let $(\overline{\mathsf{rekey}}, \bar{\pi})$ be the ideal cache randomizer. Let $(\mathcal{C}, \bar{\mathcal{C}})$ be identical RPCs except for their cache randomizers, which are (rekey, π) and $(\overline{\mathsf{rekey}}, \bar{\pi})$, respectively.*

Suppose that $\bar{\mathcal{C}}$ is N-access secure with advantage at most p against θ-time adversaries, and that the randomization function π is (N, ε)-pseudo-random against θ-time distinguishers. Then $\mathcal{C}$ is N-access secure with advantage at most $p + \varepsilon$ against θ-time adversaries.

Proof. Build a θ-time distinguisher A in the definition of (N, ε)-pseudo-random cache randomizer as follows. The distinguisher A receives as input a randomization function f and internally runs an N-access security experiment with f, simulating an arbitrary θ-time adversary $\mathcal{A}$. If the outcome is $b' = b$, then A outputs 1, and otherwise it outputs 0. Since π is (N, ε)-pseudo-random,

$$\begin{aligned} &\left| \Pr_{k \leftarrow \mathsf{rekey}()}[A^{\pi_k}() = 1] - \Pr_{f \xleftarrow{\$} \mathcal{F}}[A^f() = 1] \right| \\ &= |\Pr[b = b' \mid f = \pi_k] - \Pr[b = b' \mid f = \bar{\pi}_k]| \leq \varepsilon \end{aligned}$$

so $\Pr[b = b' \mid f = \pi_k] \leq \Pr[b = b' \mid f = \bar{\pi}_k] + \varepsilon$, and $\mathrm{Adv}^{\mathrm{RPC}}_{\mathcal{C},\mathcal{A}}(N) \leq \mathrm{Adv}^{\mathrm{RPC}}_{\bar{\mathcal{C}},\mathcal{A}}(N) + \varepsilon$.

4.3 Single-epoch security of RPCs under noise conditions

The noiseless setting explored in Section 4.1 may be the most conservative, and this assumption may make sense in contexts such as trusted execution environments [CKK+20]. In general, other system activity may introduce noise and add confusion to access-based attacks, on top of reducing the number of available attack accesses in an epoch.

To take into account the effect of noise on security, we can model noise as suggested in [PGGV21]. Concretely, we ideally assume that a ratio $\rho \in [0, 1)$ of the N accesses in the security game are noise accesses, so there are ρN noise accesses (rounded down) and $(1-\rho)N$ adversarial accesses. We also assume that noise adds maximum confusion to the system by always making cache misses. Finally, we assume that noise accesses happen in a batch, with no adversarial accesses in-between, and only in successive steps around the challenge.

To properly analyze this setting, we can envision a security definition similar to Definition 7, where ρN successive calls to $\mathsf{Prime}_{\mathcal{A}_{j,i}}$ that are contiguous to the challenge are replaced by noise accesses. We call the corresponding security definition *N-access security under noise level ρ.*

Following our conservative approach to security, we consider that noise makes it impossible for the adversary to detect the challenge bit only when the target address evicts or is evicted by noise addresses. The effects of noise could be stronger for more restricted adversaries, as happens in [PGGV21]. For instance, our results would greatly improve if we considered that attacks always fail when at least one noise access is made to the cache set of the target address.

In this noise setting, we make an additional assumption on the replacement policy. We assume that a consecutive cache misses on a cache set evict all data from it. This is indeed the case for LRU, PLRU, FIFO, and other policies. Our results can be easily adapted to policies like Nehalem's MRU [AR20], where $2a$ consecutive cache misses may instead be needed to evict a whole cache set.

The following proposition extends the result of Proposition 1, providing rekeying periods N for which an RPC with an ideal cache randomizer is N-access secure under noise level ρ with at most a certain advantage. The extension to PRF cache randomizers follows from Proposition 2, analogously to the noiseless case.

Proposition 3. *Let $\mathcal{C}$ be an RPC with an ideal cache randomizer and a replacement policy of the form outlined above. Let $|S|$ denote the number of cache sets, and suppose that the cache is a-associative. Let $p \in [0, 1]$, and*

$$N = \max\left\{ N' \ : \ \sum_{i=0}^{a-1} \sum_{j=a-i}^{(1-\rho)N'} f(i; \rho N', 1/|S|) \cdot f(j; (1-\rho)N', 1/|S|) \leq p \right\}.$$

Then, $\mathcal{C}$ is N-access secure under noise level ρ with advantage at most p.

Proof. For any adversary $\mathcal{A}$ in the security game, let E denote the event that the cache set of the target address is either accessed at least a times by noise accesses, or otherwise simply not filled during the game. If E happens, then $\mathcal{A}$ has no information on the challenge b, and so, as in the proof of Proposition 1, $\mathrm{Adv}^{\rho\text{-noise-RPC}}_{\mathcal{C},\mathcal{A}}(N) \leq \Pr(\bar{E})$.

Now, for $0 \leq i < \rho N$, let E_i be the event that, during the security game, i of the noise accesses take place in the cache set of the target address. The result follows by conditioning on the events E_i, noting that $\Pr(E_i) = f(i; \rho N, 1/|S|)$,

and that

$$\Pr(\bar{E}|E_i) \leq \begin{cases} \sum_{j=a-i}^{(1-\rho)N} f(j; (1-\rho)N, 1/|S|) & \text{if } i < a \\ 0 & \text{if } i \geq a. \end{cases}$$

4.4 Multi-epoch security of RPCs

Following an argument of Abdalla and Bellare [AB00], we can find an upper bound for the advantage of any finite-time adversary in breaking the multi-epoch security of an RPC. In this way, multi-epoch security is reduced to single-epoch security and the pseudo-randomness of the rekeying algorithm. As explained in [AB00], rekeying can have the effect of bringing significant and provable security gains, and in our case it can increase the time window where security is known to be enforced (see Section 5).

