Logical Methods in Computer Science
Volume 15, Issue 1, 2019, pp. 21:1–21:19
https://lmcs.episciences.org/

Submitted Jul. 04, 2018
Published Mar. 05, 2019

ON THE INCOMPARABILITY OF CACHE ALGORITHMS IN TERMS OF TIMING LEAKAGE

PABLO CAÑONES, BORIS KÖPF, AND JAN REINEKE

IMDEA Software Institute and Universidad Politécnica de Madrid, Madrid, Spain
e-mail address: pablo.canones@imdea.org

IMDEA Software Institute, Madrid, Spain and Microsoft Research, Cambridge, UK
e-mail address: boris.koepf@microsoft.com

Saarland University, Saarbrücken, Germany
e-mail address: reineke@cs.uni-saarland.de

Abstract. Modern computer architectures rely on caches to reduce the latency gap between the CPU and main memory. While indispensable for performance, caches pose a serious threat to security because they leak information about memory access patterns of programs via execution time.

In this paper, we present a novel approach for reasoning about the security of cache algorithms with respect to timing leaks. The basis of our approach is the notion of *leak competitiveness*, which compares the leakage of two cache algorithms on every possible program. Based on this notion, we prove the following two results:

First, we show that leak competitiveness is *symmetric* in the cache algorithms. This implies that no cache algorithm dominates another in terms of leakage via a program's total execution time. This is in contrast to performance, where it is known that such dominance relationships exist.

Second, when restricted to caches with finite control, the leak-competitiveness relationship between two cache algorithms is either asymptotically linear or constant. No other shapes are possible.

1. Introduction

Modern computer architectures rely on caches to reduce the latency gap between the CPU and main memory. Accessing data that is cached (a cache hit) can be hundreds of CPU cycles faster than accessing data that needs to be fetched from main memory (a cache miss), which translates into significant performance gains.

While caches are indispensable for performance, they pose a serious threat to security. An attacker who can distinguish between cache hits and misses via timing measurements can learn information about the memory access pattern of a victim's program. This side channel has given rise to a large number of documented attacks, e.g. [AK06, ASK07, Ber05, GBK11, KGG+18, LSG+18, LYG+15, OST06, YF14].

Key words and phrases: security, cache memory, cache algorithms, competitiveness.

LOGICAL METHODS IN COMPUTER SCIENCE
DOI:10.23638/LMCS-15(1:21)2019

From a security point of view it would be ideal to completely eliminate cache side channels by design, as in [TOL+11, ZWSM15]. Unfortunately, such conservative approaches also partially void the performance benefits of caches. In practice, one usually seeks to identify a trade-off between security and performance, which requires comparing different cache designs in terms of their security and performance properties.

While performance analysis of cache designs is an established field [AZMM04, Dor10] there are only few approaches concerned with analyzing their security. Examples are [HL17], which analyzes the effect of the size of the cache and how it is shared between different agents, and [CKR17], which measures the security of cache algorithms with respect to adversaries who can gather information about a victim's computation by probing the state of a shared cache.

In this paper, we present a novel approach for evaluating the security of caches. More precisely, we focus on the amount of information that a cache algorithm leaks to an adversary that can measure a program's overall execution time. Leakage through a program's overall execution time is practically relevant because it can be exploited remotely, and conceptually interesting because the notions of security and performance are tightly coupled – even though there are more powerful ways to spy on a program via shared caches [YF14].

The basis of our approach is a novel notion for comparing the leakage of cache algorithms, which we call *leak competitiveness*. Leak competitiveness is inspired by *competitiveness*, which is a standard notion for comparing the performance of online algorithms, and in particular cache algorithms [RG08, ST85]. However, whereas competitive performance analysis compares cache algorithms on individual traces, leak competitiveness compares the leakage of cache algorithms on *sets* of traces, which accounts for the fact that information flow is a hyperproperty [CS10].

The central contribution of this paper is a characterization of the possible leak-competitiveness relationships between any two cache algorithms:

- We find that leak competitiveness is *symmetric* in the cache algorithms. This implies that no cache algorithm dominates another in terms of leakage via execution time. Note that this is in contrast to performance, where it is known that such dominance relationships exist [RG08]. This result holds for a very general class of deterministic cache algorithms, including fully-associative caches, set-associative caches with arbitrary replacement policies, and even rather exotic caches such as skewed-associative caches.
- If we restrict our attention to caches with finite control, which is natural for hardware-based cache implementations, the leak-competitiveness relationship between two cache algorithms is either asymptotically linear in the length of the program execution or it is constant. No other shapes are possible.

The proofs of these results are based on three intermediate steps that are of independent interest.

(1) The first is to show that a pair of traces of memory accesses precisely characterizes the leak competitiveness relationship between any two cache algorithms.
(2) The second step is to show that we can actually identify a *single* trace of memory accesses for which the difference in number of misses between both algorithms matches their leak competitiveness to within a factor of 2. This is surprising in the light that leakage is a hyperproperty, i.e., it requires *sets* of traces to express.
(3) The third step is to define a congruence on the cache contents of algorithms with finite control – but potentially infinite data – and to show that the resulting quotient is

finite. Our characterization of leak competitiveness follows from the observation that, if the trace that witnesses the leak competitiveness is large enough, it will visit multiple congruent cache states, i.e. contain a cycle in the quotient. We then use a pumping argument to obtain a linear lower bound on the leak competitiveness from this cycle.

Organization of the paper. The remainder of this paper is structured as follows. In Section 2 we introduce a general model of deterministic cache algorithms. In Section 3 we introduce leak competitiveness and leak ratio, based on which we present our main results in Section 4. Sections 5–7 present the proof of our results, following the structure outlined above. We present related work in Section 8 before we conclude in Section 9.

2. Preliminaries

Caches are fast but small memories that store a subset of the main memory's contents to bridge the latency gap between the CPU and the main memory. To profit from spatial locality and to reduce management overhead, main memory is logically partitioned into a set B of memory blocks. Each block is cached as a whole in a cache line of the same size. When accessing a memory block, the cache logic has to determine whether the block is stored in the cache ("cache hit") or not ("cache miss"). In the case of a miss, the cache algorithm decides which memory block to evict and replace by a new one.