Proposition 4 ([AB00]). *Let R, N, θ be positive integers. Suppose an RPC $\mathcal{C}$ is N-access secure with advantage at most $\mathrm{Adv}_{\mathcal{C}}^{\mathrm{RPC}}(N)$ against θ-time adversaries, and that its underlying rekeying algorithm* rekey *is $(R, \mathrm{Adv}_{\mathsf{rekey}}^{\mathrm{PRG}}(R))$-pseudo-random against θ-time distinguishers. Then, $\mathcal{C}$ is R-epoch N-access secure with advantage at most*

$$\mathrm{Adv}_{\mathcal{C}}^{\mathrm{ME-RPC}}(R, N) \leq \mathrm{Adv}_{\mathsf{rekey}}^{\mathrm{PRG}}(R) + R \cdot \mathrm{Adv}_{\mathcal{C}}^{\mathrm{RPC}}(N)$$

against θ-time adversaries.

4.5 Insecure rekeying periods

In this section, we study methods to obtain rekeying periods for which attacks with enough advantage are known to exist. We do so in two separate instances, corresponding to the cases where the used replacement policy RP is either LRU or RAND.

The RP = LRU case We next propose an attack to the security game of Definition 7, for the case RP = LRU. This attack follows the same strategy than [PGGV21]. Our attack relies on the following fact: if a set L of exactly a addresses with different tags that are congruent to the target address x are accessed right before and after the challenge takes place, the second batch of accesses to L will all have latency 0 if and only if x was not accessed. Moreover, after this process, the addresses of L will all end up being stored in the cache set regardless of whether x was accessed or not. We next describe our attack.

In the first N_1 accesses, the adversary $\mathcal{A}_{1,*}$ builds a set of addresses with no auto-evictions from the ground up. We call this set of addresses an *attack set*. It is expected that, when the attack set is large enough, accessing it has a high probability of evicting the target address $x = (\sigma, \tau)$ from the cache. By construction, $\mathcal{A}_{1,*}$ ends by accessing all the addresses of the attack set. Then, the challenge takes place. In the last N_2 accesses, the adversary $\mathcal{A}_{2,*}$ outputs the addresses of the attack set, of which there are no more than N_2 by construction.

If some of the accesses given by $\mathcal{A}_{2,*}$ has latency 1, then $\mathcal{A}_3$ guesses that the target address x was accessed (i.e. $b' = 1$), and otherwise it determines that it has not ($b' = 0$). We next describe how the first N_1 steps of this attack work.

To build the attack set L, the adversary initially generates $s \in S$ and $t \in T \setminus \{\tau\}$ at random, logs the address (s,t) into an ordered array L in its state, and outputs (s,t). Then, it starts an iterative process, with a total of ℓ loops (counting one less for the previous access). In each step of the loop, the adversary first generates an address $(s,t) \in S \times T \setminus \{\tau\}$ uniformly at random, subject to the restriction that t is not the tag of any address in the attack set, and then it stores the address (s,t) in its state and outputs it. We call (s,t) an *attack candidate*. Then, it finishes the step of the loop by sequentially returning every address of L. If the accesses to the attack set L have all had latency 0, the attack candidate (s,t) is logged as a new element of L, since it has not been evicted in the previous loop. Once all loops are finished, the subsequent outputs of the adversary are random elements of the attack set L.

Note that the described attack is only feasible when the attacker can use at least λ different tags from the tag space T. We write our attack in Algorithm 1.

Algorithm 1 Our attack in the $\mathsf{RP} = \mathsf{LRU}$ case (first N_1 accesses).

```
Let i = 0
A_{1,i} generates (s,t) ∈ S × (T \ {τ}) uniformly at random
A_{1,i} logs the attack candidate (s,t) into L
A_{1,i} outputs (s,t); i = i + 1
for j = 2, ..., λ do
    A_{1,i} generates (s,t) ∈ S × (T \ {τ}) uniformly at random, with t not in L
    A_{1,i} outputs (s,t); i = i + 1
    for a in L do
        A_{1,i} outputs a; i = i + 1
    end for
    if the previous latencies of accesses to L are all 0 then
        A_{1,i} logs the latest attack candidate (s,t) into L
    end if
end for
while i ≤ N_1 do
    A_{1,i} outputs a random element of L; i = i + 1
end while
```

In this case, we fix the number of loops to

$$\lambda = \min\left\{N_2, \left\lfloor \frac{\sqrt{8N_1+1}-1}{2} \right\rfloor\right\}.$$

This choice guarantees that the attack set L obtained after the first N_1 accesses does not have more than N_2 elements, and so it can all be re-accessed after the challenge takes place. Also, since executing λ' loops consumes at most $\lambda'(\lambda'+1)/2$

accesses, our choice of λ guarantees that λ whole loops can be executed during the first N_1 accesses.

The next proposition determines the advantage of our attack in breaking (N_1, N_2)-Access Security as stated in Definition 7.

Proposition 5. *Let $\mathcal{C}$ be an RPC with an ideal cache randomizer and replacement policy* LRU. *Let $|S|$ denote the number of cache sets, and suppose that the cache is a-associative. Let $p \in [0,1]$, and let*

$$N = \min \left\{ N' \ : \ \max_{N_1+N_2=N'} \left\{ F\left(\lambda - a; \lambda, 1 - \frac{1}{|S|}\right) \right\} \geq p \right\}.$$

Then, the adversary $\mathcal{A}$ described above breaks the game of (N_1, N_2)-Access Security of Definition 7 with advantage at least p.

Proof. Consider arbitrary positive integers N', N_1, N_2 so that $N_1 + N_2 = N'$. For the adversary $\mathcal{A}$ stated above, let E be the event that the addresses of the attack set fill the cache set of x, i.e, that L has a addresses that are congruent with x. Then, as in the beginning of the proof of Proposition 1, we find $\mathrm{Adv}_{\mathcal{A}}(\mathcal{C}) = 2 \cdot |(\Pr[b = b' \mid E] - 1/2)| \cdot \Pr[E]$.

If E happens, then the attack perfectly detects the challenge access. Therefore, $\Pr[b = b' \mid E] = 1$, and so $\mathrm{Adv}_{\mathcal{A}}(\mathcal{C}) = \Pr[E]$. Since the chosen number of attack candidates λ does not exceed N_2 and can always be accessed in N_1 steps, E is the event that the attack candidates fill the cache set of x, i.e., that at least a of the λ attack candidates are congruent with x. Hence, $\mathcal{A}$ that has advantage

$$\mathrm{Adv}_{\mathcal{A}}(\mathcal{C}) = F\left(\lambda - a; \lambda, 1 - 1/|S|\right)$$

in the game of (N_1, N_2)-Access Security of Definition 7. The result follows.

In Figure 6 we show the rekeying period given by Proposition 5 in the ideal cache randomizer case. We arbitrarily take the advantage bound $p = 0.01$, and so these rekeying periods N guarantee that there exists an attack that breaks N-access security with at least 1% advantage. We represent the bounds in logarithmic scale, for the typical numbers of cache sets $|S| = 2^5, 2^6, \ldots, 2^{14}$ and associativities $a = 1, 2, 4, 8, 16$.

The RP = RAND case Our attack for the RP = RAND case is slightly different than that of the RP = LRU case. Consider, as in the previous attack, a set L of a addresses with different tags that fill a cache set, and an attack candidate y. It is indeed the case that, if all addresses of L are stored in the cache and then one accesses y and L again, the last accesses to L all have latency 0 if and only if y did not evict any of the addresses in the cache set. However, due to the behavior of RAND, if the access to y evicted some address of the cache set, after the last access to L it may be the case that y still resides in the cache set while some unknown address of L does not. This greatly complicates the task of detecting future evictions, rendering obsolete the method of building attack sets explained earlier.

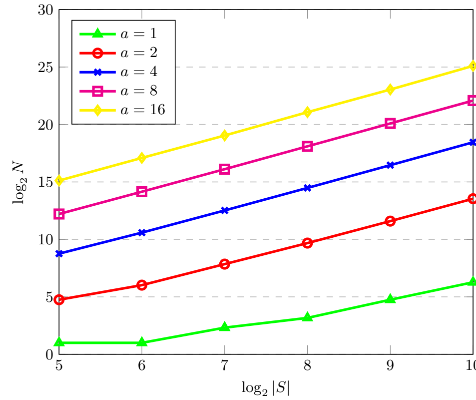

Fig. 6: Rekeying periods N so that there exists an attack on $\mathcal{C}$ that breaks N-access security with advantage at least 0.01, where $|S|$ denotes the number of cache sets and a the associativity.

To mitigate the problem above, after our adversary $\mathcal{A}_{1,*}$ accesses an attack candidate y, the addresses in L are accessed a total of m times, for some suitable value of $m \in \{1, \ldots, N_1 - 2\}$. Each access to L has probability at least $1/a$ of evicting y and, when it does, L ends up being stored in the cache. Moreover, if y is evicted in the first $m-1$ passes of L, the latencies in the m-th pass will all be 0. If for some of the loops this does not happen, then our attack fails.

The adversary $\mathcal{A}_{2,*}$ behaves as in the LRU case, accessing all elements of L. If in some loop of $\mathcal{A}_{1,*}$ the attack failed, then $\mathcal{A}_3$ returns a random coin. Otherwise, if some access issued by $\mathcal{A}_{2,*}$ has had latency 1, then $\mathcal{A}_3$ guesses that the challenge x was accessed (i.e. $b' = 1$), and else it determines that it has not ($b' = 0$).

Note that our attack requires $N \geq 5$ and $3 \leq N_1 \leq N - 2$, and that it is only feasible when the attacker can use at least λ different tags from the tag space T. We describe our attack in Algorithm 2.

As in the previous case, we fix the number of loops to

$$\lambda = \min\left\{N_2, \left\lfloor \frac{\sqrt{8mN_1 + (2-m)^2} + m - 2}{2m} \right\rfloor\right\}.$$

This choice guarantees that the attack set L obtained after the first N_1 accesses does not have more than N_2 elements, and so it can all be re-accessed after the challenge takes place. Also, since executing λ' loops consumes at most $\sum_{i=0}^{\lambda'-1}(im+1)$ accesses, our choice of λ guarantees that λ whole loops can be executed during the first N_1 accesses.

Proposition 6. *Let $\mathcal{C}$ be an RPC with an ideal cache randomizer and replacement policy* RAND. *Let $|S|$ denote the number of cache sets, and suppose that*

Algorithm 2 Our attack in the $\mathsf{RP} = \mathsf{RAND}$ case (first N_1 accesses).

```
Let i = 0
A_{1,i} generates (s,t) ∈ S × (T \ {τ}) uniformly at random
A_{1,i} logs the attack candidate (s,t) into L
A_{1,i} outputs (s,t); i = i + 1
for j = 2, ..., λ do
    A_{1,i} generates (s,t) ∈ S × (T \ {τ}) uniformly at random, with t not in L
    A_{1,i} outputs (s,t); i = i + 1
    for l = 1, ..., m do
        for a in L do
            A_{1,i} outputs a; i = i + 1
        end for
    end for
    if the latencies of accesses to L in the loop l = 1 are all 0 then
        A_{1,i} logs the latest attack candidate (s,t) into L
    else
        if the latencies of accesses to L in the loop l = m are not all 0 then
            A registers in its state that the attack fails
        end if
    end if
end for
while i ≤ N_1 do
    A_{1,i} outputs a random element of L; i = i + 1
end while
```

the cache is a-associative. Let $p \in [0,1]$, and

$$N = \min\left\{ N' \;:\; \max_{\substack{N_1+N_2=N' \\ 1\le m\le N_1-2}} \left\{ \left(1-(\lambda-a)\left(1-\frac{1}{a}\right)^{m-1}\right) \cdot F\left(\lambda-a;\lambda,1-\frac{1}{|S|}\right)\right\} \ge p \right\}.$$

Then, the adversary $\mathcal{A}$ described above breaks the game of (N_1,N_2)-Access Security of Definition 7 with advantage at least p.