Definition 2.1. A *cache algorithm* (or *algorithm*) is a tuple

$$P = (S_P, i_P, n_P, tr_P, evict_P),$$

which consists of the following components:

- The set of *control states*, S_P.
- The *initial* control state, $i_P \in S_P$.
- The *capacity* of the cache, $n_P \in \mathbb{N}$.
- The *transition* function, $tr_P : S_P \times \{0, \dots, n_P - 1\} \to S_P$, that, upon a hit to one of its n_P cache lines, determines the new control state of the cache.
- The *evict* function, $evict_P : S_P \times B \to S_P \times \{0, \dots, n_P - 1\}$, that, upon a miss, determines the new control state of the cache and the cache line to evict.

During runtime a *cache configuration* consists of the cache's control state and of its current *content*. The content is captured by a function $c : C_P = \{0, \dots, n_P - 1\} \to B \cup \{\bot\}$ that maps each cache line to the memory block it holds, or $\bot$ if the line is invalid. A cache configuration $g = (s, c) \in G_P = S_P \times C_P$ is updated as follows upon a memory access:

$$update_P((s, c), b) := \begin{cases} (s', c) & \text{if } \exists j : c(j) = b \wedge s' = tr_P(s, j), \\ (s', c[j \leftarrow b]) & \text{if } \forall k : c(k) \neq b \wedge (s', j) = evict_P(s, b). \end{cases} \tag{2.1}$$

Upon a hit, the update function is used to obtain the new control state. Upon a miss, the accessed block replaces one of the cached blocks, determined by the evict function.

The above definition of a cache algorithm is quite general: it captures arbitrary deterministic caches that operate on a bounded capacity buffer. This includes direct-mapped, set-associative, fully-associative caches, and even skewed-associative caches with arbitrary deterministic replacement policies. Well-known *deterministic* replacement policies which fit our model are *least-recently used* (LRU), used in various Freescale processors such as the MPC603E and the TriCore17xx, as well as the recent Kalray MPPA 256; *pseudo-LRU*

(PLRU), a cost-efficient variant of LRU, used in the Freescale MPC750 family and multiple Intel microarchitectures; *most-recently used* (MRU), also known as *not most-recently used* (NMRU), another cost-efficient variant of LRU, used in the Intel Nehalem; *first-in first-out* (FIFO), also known as Round Robin, used in several ARM and Freescale processors such as the ARM922 and the Freescale MPC55xx family; Pseudo-Round Robin, used in the NXP Coldfire 5307.

Notation: The update function is lifted to traces $t \in B^*$ of blocks recursively as follows:

$$update_P((s,c),\epsilon) := (s,c),$$
$$update_P((s,c),b \circ t) := update_P(update_P((s,c),b),t).$$

The number of misses $P((s,c),t)$ of an algorithm P on a trace $t \in B^*$ starting in configuration (s,c) is determined recursively as follows:

$$P((s,c),\epsilon) := 0,$$
$$P((s,c),b \circ t) := miss(b,c) + P(update_P(b,(s,c)),t),$$

where $miss(b,c) = (\forall j : c(j) \neq b \ ?\ 1 : 0)$.

We use $P(t)$ as a shortcut for $P((i_P, \lambda j.\bot), t)$, i.e., the number of misses on the trace t when starting in the initial configuration of the cache. Also, $P(t,t')$ is a shortcut for $P(tt') - P(t)$, i.e., the number of misses on the suffix t'.

3. Leak Ratio

In this section we define a measure for comparing the security of cache algorithms, which we call the leak ratio. The leak ratio is inspired by quantitative notions of security in information flow analysis [KB07, Smi09] and by the notion of relative miss competitiveness [RG08] from the real-time systems community. We revisit both notions first.

3.1. Relative Miss Competitiveness.

Relative competitiveness [RG08] is a notion for comparing the worst-case performance of two cache algorithms. It is based on the classic notion of competitiveness [ST85], which compares an online algorithm with the optimal offline algorithm. Below, we reproduce a slightly simplified version of the definition of relative competitiveness from [RG08]:

Definition 3.1. For $r \in \mathbb{R}_{>0}$, we say that a algorithm P is *r-miss-competitive relative to* algorithm Q if there exists $c \in \mathbb{R}_{>0}$ such that

$$P(t) \leq r \cdot Q(t) + c,$$

for all traces $t \in B^*$.

Example 3.2. LRU of associativity 4 is 1-miss-competitive relative to FIFO of associativity 2. On the other hand, FIFO of associativity 2 is not r-miss-competitive to LRU of associativity 4 for any r. Therefore, LRU of associativity 4 outperforms FIFO of associativity 2 in number of misses. See [RG08] for details and more examples.

3.2. Leak Competitiveness.

We next introduce a notion based on relative competitiveness that compares the amount of information that two cache algorithms leak via their timing behavior. We begin by recalling basic concepts from quantitative information-flow analysis.

3.2.1. *Quantifying Leaks.* As is common in side-channel analysis based on quantitative information-flow [KB07], we quantify the amount of information a system leaks in terms of the number of observations an adversary can make. This number represents an upper bound on entropy loss, for different notions of entropy including Shannon entropy, min-entropy [Smi09], or g-vulnerability [ACPS12]. Each of those notions of entropy is associated with an interpretation in terms of security. For example, using min-entropy as a basis for the interpretation, a bound on the number of observations corresponds to an upper bound on the factor by which guessing becomes easier through side-channel information.

For formalizing a program's leakage through cache timing effects, we abstract the program in terms of the set T of traces of memory accesses it can perform. We always consider traces of finite length l, hence $T \subseteq B^l$. We capture the attacker's observation of a program execution as the number of cache misses produced by the corresponding trace. If l is known to the adversary, then she can deduce the number of misses from the program's overall execution time, due to the large latency gap between cache hits and cache misses[1]. The information the program leaks through timing is hence captured by $P(T) \subseteq \mathbb{N}$, the image of T under P, and quantified by $|P(T)| \in \mathbb{N}$.