Proof. Let N', N_1, N_2, m be positive integers with $N' = N_1 + N_2$ and $1 \le m \le N_1 - 2$. For the adversary $\mathcal{A}$ stated above, let H denote the event that the attack fails. If H happens, then $\mathcal{A}_3$ emits a uniformly random guess b', and so

$$\begin{aligned}
\mathrm{Adv}_{\mathcal{A}}(\mathcal{C}) :=& 2 \cdot |\Pr[b'=b] - 1/2| \\
=& 2 \cdot \left|\Pr[b'=b \mid H] \cdot \Pr[H] + \Pr\left[b'=b \mid \bar{H}\right] \cdot \Pr\left[\bar{H}\right] - 1/2\right| \\
=& 2 \cdot \Pr\left[\bar{H}\right] \cdot \left|\Pr\left[b'=b \mid \bar{H}\right] - 1/2\right|
\end{aligned}$$

Now let E denote the event that the addresses of the attack set fill the cache set of x, i.e, that L has a addresses that are congruent with x. On one hand, if E happens and the attack does not fail, our adversary perfectly detects the challenge access, i.e. $\Pr\left[b'=b \mid E \cap \bar{H}\right] = 1$. On the other hand, if E does not

happen and the attack does not fail, our adversary outputs $b' = 0$ regardless of the value of b, so $\Pr\left[b' = b \mid \bar{E} \cap \bar{H}\right] = 1/2$. Hence

$$= 2 \cdot \Pr\left[\bar{H}\right] \cdot \left|\Pr\left[b' = b \mid E \cap \bar{H}\right] \cdot \Pr\left[E|\bar{H}\right] + \Pr\left[b' = b \mid \bar{E} \cap \bar{H}\right] \cdot \Pr\left[\bar{E}|\bar{H}\right] - 1/2\right|$$
$$= \Pr\left[E|\bar{H}\right] \cdot \Pr\left[\bar{H}\right].$$

Now, $\Pr\left[E|\bar{H}\right]$ is the probability that at least a of the λ attack candidates are congruent with x. Hence $\Pr\left[E|\bar{H}\right] = F\left(\lambda - a; \lambda, 1 - 1/|S|\right)$. To lower bound $\Pr\left[\bar{H}\right]$, denote by H_j the event that our attack fails in its j-th loop. Note that our attack can not fail during the first a loops. By the union bound

$$\Pr[\bar{H}] = \Pr\left[\cap_{j=a+1}^{\lambda} \bar{H}_j\right] = 1 - \Pr\left[\cup_{j=a+1}^{\lambda} H_j\right] \geq 1 - \sum_{j=a+1}^{\lambda} \Pr\left[H_j\right].$$

Our attack fails in a given loop when the attack candidate corresponding to that loop evicts an element of the attack set from the cache, and the successive $m-1$ accesses to L fail to evict the attack candidate from the cache. Each access to L has probability at most $(a-1)/a$ of not evicting the attack candidate from the cache, and so in a given loop we get $\Pr[H_j] \leq \left(1 - \frac{1}{a}\right)^{m-1}$. The result follows.

5 Experimental results

In this section, we portrait how to apply the results of this article, as well as present a performance analysis using a cache simulator. The scripts used to compute the rekeying periods and number of epochs from Propositions 1, 3 and 4 have been made available at `https://doi.org/10.5281/zenodo.6397296`.

To present an application case, we choose the L3 cache of an Intel® Core i7-8700K (Coffee Lake) commodity processor [AR20]. This LLC is 12MB in size, it has associativity $a = 16$, a total of 12 slices, and 1024 cache sets per slice. The number of cache sets is then $|S| = 12 \cdot 1024 = 12288$.

5.1 Application case

Suppose we want to use an RPC to hamper access-based attacks with advantage larger than 1% in this LLC in the noiseless scenario. According to Proposition 1, assuming an ideal cache randomizer, the corresponding RPC is 100532-access secure with advantage at most 0.01. This means that a rekeying period of $N = 100532$ prevents single-epoch attacks with advantage larger than $p = 0.01$.

In the noise scenario, the rekeying period can slightly increase. In our results, this effect is only noticeable for large noise levels ρ. For example, according to Proposition 3 in order to guarantee that no attack has at most 1% advantage, our rekeying period rises from $N = 100532$ in the noiseless setting to $N = 106963$ for noise level $\rho = 0.9$. Or, in the case we want to guarantee no attack succeeds with more than 10% advantage, our rekeying period rises from $N = 136832$ in the noiseless setting to $N = 159006$ for noise level $\rho = 0.9$.

The ideal cache randomizer assumption can be lifted by virtue of Proposition 2. For example, if we take as cache randomizer a pseudo-random function that is $(100532, 0.04)$-pseudo-random (according to Definition 9), the corresponding RPC is 100532-access secure with advantage at most $p = 0.05$. That is, a rekeying period of $N = 100532$ guarantees that any single-epoch attack has at most 5% advantage. Similarly, this RPC is 106963-access secure under noise level $\rho = 0.9$ with advantage at most $p = 0.05$.

To illustrate the multi-epoch case, take the number of epochs $R = 10$, and assume that the used rekeying algorithm rekey is $(10, 10^{-5})$-pseudo-random. If the employed cache randomizer is $(78705, 4 \cdot 10^{-3})$-pseudo-random, then the considered RPC is 78705-access secure with advantage at most $4.999 \cdot 10^{-3}$. By Proposition 4, the considered RPC is 10-epoch 78705-access secure with advantage at most $10^{-5} + 10 \cdot (4.999 \cdot 10^{-3}) = 0.05$. Hence, security is extended to $R \cdot N = 787050$ accesses.

The pseudo-randomness of the underlying rekeying algorithm can have a big impact on security across epochs. For instance, assume instead that the used cache randomizer is $(63486, 4 \cdot 10^{-4})$-pseudo-random. Then, we can extend the same security guarantees to $R = 100$ epochs of $N = 63486$ accesses long. That is, to $R \cdot N = 6348600$ accesses.

5.2 Performance analysis

We have used Champsim, a trace-based simulator with a detailed cache model [Cha], to evaluate the impact of the rekeying period on the CPU's performance. We use this simulator to model the LLC of the Intel® Core i7-8700K (Coffee Lake) processor. Concretely, we model a cache with $1024 \cdot 16$ sets, 16 ways, and a cache set size of 64 bytes. In our simulation we choose the `PLRU` replacement policy, which is common in LLCs. In our cache setup, as in the modeled processor, the L1 and L2 private caches have 8 ways, and 64 and 1024 cache sets, respectively.

As workload, we use traces of the SPEC2006 bechmark suite. To model randomized caches, we use an xor-based parametric randomization function [THAC18], and vary the encription key every time we reach the number of accesses to the LLC dictated by the rekeying period.

Figure 7 shows the instructions per cycle (IPC) of each workload for a randomized cache setup, normalized to a cache system without randomization. To measure the impact of the rekeying period, we show results for rekeying periods ranging from just 100 acceses to 160000 accesses. As shown in Figure 7, the impact of rekeying very frequently can be severe for some workloads. For instance, for the `mcf` workload, performance is slowed down by 37% with a rekey period of 100 accesses, and only by 13.9% when using a rekeying period of 160000 accesses. On the contrary, other workloads like `perlbench` or `exchange2` have a negligible impact in performance. Interestingly, results for `fotonik3D` are better with randomization, and benefit from the cache remapping ability to remove pathological conflict misses.

In general, the impact of rekeying is not very significant when using large rekeying periods. In average, the performance is slowed down by 4.80%, 4.00%,

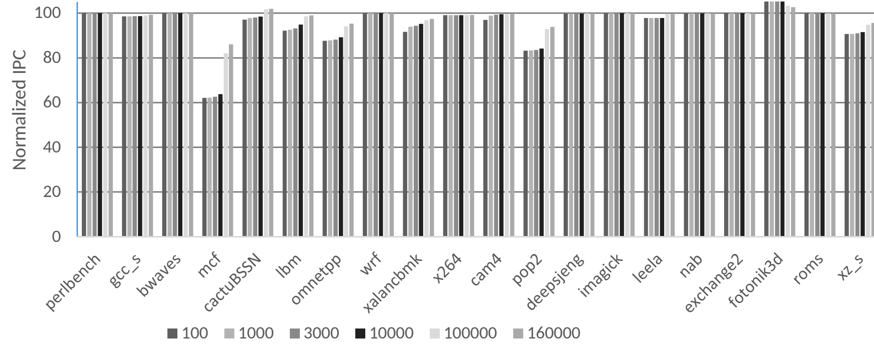

Fig. 7: Instructions per Cycle (IPC), normalized to a cache system without randomization for different rekeying periods.

1.90%, and $1,5\%$ for rekeying periods of 100, 1000, 100000, and 160000, respectively.

6 Conclusions and further work

In this paper, we formally define and analyze the security of randomization-based protected caches (RPCs). We provide a security definition for RPCs, called N-access security, that protects against access-based attacks to a single target address in one epoch in the setting where a rekeying period N is set. Additionally, we provide results that allow to calibrate the rekeying period N, the cache randomizer, and the rekeying algorithm in order to obtain specific security guarantees. Hence, the present work shows that it is possible to formalize security for RPCs, and that doing so allows for robust constructions and security claims. The obtained results are also extended to a scenario with noise conditions, and we give tools to quantify security across multiple epochs. Our approach improves on previous work by precisely characterizing security, and by providing formal proofs of the security of RPCs for chosen parameters.

We next highlight a few suggestions for further research in the line of this article.

A first extension of this work would be to consider RPCs that work over skewed caches. Skewed RPCs [Qur19,WUG$^+$19] use skewed caches [SSW14,Sez93] to introduce additional true randomness in the access function, and this is seen to substantially improve security against known attacks [PGGV21]. Whilst our security definition applies to skewed RPCs, our security analysis does not account for randomness in the cache system. In particular, it would be interesting to observe how this randomness improves the rekeying period of RPCs.

Another path to take is to improve the provided bounds. Tighter lower bounds could probably be found, especially for the $\mathsf{RP} = \mathsf{RAND}$ case and the

noise scenario, by carrying a more involved probability analysis or considering different threat models.

Variants of our security definitions could also be explored. While our N-access security definition is rather general and applies to all state-of-the-art RPCs, it only concerns detecting accesses to a single target address through access-based attacks. It is possible to study the multi-target case, adapting the security definition so that the goal of the adversary is to detect accesses to one, some or all target addresses. The results obtained for the single-target case could be of use in reductions from multi-target cases. Different attacker objectives could be considered as well, for instance learning an eviction set of a target address.

It is possible to develop a similar analysis to the one presented here for security against other attacks that require more privileges from the attacker, particularly in the case of timing-based attacks [LWML16]. These attacks assume that the attacker can test whether an access to the target address hits or misses before being presented the challenge. Analyzing this case would require modifying the game in the security definition, and the derived rekeying periods would be smaller due to the increased abilities of the adversary.

Finally, recent work as ClepsydraCache [TNF+21] or Entropy-Shield [DMR+20] considers additional techniques on top of randomization, such as time-to-live features or noise. One could attempt to follow the line of the present work, and provide a formal analysis that allows to assess the impact of these auxiliary techniques on security.

Acknowledgements

This research was supported by the European Union Regional Development Fund within the framework of the ERDF Operational Program of Catalonia 2014-2020 with a grant of 50% of the total cost eligible, under the DRAC project [001-P-001723], and by the Spanish Government, under the CONSENT project [RTI2018-095094-B-C21]. Carles Hernández is partially supported by Spanish Ministry of Science, Innovation and Universities under "Ramón y Cajal", fellowship No. RYC2020-030685-I. Vatistas Kostalabros is partially supported by the Agency for Management of University and Research Grants (AGAUR) of the Government of Catalonia, under "Ajuts per a la contractació de personal investigador novell", fellowship No. 2019FI B01274. Miquel Moretó is partially supported by the Spanish Ministry of Economy, Industry and Competitiveness under "Ramón y Cajal", fellowship No. RYC-2016-21104.