3.2.2. *Comparing Leaks.* Notions of performance such as relative competitiveness (see Definition 3.1) are based on trace properties, hence the point of comparison are individual traces. In contrast, information-theoretic notions of leakage are hyperproperties, which makes *sets* of traces the natural point of comparison. We now define *leak competitiveness*, a concept that enables us to compare the timing leakage of two cache algorithms, and that is based on lifting miss competitiveness from traces to sets of traces.

Definition 3.3. For a function $r\colon \mathbb{N} \to \mathbb{R}_{\geq 0}$, we say that algorithm P is *r-leak-competitive* relative to algorithm Q if, for all $l \in \mathbb{N}$,

$$|P(T)| \leq r(l) \cdot |Q(T)|\,,$$

for all set of traces of blocks $T \subseteq B^l$.

Even though the definition of leak competitiveness is based on a lifting of miss competitiveness, there are important differences. Most importantly, leak competitiveness of two algorithms P, Q bounds the ratio of leakage for each $l \in \mathbb{N}$, whereas miss competitiveness bounds the ratio of hits and misses for all l. For traces of length l and an empty initial cache, the number of misses any cache algorithm can produce is in $\{1, \ldots, l\}$, which means that any two algorithms are r-leak-competitive for $r(l) = l$. The question is hence not whether two cache algorithms are leak-competitive, but rather what shape this relationship takes. We introduce the leak ratio to facilitate reasoning about this shape.

Definition 3.4. Given a pair of algorithms P and Q we define the *leak ratio* $r_{P,Q}$ as:

$$r_{P,Q}(l) = \min\{r(l) \mid r\colon \mathbb{N} \to \mathbb{R}_{\geq 0}, P \text{ is } r\text{-leak-competitive relative to } Q\}.$$

As we are mostly interested in the asymptotic behavior of $r_{P,Q}$, the lack of an additive slack in the definition of miss competitiveness is not essential.

[1]We consider "noiseless" attacks where the attacker is able to obtain the maximum amount of information from the cache. This assumption is common in the literature on security to safely over-approximate the security for real life attacks.

4. Characterizing the Leak Ratio

In this section we present our main result, which is a characterization of the asymptotic behavior of the leak ratio for any pair of cache algorithms. We then give interpretations of this behavior in terms of security. We present the proofs of the technical results in Sections 5-7.

4.1. Non-Dominance.

The key question motivating our work is whether some cache algorithms are preferable to others in terms of their leakage via timing. This is a natural question to ask because it is well-known that such preferences relations exist for performance, see Example 3.2. The following theorem gives a negative answer to the question above.

Theorem 4.1. *For each pair of algorithms P, Q we have, as l grows:*

$$\mathcal{O}(r_{P,Q}(l)) = \mathcal{O}(r_{Q,P}(l)).$$

Theorem 4.1 shows that cache algorithms are incomparable in the sense that, for every $l \in \mathbb{N}$ and every set of traces T that witnesses an advantage for P over Q in terms of leakage, there is a set of traces T' that witnesses a comparable advantage of Q over P. The following examples exhibits such witnesses for $P = \text{LRU}$ and $Q = \text{FIFO}$.

Example 4.2. Consider two fully-associative caches of capacity two, one with LRU and the other with FIFO replacement, and the following sets of traces:

$$T = \left\{ \begin{array}{l} \text{ABACACBBB}, \\ \text{ABACDAAAA}, \\ \text{ABACBADDD}, \\ \text{ABACBACBB}, \\ \text{ABACBACBA} \end{array} \right\} \qquad T' = \left\{ \begin{array}{l} \text{ABACBAAAA}, \\ \text{ABACDAAAA}, \\ \text{ABACABCCC}, \\ \text{ABACACBCA} \end{array} \right\}$$

Starting from an empty initial cache state, LRU produces 5 different observations on T, whereas FIFO produces only one. In contrast, FIFO produces 4 different observations on T' whereas LRU produces only one.

The root cause for this divergent behavior is that, after accessing the prefix ABAC, the content of both caches differs: block C evicts the least recently used block for LRU (i.e., B) but the first block to enter the cache for FIFO (i.e., A). The suffixes of the traces are constructed in such a way that the difference in cache content maps to different observable behavior. The full diagram of updates of the cache when using these sets of traces is given in Figure 2 in the Appendix.

The proof of Theorem 4.1 is based on a systematic way of constructing sets of traces such as the ones in Example 4.2. Formally, the theorem follows from applying Theorem 6.3, introduced in Section 6, to P, Q and to Q, P. Moreover, we will see later that these sets can be obtained from only two traces of memory blocks and that those two traces are enough to characterize the leak ratio.

4.2. **Shapes of $r_{P,Q}$.** In Section 3 we have already observed that the leak ratio $r_{P,Q}$ between any two algorithms P and Q is upper bounded by a linear function. The interesting question is hence what sublinear shapes $r_{P,Q}$ can take. We answer this question for cache algorithms with finite sets of control states, which encompasses most hardware-based cache implementations. For this important class, the following theorem shows that the leak ratio is either asymptotically constant or linear, ruling out any nontrivial sublinear shape.

Theorem 4.3. *For each pair of algorithms P, Q with finite control we have either*
- $r_{P,Q}(l) \in \Theta(l)$, *or*
- $r_{P,Q}(l) \in \Theta(1)$.

These results are a direct consequence of Theorem 7.4, introduced in Section 7, which shows that the leak ratio of two finite-control algorithms P, Q is lower bounded by a linear factor if and only if there exist traces that witness that the difference in misses between P and Q is unbounded. Whether such traces exist determines in which of the two classes described by Theorem 4.3 the algorithms P and Q fall. If they do not exist, note that Corollary 6.2 implies that $r_{P,Q} \in \mathcal{O}(1)$.

For example, any pair of algorithms with different capacities falls into the first class. This is because one algorithm always contains a block that the other does not, which allows to construct a trace of unbounded difference in misses.

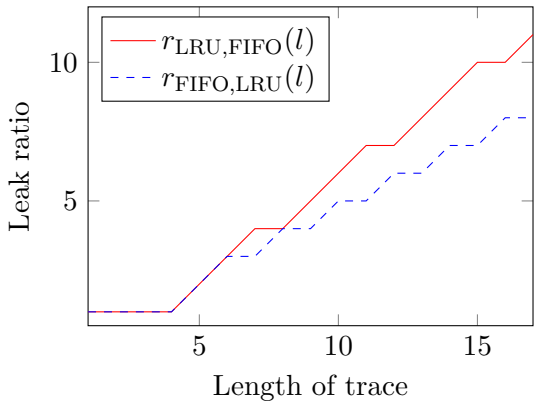
Together with Theorem 4.1, Theorem 4.3 leads to a stronger non-dominance result for finite-control algorithms, namely that for every $l \in \mathbb{N}$ there are sets of traces $T_P^l, T_Q^l \subseteq B^l$ such that one algorithm asymptotically leaks the largest possible amount of information whereas the other leaks almost nothing. That is, $P(T_P^l) \in \Theta(1)$ and $Q(T_P^l) \in \Theta(l)$, whereas $P(T_Q^l) \in \Theta(l)$ and $Q(T_Q^l) \in \Theta(1)$.

The non-dominance results from Theorems 4.1 and 4.3 also show why we cannot define leak competitiveness in the same way as miss-competitiveness, that is, where each pair of cache algorithms has a constant leak ratio for all lengths of traces. Except for the case where both leak ratios are in $\Theta(1)$, if we can find, for each length, sets of traces where one cache algorithm leaks more and more information as we increase the length whereas the other leaks a constant amount, no constant value of the leak ratio satisfies the leak-competitiveness definition for all lengths.

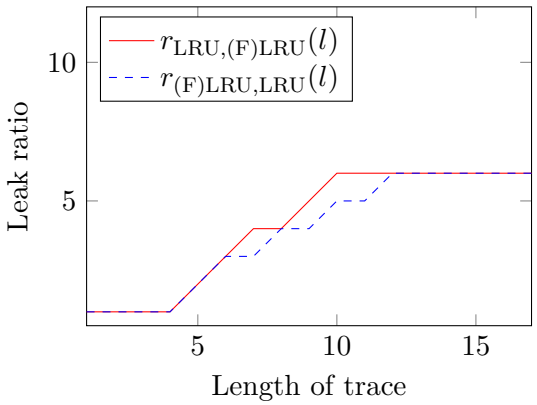
We now compute the leak ratio functions for two pairs of cache algorithms to showcase how the constants ignored in the asymptotic results in Theorems 4.1 and 4.3 can mean a small advantage of one cache algorithm over the other, provided the leak ratios are not in $\Theta(1)$.

Example 4.4. We can compute the leak ratios for small lengths of traces by computing all traces of hits and misses of a given length and then choosing the subset of traces that produces the largest ratio in the number of observations in favor of each algorithm.

To compute these traces of hits and misses we simulate them by exhaustively enumerating all possible traces of memory blocks B^l. We now argue the size of the set B needed for the case of capacity two. For every pair of configurations updated from the initial by the same trace of memory blocks, the last accessed block is cached (2.1), accessing this block again produces a hit for both configurations. The other line of the configuration may store a different memory block for each cache algorithm so that accessing one of them produces a hit for one algorithm and a miss for the other and vice versa. Finally, accessing any memory block not cached for any of the cache algorithms produces a miss for both. Then, for cache



(A) Comparison of the leak ratios of LRU relative to FIFO and vice versa.



(B) Comparison of the leak ratios of LRU relative to (F)LRU and vice versa.

FIGURE 1. Example of the behavior of the leak ratios of the cache algorithms from Example 4.4. Figure 1A shows two leak ratios that grow asymptotically linearly and where the slopes can give a slight advantage of one algorithm over the other. Figure 1A shows two leak ratios that eventually become constant functions and were, for large lengths of traces, there is no advantage of one algorithm over the other.

algorithms with capacity two, four memory blocks are enough to simulate all possible traces of hits and misses.

Consider two pairs of fully-associative caches of capacity two, one pair considers replacements LRU and FIFO while the other considers LRU and a cache algorithm that we denote (F)LRU that starts behaving like FIFO but, after seven accesses to memory, behaves like LRU for the remaining of the accesses. The leak ratios for LRU and FIFO are shown in Figure 1A and the ones for LRU and (F)LRU are shown in Figure 1B.

We see that the leak ratios of LRU and FIFO exemplify the first case of Theorem 4.3 and that, by looking at the slopes, that the asymptotic approach ignores, we conclude that FIFO has a small advantage over LRU since the leak ratio of FIFO relative to LRU grows slower than that of LRU relative to FIFO, Figure 1A.

On the other hand, the leak ratios of LRU and (F)LRU exemplify the second case of Theorem 4.3. The leak ratios for both cache algorithms start growing at different rates but, once both algorithms behave like LRU the leak ratios end up coinciding and become constant functions. Although both cache algorithms behave the same starting from length eight, not all pairs of configurations contain the same memory blocks at this point, which still allows for both leak ratios to grow with the length. Once all traces update the pairs of configurations to having the same blocks for both algorithms, the leak ratios become constant functions.

A pair of cache algorithms that are the same or eventually become the same are the only ones that verify the second case of Theorem 4.3. This is because the leak ratios grow when the pairs of configurations do not have the same memory blocks cached and the access to a specific memory block produces a hit for one algorithm and a miss for the other. If, at some point, every access to a memory block has the same effect for both algorithms, the leak ratios do not grow anymore.

By using a constant notion of leak competitiveness, all cache algorithms, except for the ones in $\Theta(1)$, would be deemed incomparable. On the other hand, by defining the leak ratios as functions of the length of the trace and observing the growth rate as in Example 4.4, we can establish a comparison between cache algorithms.

4.3. **Discussion.** We now discuss the implications of Theorems 4.1 and 4.3 (short: *our results*) in practice.