References

AB00. Michel Abdalla and Mihir Bellare. Increasing the lifetime of a key: a comparative analysis of the security of re-keying techniques. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 546–559. Springer, 2000.

And19. Ruth Anderson. CSE 351: The hardware/software interface. Lecture notes: Virtual memory III, May 2019.

AR20. Andreas Abel and Jan Reineke. nanobench: A low-overhead tool for running microbenchmarks on x86 systems. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, August 2020.

BDY+20. Thomas Bourgeat, Jules Drean, Yuheng Yang, Lillian Tsai, Joel Emer, and Mengjia Yan. Casa: End-to-end quantitative security analysis of randomly mapped caches. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2020, Athens, Greece, October 17-21, 2020*, pages 1110–1123. IEEE, 2020.

Ber05. Daniel J. Bernstein. Cache-timing attacks on AES, 2005.

BGS+20. Rahul Bodduna, Vinod Ganesan, Patanjali SLPSK, Kamakoti Veezhinathan, and Chester Rebeiro. Brutus: Refuting the security claims of the cache timing randomization countermeasure proposed in CEASER. *IEEE Comput. Archit. Lett.*, 19(1):9–12, 2020.

BKR98. Mihir Bellare, Ted Krovetz, and Phillip Rogaway. Luby-rackoff backwards: Increasing security by making block ciphers non-invertible. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 266–280. Springer, 1998.

BM06. Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against aes. In Louis Goubin and Mitsuru Matsui, editors, *Cryptographic Hardware and Embedded Systems - CHES 2006*, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

BRS+10. Lawrence Bassham, Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, Stefan Leigh, M Levenson, M Vangel, Nathanael Heckert, and D Banks. A statistical test suite for random and pseudorandom number generators for cryptographic applications, September 2010.

BSN+19. Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMoTherSpectre: Exploiting speculative execution through port contention. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 785–800. ACM Press, November 2019.

Cha. Champsim simulator, available at `https://github.com/ChampSim/ChampSim`.

CKK+20. Somnath Chakrabarti, Thomas Knauth, Dmitrii Kuvaiskii, Michael Steiner, and Mona Vij. Chapter 8 - trusted execution environment with intel sgx. In Xiaoqian Jiang and Haixu Tang, editors, *Responsible Genomic Data Sharing*, pages 161–190. Academic Press, 2020.

CVS+19. Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In Nadia Heninger and Patrick Traynor, editors, *USENIX Security 2019*, pages 249–266. USENIX Association, August 2019.

DJL+12. Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael B. Abu-Ghazaleh, and Dmitry Ponomarev. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Trans. Archit. Code Optim.*, 8(4):35:1–35:21, 2012.

DKMPHL20. Max Doblas, Ioannis-Vatistas Kostalabros, Miquel Moreto Planas, and Carles Hernández Luz. Enabling hardware randomization across the cache hierarchy in Linux-Class processors. In *Fourth Workshop on Computer Architecture Research with RISC-V (CARRV 2020): Virtual Workshop, Friday, May 29th, 2020: co-located with ISCA 2020*, pages 1–7, 2020.

DMR+20. Abhijitt Dhavlle, Raj Mehta, Setareh Rafatirad, Houman Homayoun, and Sai Manoj Pudukotai Dinakarrao. Entropy-shield: Side-channel entropy maximization for timing-based side-channel attacks. In *21st International Symposium on Quality Electronic Design, ISQED 2020, Santa Clara, CA, USA, March 25-26, 2020*, pages 161–166. IEEE, 2020.

DXS20. Shuwen Deng, Wenjie Xiong, and Jakub Szefer. *A Benchmark Suite for Evaluating Caches' Vulnerability to Timing Attacks*, page 683–697. Association for Computing Machinery, 2020.

GGM86. Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *J. ACM*, 33(4):792–807, aug 1986.

GMF+16. Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 368–379. ACM Press, October 2016.

GMWM16. Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+flush: A fast and stealthy cache attack. In Juan Caballero, Urko Zurutuza, and Ricardo J. Rodríguez, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment - 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings*, volume 9721 of *LNCS*, pages 279–299. Springer, 2016.

GRB+17. Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the line: Practical cache attacks on the MMU. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017.

HWKS98. Chris Hall, David Wagner, John Kelsey, and Bruce Schneier. Building prfs from prps. In *Annual International Cryptology Conference*, pages 370–389. Springer, 1998.

IGI+16. Mehmet Sinan Inci, Berk Gülmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cache attacks enable bulk key recovery on the cloud. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *CHES 2016*, volume 9813 of *LNCS*, pages 368–388. Springer, Heidelberg, August 2016.

KHF+19. Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy*, pages 1–19. IEEE Computer Society Press, May 2019.

KLA+18. Vladimir Kiriansky, Ilia A. Lebedev, Saman P. Amarasinghe, Srinivas Devadas, and Joel S. Emer. DAWG: A defense against cache timing attacks in speculative execution processors. In *51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka,*

Japan, October 20-24, 2018, pages 974–987. IEEE Computer Society, 2018.

LL14. Fangfei Liu and Ruby B. Lee. Random fill cache architecture. In *47th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2014, Cambridge, United Kingdom, December 13-17, 2014*, pages 203–215. IEEE Computer Society, 2014.

LSG+18. Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In William Enck and Adrienne Porter Felt, editors, *USENIX Security 2018*, pages 973–990. USENIX Association, August 2018.

LWML16. Fangfei Liu, Hao Wu, Kenneth Mai, and Ruby B. Lee. Newcache: Secure cache architecture thwarting cache side-channel attacks. *IEEE Micro*, 36(5):8–16, 2016.