(1) Our results are asymptotic in nature. The constants hidden behind the $\mathcal{O}$-notation can indicate a (gradual) preference between algorithms on finite sets of traces. E.g., the traces in Example 4.2 and the different slopes on Example 4.4 show a slight advantage of FIFO over LRU.
(2) Our results rely on the construction of sets of traces that witness advantages of one cache algorithm over another, see Example 4.2. However, the constructed traces need not correspond to a program of interest. Restricting to a specific class of programs corresponds to the constraint that witnesses be picked from a subset $T \subseteq B^l$ instead of B^l. Under such constraints, it may be possible that a preference relation between cache algorithms exists.
(3) Our results rely on the assumptions that the caching algorithm is deterministic and based on demand paging, i.e., it loads blocks only when they are requested by the program. It is possible that randomized policies or features such as prefetching enable one to sidestep our results. For example, for miss-competitiveness it is known that randomized policies [FKL$^+$91] achieve better bounds than those possible for deterministic policies [ST85]. The study of leak competitiveness for randomized cache algorithms is out of the scope of this paper. We are aware of non-dominance results for a similar notion of leak-competitiveness that consider the RANDOM cache algorithm that, upon a miss, evicts a memory block randomly [Sch18].
(4) Our results rely on an adversary that can observe the overall execution time of the program as in, e.g. [Ber05, OST06]. They do not necessarily hold for adversaries that can observe the cache state after or during the computation of the victim. Such attacks are possible whenever the adversary shares the cache with the victim, which has shown to be the most effective attack vector. In contrast, our results are relevant for remote attacks, which are less effective, but harder to detect and defend against. We briefly discuss the case of access-based adversaries in Section 8.

Despite these limitations in scope, we do believe that our results lay an interesting basis for theory research in the domain of microarchitectural side-channel attacks, where foundational results are still scarce.

5. Leak Ratio from a Pair of Traces

Information leakage is a hyperproperty, i.e., a property of sets of traces. We now show that the leak can always be expressed in terms of the difference in observations of only two traces of memory blocks.

For an algorithm Q and $l \in \mathbb{N}$, we say that $t_1, t_2 \in B^l$ are Q-*equivalent* whenever $Q(t_1) = Q(t_2)$. We say that a set $T \subseteq B^l$ is Q-*dense* if the image of T under Q is a contiguous sequence of natural numbers, i.e. $Q(T) = \{j, j+1, \ldots, j+k\}$ for some $j, k \in \mathbb{N}$.

Proposition 5.1. *For all pairs of algorithms P and Q, all lengths l, and all pairs of Q-equivalent traces of memory blocks $t_1, t_2 \in B^l$:*

$$P(t_2) - P(t_1) \leq r_{P,Q}(l) - 1. \tag{5.1}$$

Moreover, there exist pairs of traces of Q-equivalent memory blocks $t_1, t_2 \in B^l$ such that (5.1) *is an equality.*

That is, every pair of traces that coincides in timing observation on one algorithm cannot differ by more than the leak ratio on the other algorithm. Moreover, there exists a pair of traces that matches this bound.

The proof of the upper bound is based on constructing a set $T \subseteq B^l$ of Q-equivalent traces from a pair $t, t' \in B^l$ of Q-equivalent traces. The set T is P-dense with maximum $P(t')$ and minimum $P(t)$. It satisfies

$$r_{P,Q}(l) \geq \frac{|P(T)|}{|Q(T)|}, \tag{5.2}$$

which equals $P(t) - P(t') + 1$ by construction.

The following lemma describes the construction of traces Q-equivalent to t and t' whose number of misses for P cover every value in between $P(t)$ and $P(t')$. The set T is composed of these traces.

Lemma 5.2. *Consider two Q-equivalent traces t, $t' \in B^l$ with $P(t) \leq P(t')$. Then, for every $P(t) \leq k \leq P(t')$ there exists a trace $t^* \in B^l$ such that $P(t^*) = k$ and that is Q-equivalent to t and t'.*

Proof. We begin with a continuity argument to identify a prefix of the trace t, which we later extend to t^*. For this, note that the difference in misses, $Q - P$, between both algorithms on trace t is initially zero, i.e. $Q(\epsilon) - P(\epsilon) = 0$, and increases or decreases by at most 1 per added block, until it reaches $Q(t) - P(t)$. We first consider the case $Q(t) \geq P(t)$. For any k with $0 \leq Q(t) - k \leq Q(t) - P(t)$, we hence find a prefix $b_1 \cdots b_u$ of t such that the value of $Q - P$ on the prefix is exactly $Q(t) - k$:

$$Q(b_1 \cdots b_u) - P(b_1 \cdots b_u) = Q(t) - k. \tag{5.3}$$

We create a trace t^* with prefix $b_1^* \ldots b_u^* = b_1 \ldots b_u$, which we extend by blocks $b_{u+1}^* \ldots b_v^*$ that produce misses on both P and Q until

$$Q(b_1^* \ldots b_v^*) = Q(t)\ . \tag{5.4}$$

For the blocks $b_{u+1}^* \ldots b_v^*$ to miss they must be uncached in both P and Q; such blocks can be found whenever B is larger than the sum of the the capacities of both algorithms. We further extend $b_1^* \ldots b_v^*$ with $l - v$ copies of b_v^* to the trace t^* of length l. Repeatedly accessing b_v^* is guaranteed to produce hits on both P and Q.

As the blocks $b_{u+1}^* \ldots b_l^*$ produce identical outputs on P and Q, the trace t^* still satisfies (5.3), i.e.,

$$Q(t^*) - P(t^*) = Q(t) - k\ .$$

Moreover, t^* also still satisfies (5.4), i.e., $Q(t^*) = Q(t)$, from which we conclude that t^* is Q-equivalent to t and $P(t^*) = k$. Note that we only handled the case $P(t) \leq Q(t)$ so that $k \leq Q(t)$. The case $P(t) > Q(t)$ where $k > Q(t)$ proceeds in the same way but extending a prefix of t' instead of t and reformulating (5.3) to $P(b'_1 \cdots b'_u) - Q(b'_1 \cdots b'_u) = k - Q(t')$. □

Example 5.3. Consider the cache algorithms $P = \text{LRU}$ and $Q = \text{FIFO}$ and the traces of memory blocks ABACACBBB and ABACBACBA, as in Example 4.2. Both traces are FIFO-equivalent, however $\text{LRU}(\text{ABACACBBB}) = 4$ and $\text{LRU}(\text{ABACBACBA}) = 8$. Then, following Lemma 5.2, there exist three traces of memory blocks that are FIFO-equivalent but where LRU produces between 5 and 7 misses, namely:

$$\{\text{ABACDAAAA}, \text{ABACBADDD}, \text{ABACBACBB}\}$$

The union of this set with the two initial traces yields the set T from Example 4.2.

The proof of the tightness of the upper bound in Proposition 5.1 follows from the fact that every set T that satisfies equality in (5.2) contains within it a subset T^* of Q-equivalent traces that also satisfies equality in (5.2). We show that this set T^* is P-dense, which means that the elements $t, t' \in T^*$ that produce the maximal difference in misses under P satisfy $P(t) - P(t') = r_{P,Q} - 1$.

The following lemma shows how to find such a T^*.

Lemma 5.4. *Every set $T \subseteq B^l$ that satisfies equality in* (5.2) *contains a P-dense subset of Q-equivalent traces that also satisfies equality in* (5.2)*.*

Proof. We partition $T = T_1 \uplus \cdots \uplus T_k$, into classes of Q-equivalent traces. Without loss of generality assume that $P(T_1) \geq P(T_j)$, for $j > 1$. Then we have:

$$\frac{|P(T)|}{|Q(T)|} \leq \frac{\sum_{j=1}^{k} |P(T_j)|}{\sum_{j=1}^{k} |Q(T_j)|} = \frac{\sum_{j=1}^{k} |P(T_j)|}{\sum_{j=1}^{k} 1} \overset{(*)}{\leq} |P(T_1)| \ ,$$

where $(*)$ follows from the fact that, for any sequence of natural numbers $a_1, \ldots, a_k$, $\sum_{j=1}^{k} a_j \leq k \max(a_1, \ldots, a_k)$. As a consequence, T_1 also satisfies $|P(T_1)| = r_{P,Q}(l)$. Moreover, $P(T_1)$ is a contiguous set of natural numbers. If it were not, we could apply Lemma 5.2 to augment T_1 by a trace that produces the missing number of observations, contradicting that $r_{P,Q}$ is an upper bound. □

6. Approximation of the Leak Ratio from a Single Trace

In Section 5 we have seen that the leak ratio of two cache algorithms, which is defined as a property of arbitrary sets of traces, is fully characterized by a pair of traces. In this section, we show that the leak ratio can be approximated to within a factor of 2 using a single trace.

Lemma 6.1. *Let $t \in B^l$ be an arbitrary trace. Then, there is a trace $t' \in B^l$ with $P(t') = Q(t') = Q(t)$.*

Proof. We construct the trace $t' \in B^l$ as the concatenation of two subtraces t'_{miss} and t'_{hit}: t'_{miss} is a trace of length $Q(t)$ in which all accesses are chosen such that they result in misses in both P and Q. This is always possible, as there are at most $n_P + n_Q$ blocks cached in P and Q at any time and accesses to any other block will result in a miss. Let $b \in B$ be the final access in t'_{miss}. Independently of the cache algorithm, b must be cached in both P and Q following t'_{miss}. The second subtrace t'_{hit} then simply consists of $|t| - Q(t)$ accesses to b, which will result in hits in both P and Q. □

The following corollary of the previous lemma and of Proposition 5.1 shows that the leakage ratio is "almost" a trace property, as it can be approximated to within a factor of two based on the number of misses of P and Q on a single trace:

Corollary 6.2. *For all pairs of cache algorithms P and Q, all lengths l, and all traces of memory blocks $t \in B^l$:*

$$|P(t) - Q(t)| \leq r_{P,Q}(l) - 1. \tag{6.1}$$

Moreover, there exists a trace $t \in B^l$ such that:

$$\frac{r_{P,Q}(l) - 1}{2} \leq |P(t) - Q(t)|.$$

Proof. Let $t \in B^l$ be an arbitrary trace. By Lemma 6.1, there is a trace t' such that $P(t') = Q(t') = Q(t)$. So t and t' are Q-equivalent. Thus, by Proposition 5.1, we have both

$$P(t) - P(t') \leq r_{P,Q}(l) - 1 \quad \textit{and} \quad P(t') - P(t) \leq r_{P,Q}(l) - 1,$$

which implies that $|P(t) - Q(t)| = |P(t) - P(t')| \leq r_{P,Q}(l) - 1$.

By Proposition 5.1, there is a pair of Q-equivalent traces $t_1, t_2 \in B^l$ such that:

$$P(t_2) - P(t_1) = r_{P,Q}(l) - 1.$$

Let $q = Q(t_1) = Q(t_2)$. Either $2 \cdot |P(t_2) - q| \geq P(t_2) - P(t_1)$ or $2 \cdot |P(t_1) - q| \geq P(t_2) - P(t_1)$, where equality is achieved on one of the two inequalities if q is centered between $P(t_1)$ and $P(t_2)$. Assume that $2 \cdot |P(t_2) - q| \geq P(t_2) - P(t_1)$. Then $|P(t_2) - Q(t_2)| \geq \frac{P(t_2) - P(t_1)}{2} = \frac{r_{P,Q} - 1}{2}$. Otherwise, $|P(t_1) - Q(t_1)| \geq \frac{P(t_2) - P(t_1)}{2} = \frac{r_{P,Q} - 1}{2}$. □

Theorem 6.3. *For all pairs of cache algorithms P and Q and all lengths l:*

$$r_{P,Q}(l) \leq 2 \cdot r_{Q,P}(l) - 1$$

Proof. By Corollary 6.2, there is a trace $t \in B^l$, such that

$$\frac{r_{Q,P}(l) - 1}{2} \leq |Q(t) - P(t)| = |P(t) - Q(t)| \leq r_{P,Q}(l) - 1.$$