LYG+15. Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622. IEEE Computer Society Press, May 2015.

Mon18. John V. Monaco. SoK: Keylogging side channels. In *2018 IEEE Symposium on Security and Privacy*, pages 211–228. IEEE Computer Society Press, May 2018.

MWS+17. Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the other side: SSH over robust cache covert channels in the cloud. In *NDSS 2017*. The Internet Society, February / March 2017.

OKSK15. Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The spy in the sandbox: Practical cache attacks in JavaScript and their implications. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015*, pages 1406–1418. ACM Press, October 2015.

OST06. Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In David Pointcheval, editor, *CT-RSA 2006*, volume 3860 of *LNCS*, pages 1–20. Springer, Heidelberg, February 2006.

Per05. Colin Percival. Cache missing for fun and profit, 2005.

PGGV21. Antoon Purnal, Lukas Giner, Daniel Gruss, and Ingrid Verbauwhede. Systematic analysis of randomization-based protected cache architectures. In *42th IEEE Symposium on Security and Privacy*, volume 5, 2021.

QBBC09. Eduardo Quiñones, Emery D. Berger, Guillem Bernat, and Francisco J. Cazorla. Using randomized caches in probabilistic real-time systems. In *21st Euromicro Conference on Real-Time Systems, ECRTS 2009, Dublin, Ireland, July 1-3, 2009*, pages 129–138. IEEE Computer Society, 2009.

Qur18. Moinuddin K. Qureshi. CEASER: mitigating conflict-based cache attacks via encrypted-address and remapping. In *51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018*, pages 775–787. IEEE Computer Society, 2018.

Qur19. Moinuddin K. Qureshi. New attacks and defense for encrypted-address cache. In Srilatha Bobbie Manne, Hillery C. Hunter, and Erik R. Altman, editors, *Proceedings of the 46th International Symposium on Computer Architecture, ISCA 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 360–371. ACM, 2019.

Sar19. John Sarraillé. Operating systems I: CS 3750. Lecture notes: Main memory. Chapter nine, June 2019.

Sez93. André Seznec. A case for two-way skewed-associative caches. In Alan Jay Smith, editor, *Proceedings of the 20th Annual International Symposium on Computer Architecture, San Diego, CA, USA, May 1993*, pages 169–178. ACM, 1993.

SLG+18. Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. KeyDrown: Eliminating software-based keystroke timing side-channel attacks. In *NDSS 2018*. The Internet Society, February 2018.

SSW14. Somayeh Sardashti, André Seznec, and David A. Wood. Skewed compressed caches. In *47th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2014, Cambridge, United Kingdom, December 13-17, 2014*, pages 331–342. IEEE Computer Society, 2014.

SWG+17. Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using SGX to conceal cache attacks. *CoRR*, abs/1702.08719, 2017.

THAC18. David Trilla, Carles Hernández, Jaume Abella, and Francisco J. Cazorla. Cache side-channel attacks and time-predictability in high-performance critical real-time systems. In *Proceedings of the 55th Annual Design Automation Conference, DAC 2018, San Francisco, CA, USA, June 24-29, 2018*, pages 98:1–98:6. ACM, 2018.

TNF+21. Jan Philipp Thoma, Christian Niesler, Dominic A. Funke, Gregor Leander, Pierre Mayr, Nils Pohl, Lucas Davi, and Tim Güneysu. Clepsydracache - preventing cache attacks with time-based evictions. *CoRR*, abs/2104.11469, 2021.

VKM19. Pepe Vila, Boris Köpf, and José F. Morales. Theory and practice of finding eviction sets. In *2019 IEEE Symposium on Security and Privacy*, pages 39–54. IEEE Computer Society Press, May 2019.

VMW+18. Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In William Enck and Adrienne Porter Felt, editors, *USENIX Security 2018*, pages 991–1008. USENIX Association, August 2018.

WL07. Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. In Dean M. Tullsen and Brad Calder, editors, *34th International Symposium on Computer Architecture (ISCA 2007), June 9-13, 2007, San Diego, California, USA*, pages 494–505. ACM, 2007.

WUG+19. Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. ScatterCache: Thwarting cache attacks via cache set randomization. In Nadia Heninger and Patrick Traynor, editors, *USENIX Security 2019*, pages 675–692. USENIX Association, August 2019.

YCJQ18. Vinson Young, Chia-Chen Chou, Aamer Jaleel, and Moinuddin K. Qureshi. ACCORD: enabling associativity for gigascale DRAM caches by coordinating way-install and way-prediction. In Murali Annavaram, Timothy Mark Pinkston, and Babak Falsafi, editors, *45th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2018, Los Angeles, CA, USA, June 1-6, 2018*, pages 328–339. IEEE Computer Society, 2018.

ZJRR14. Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in PaaS clouds. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 2014*, pages 990–1003. ACM Press, November 2014.

A Assumptions for our RPC model

Our models for cache systems and cache randomizers make a few assumptions to facilitate the security analysis, and leave aside superfluous features. As per cache systems, we make the following assumptions:

Cache-only architecture: Since this work is only concerned with the interaction with the cache, our model completely disregards the main memory, the MMU and the CPU. We give notation for the cache and for cache randomizers. The constraints that other components impose on the threat model are explained in Appendix B.

Offset dismissal: All memory blocks addressed with the same tag and set index get mapped to the same cache line, and the offset part determines where the block does get stored in the line. Therefore, the cache state and the observed latencies are unaffected by the offset. As in all previous work, we dismiss the offset part of the address. This choice does not affect security.

Tag-only storage: The data retrieved from main memory does not contribute to the functionality of the cache or the observed latencies. Thus, our cache model holds at most one tag per cache line, and no data.

Cache hierarchy: As a component, the cache usually consists of several memory devices that form a leveled *cache hierarchy*, with lower-level caches sitting closer to the CPU. This hierarchy is often *inclusive*, meaning that lower-level caches always hold a subset of the data of higher-level caches. The LLC is the farthest to the core, it usually has the highest latency, and it is the only one shared among different cores. In this context, as in prior work [Qur18,THAC18,Qur19,WUG+19] and without loss of generality, we restrict to the protection of the LLC and see the cache as a single memory device as laid out in Section 2.1. Partition-based solutions [WL07,DJL+12,KLA+18] could be better suited for lower-level caches.