Multiplying both sides by 2 and adding 1 finish the proof. □

7. A Linear Lower Bound on the Leak Ratio

In this section, we show that if the difference in misses between two cache algorithms is unbounded, then there are traces on which the difference in misses grows linearly in the length of the trace. Together with the result from the previous section, this implies that the leak ratio between two algorithms grows linearly in the length of the trace if and only if the difference between the two algorithms is unbounded. This result does not hold for arbitrary caches conforming to the model introduced in Section 2. We need to make two additional assumptions:

(1) We assume the set of control states S_P of a cache algorithm to be finite. This is naturally the case for hardware-based caches that maintain a finite set of status bits to guide future eviction decisions.
(2) We assume that the *evict* function, $evict_P : S_P \times B \to S_P \times \{0, \ldots, n_P - 1\}$ is independent of its second parameter, i.e., $evict_P(s, b) = evict_P(s, b')$ for all $s \in S_P$ and $b, b' \in B$. This assumption is naturally fulfilled by fully-associative caches, where there is no restriction

on the placement of a memory block based on its address. This assumption could be significantly weakened at the expense of a more complicated proof.[2]

For the proof of the result we argue that, while there is an unbounded number of different cache configurations, even assuming an unbounded supply of memory blocks B, there are only finitely many "non-congruent" pairs of cache configurations, where congruent will be defined precisely below. Intuitively, congruent pairs of cache configurations behave similarly to each other, if their cache contents are appropriately renamed.

Such a renaming can be captured by a bijection. Let $\pi : B \to B$ be a bijection on memory blocks and let π^* denote its extension to cache contents that maps $\bot$ to $\bot$:

$$\pi^*(c) = \lambda l. \begin{cases} \pi(c(l)) & : c(l) \in B \\ \bot & : c(l) = \bot \end{cases}$$

We also lift π to cache configurations with $\pi^*(s, c) = (s, \pi^*(c))$ and to traces with $\pi^*(\epsilon) = \epsilon$ and $\pi^*(b \circ t) = \pi(b) \circ \pi^*(t)$.

Let (s, c) be an arbitrary cache configuration. Observe that:

$$\forall t \in B^* : \pi^*(update_P((s, c), t)) = update_P(\pi^*(s, c), \pi^*(t)), \tag{7.1}$$

i.e. renamed cache configurations behave the same on renamed accesses. Also observe that:

$$miss_P(b, c) = miss_P(\pi(b), \pi^*(c)), \tag{7.2}$$

which holds because $\pi(b)$ is contained in $\pi^*(c)$ if and only if b is contained in c. From these two observations, it follows that:

$$P((s, c), t) = P(\pi^*(s, c), \pi(t)). \tag{7.3}$$

Definition 7.1 (Congruent cache configurations)**.** Two pairs of cache configurations (g_P, g_Q) and (g'_P, g'_Q) are *congruent*, denoted by $(g_P, g_Q)) \equiv (g'_P, g'_Q)$, if there is a bijection $\pi : B \to B$, such that $g'_P = \pi^*(g_P)$ and $g'_Q = \pi^*(g_Q)$. To indicate a bijection π that is a witness to the congruence of two pairs of cache configurations we also write $(g_P, g_Q) \equiv_\pi (g'_P, g'_Q)$.

Note that congruence is an equivalence relation. We denote the equivalence class of a pair of cache configuration (g_P, g_Q) by

$$[g_P, g_Q] := \{(g'_P, g'_Q) \in G_P \times G_Q \mid (g'_P, g'_Q) \equiv (g_P, g_Q)\}.$$

While the set of pairs of cache configurations is infinite, its quotient w.r.t. to the congruence relation is finite:

Theorem 7.2 (Index of $\equiv$)**.** *Let P and Q be two finite-control-state cache algorithms. Then, the quotient*

$$G_P \times G_Q/_\equiv \; = \{[g_P, g_Q] \mid (g_P, g_Q) \in G_P \times G_Q\}$$

is finite.

Proof. Remember that n_P and n_Q denote the capacities of P and Q. Let $B_{P,Q}$ be an arbitrary but fixed subset of B, such that $|B_{P,Q}| = n_P + n_Q$.

We show below that each pair $((s_P, c_P), (s_Q, c_Q))$ of cache configurations is congruent to a pair of cache configurations $((s_P, c'_P), (s_Q, c'_Q))$ in which only blocks from $B_{P,Q}$ may occur in the cache contents c'_P and c'_Q. As $B_{P,Q}$ is finite, there are only finitely many different

[2] A weaker, yet sufficient condition would be that there is a finite partition of B, such that $evict_P(s, b) = evict_P(s, b')$ for all $s \in S_P$ and b, b' that are in the same block of the partition. This weaker assumption is fulfilled by arbitrary set-associative caches.

cache contents c'_P and c'_Q containing only blocks from $B_{P,Q}$. The sets of control states S_P and S_Q are finite by assumption. Together, this implies that the set of equivalence classes of $\equiv$ is finite.

Below we show how to incrementally construct a bijection $\pi : B \to B$ such that the contents of $c'_P = \pi^*(c_P)$ and $c'_Q = \pi^*(c_Q)$ contain only blocks from $B_{P,Q}$:

(1) Initially, let π be the identity function on B, and let $D = B_{P,Q}$.
(2) For $i = 0, \ldots, n_P - 1$:
    If $c_P(i) \in B_{P,Q}$ then modify D to $D = D \setminus \{c_P(i)\}$.
(3) For $j = 0, \ldots, n_Q - 1$:
    If $c_Q(j) \in B_{P,Q}$ then modify D to $D = D \setminus \{c_Q(j)\}$.
(4) For $i = 0, \ldots, n_P - 1$:
    If $c_P(i) \neq \bot$ and $\pi(c_P(i)) \notin B_{P,Q}$, then pick $b \in D$ and modify π and D as follows:
    $\pi = \pi[c_P(i) \mapsto b][b \mapsto c_P(i)]$ and $D = D \setminus \{b\}$.
(5) For $j = 0, \ldots, n_Q - 1$:
    If $c_Q(j) \neq \bot$ and $\pi(c_Q(j)) \notin B_{P,Q}$, then pick $b \in D$ and modify π and D as follows:
    $\pi = \pi[c_Q(j) \mapsto b][b \mapsto c_Q(j)]$ and $D = D \setminus \{b\}$.