Slicing: Many systems divide the cache sets of their LLC into disjoint *slices*, and a slice is selected on every access through a hash of the tag and the set index. Seeing this hash as part of the cache randomization process, our model combines all slices into one that contains all cache sets.

And, as per the cache randomizer, we make the following assumptions:

Randomizer as a component: Regarding the actual placement of the randomizer in the architecture, there exist two approaches. In works such as CAESER [Qur18], addresses go through an encryption process before entering the cache. Since plain physical addresses should be used in main memory (otherwise, changing keys would shuffle the main memory), any address outgoing from the cache is decrypted to retrieve the original physical address. Other work, such as ScatterCache [WUG+19], applies the address randomization in the cache, temporarily storing the physical address so that it can be output as is if needed. Without loss of generality, we follow the latter approach, since it does not require defining an inverse to the randomizer and so it allows to consider a wider range of solutions.

Set index randomization: In some of the previous work, randomization is only applied to the set index part of the address, while others randomize both the tag and the set index. Since we assume process isolation is enforced with the tag (see Appendix B), randomizing the tag does not report any advantage against contention attacks. We take the set index randomization approach.

B Threat model

Access-based attacks on caches can have different objectives. For instance, the attacker may be interested in knowing whether a process accesses one known or unknown address, or a certain cache set, or in monitoring the addresses accessed by another process. Moreover, there are several points to be taken into account when modeling the adversary and the attack environment.

In this work, an RPC is deemed insecure when an adversary can detect an access to a target address by observing the latency of adaptive accesses. The attacker objectives and capabilities to be taken into account when modelling our security definition (see Definition 7) are driven by the following considerations:

Targeted addresses: Instead of targeting a particular address [PGGV21], attacks can also target arbitrary addresses [VKM19]. The former attacks aim at building sets of addresses that, when sequentially accessed, have a high probability of causing auto-evictions (i.e., not all of them ending up stored in the cache). Both options are of a different nature, and we consider only targeted attacks since they pose an arguably greater threat.

Number of target addresses: It is possible for targeted attacks to aim at monitoring accesses to one or to many addresses at the same time (as in Bernstein's attack on AES [Ber05]), and the attack could succeed if accesses to all or just some of the accesses are detected. This depends on the process under attack, and different security definitions may apply in different cases. The most conservative objective is to detect accesses to any address out of a set of target addresses. However, in this article we focus on the more natural case of detecting one access to a single target address (σ, τ), which makes sense for some attacks presented in the literature [PGGV21]. In the

terminology of Bourgeat et al. [BDY+20], we aim at security against *single-address transmitters*. At any rate, our results may translate to other cases.

Global keying: As per keying, cache randomizers may require a single key in the system, or they can manage a different key per process. In the first case, the attacker may learn congruence relations between addresses that also hold for the victim process. However, knowing congruence relations is not necessary to perform access-based attacks such as [OST06], and does not affect security as we define it. We restrict to the global keying case for clarity.

Rekeying period: This period can be fixed using the clock time, the number of accesses issued to the cache, or by other means. We consider a fixed number of cache accesses in each epoch. Note that a smaller rekeying period impacts the efficiency of the final solution, because all local writes have to be flushed to main memory prior to rekeying. We also assume that flushing completely cleans the cache; otherwise, stale data could affect the reliability of the system. In our security definition, we make this explicit with the assignment $\mathtt{C}[s] \leftarrow (\mathtt{NULL})_j$ for every $s \in S$.

Number of accesses: We adopt a flexible approach when counting the number of accesses in each epoch. We assume that N_1 accesses are issued to prepare the cache before the possible access to the target address, and that N_2 accesses are issued after it to detect if it was carried or not. This accounts for epochs lasting at most $N = N_1 + N_2$ accesses. We also assume that N_1 and N_2 are known to the adversary.

Access to the actual cache: Since we study latency-based attacks, we assume that the attacker can not view the cache $\mathtt{C}$, nor access it by any other means than a black-box evaluation of the access_{π_k} function.

Access to the cache randomizer: As other work [PGGV21], in a real attack setting we may assume that the design of the randomization function π is publicly available but that the actual key k is not, and that the attacker can not evaluate π_k directly due to hardware restrictions. In particular, this implies that the attacker can not see which cache set is modified by the access_{π_k} function, or in which line, but it may be able to infer some information from latency measures.

Process isolation in the cache: Isolation should be enforced in hardware, so that the accesses of the attacker do not collide with accesses by victim processes in the cache if this is not authorized. However, different addresses could collide in the cache after randomization if the whole address is randomized. And, in physically-indexed caches, tags could be the same for different data, and set indexes could then coincide after randomizing. Previous work implements measures to preserve process isolation in the cache even in those cases, e.g. taking the whole frame number as the tag [THAC18].

In this article, without loss of generality, we assume that the attacker can not generate collisions with the target address $x = (\sigma, \tau)$ in the cache, by preventing it from using the tag τ. This is in line with the measures to enforce process isolation taken in previous work, for instance by Trilla et

al. [THAC18]. Nevertheless, we assume that the attacker has full knowledge of the target address x.

Physical indexing: As explained in Section 2.1, the set index of the address can be part of the virtual or the physical address. In the first case, which is typical of lower-level caches, the attacker is in direct control of the set index. In the second case, which is more typical of higher-level caches (such as the LLC), part of the set index depends on the address translation enforced by the OS. As the most conservative choice, and since the attacker could have some knowledge of the page table, we give the attacker the freedom to input to access_{π_k} any address in $S \times T \setminus \{\tau\}$ in order to set up an attack. In an honest scenario, however, we note that freely accessing randomized addresses with the same tag could easily lead to data integrity and coherency issues if no additional measures are taken.