Note that there is always a $b \in D$ available, when the above algorithm needs one, because the operation is applied at most $|B_{P,Q}| = n_P + n_Q$ times. Throughout its execution, the algorithm maintains the invariant that π is a bijection. Further, the resulting bijection satisfies $\pi^*(c_P), \pi^*(c_Q) \subseteq B_{P,Q} \cup \{\bot\}$. □

We can exploit Theorem 7.2 in a manner similar to the application of the pumping lemma for regular languages in the proof of the following theorem.

Theorem 7.3. *Let P and Q be two finite-control-state cache algorithms. Further, let the difference in misses between P and Q be unbounded, i.e.,*

$$\forall m \in \mathbb{N} : \exists t \in B^* : |P(t) - Q(t)| > m.$$

Then, there is an $f \in \mathbb{R}, f > 0$ and an $m_0 \in \mathbb{N}$, such that

$$\forall m \in \mathbb{N}, m > m_0 : \exists t \in B^m : |P(t) - Q(t)| > f \cdot |t|.$$

Proof. Let P and Q be two finite-control-state cache algorithms such that the difference in misses between P and Q is unbounded. Let $l = |G_P \times G_Q/_{\equiv}| + 1$, which must be finite due to Theorem 7.2.

As the difference in misses between P and Q is unbounded, there must be a $t \in B^*$ such that $|P(t) - Q(t)| = l$. We will assume without loss of generality[3] that $P(t) > Q(t)$ for such traces t, and so $|P(t) - Q(t)| = P(t) - Q(t)$. Then, let $t_1, \ldots, t_l$ be prefixes of t, s.t. $P(t_j) - Q(t_j) = j$ for all $1 \leq j \leq l$.

Let $ig_P = (i_P, \lambda j.\bot)$ and $ig_Q = (i_Q, \lambda j.\bot)$ be the initial configurations of P and Q. Also, let $p_j = update_P(ig_P, t_j)$ and $q_j = update_Q(ig_Q, t_j)$ for all $1 \leq j \leq l$.

Due to the pigeonhole principle, there must be at least two prefixes t_j and t_k, with $j < k$, such that the pairs of cache configurations (p_j, q_j) and (p_k, q_k) resulting from executing these prefixes are congruent. Assume that t_j and t_k are two such prefixes.

As t_j is a prefix of t_k, we can decompose t_k into t_j and $t_{j \to k}$, such that $t_k = t_j \circ t_{j \to k}$. From $P(t_j) - Q(t_j) = j$ and $P(t_k) - Q(t_k) = k$ we can conclude that $P(t_j, t_{j \to k}) - Q(t_j, t_{j \to k}) = (P(t_k) - P(t_j)) - (Q(t_k) - Q(t_j)) = k - j \geq 1$.

[3] If $P(t) < Q(t)$ the following arguments hold with P and Q exchanged.

We can arbitrarily extend t_j using the following construction of the traces τ_m and ω_m:

$$\begin{aligned}\tau_0 &= t_j,\\ \tau_{m+1} &= \tau_m \circ \omega_m,\\ \omega_0 &= t_{j\to k},\\ \omega_{m+1} &= \pi^*(\omega_m).\end{aligned}$$

Let $u_m = update_P(ig_P, \tau_m)$ and $v_m = update_Q(ig_Q, \tau_m)$. For the following induction proof, it will be helpful to express u_{m+1} and v_{m+1} in terms of u_m and v_m. We have that $u_{m+1} = update_P(ig_P, \tau_m \circ \omega_m) = update_P(update_P(ig_P, \tau_m), \omega_m) = update_P(u_m, \omega_m)$ and similarly $v_{m+1} = update_Q(v_m, \omega_m)$.

We can show by induction that $(u_m, v_m) \equiv_\pi (u_{m+1}, v_{m+1})$:

- (Induction base) For $m = 0$, $\tau_0 = t_j$ and $\tau_1 = \tau_0 \circ \omega_0 = t_j \circ t_{j\to k} = t_k$. Thus we have that $u_0 = update_P(ig_P, \tau_0) = update_P(ig_P, t_j) = p_j$ and $v_0 = update_Q(ig_Q, \tau_0) = update_Q(ig_Q, t_j) = q_j$. Similarly, $u_1 = p_k$ and $v_1 = q_k$, and we already know that $(p_j, q_j) \equiv_\pi (p_k, q_k)$.
- (Induction step) For $m > 0$, we know from the induction hypothesis that $(u_{m-1}, v_{m-1}) \equiv_\pi (u_m, v_m)$. Applying (7.1) with $t = \omega_{m-1}$ yields $\pi^*(u_m) = \pi^*(update_P(u_{m-1}, \omega_{m-1})) = update_P(u_m, \pi^*(\omega_{m-1})) = update_P(u_m, \omega_m) = u_{m+1}$, and similarly $\pi^*(v_m) = v_{m+1}$. Thus $(u_m, v_m) \equiv_\pi (u_{m+1}, v_{m+1})$.

Since we have that $u_{m+1} = \pi(u_m)$ and $v_{m+1} = \pi(v_m)$, applying (7.3) yields that $P(u_{m+1}, \omega_{m+1}) = P(u_{m+1}, \pi^*(\omega_m)) = P(u_m, \omega_m)$ and similarly we have $P(v_{m+1}, \omega_{m+1}) = P(v_m, \omega_m)$ for all m. In other words, the number of misses on the subtraces ω_m are always the same in both P and Q. We also know that $P(u_0, \omega_0) = P(t_j, t_{j\to k})$ and $Q(v_0, \omega_0) = Q(t_j, t_{j\to k})$. Thus, we have

$$\begin{aligned}P(\tau_m) &= P(t_j) + m \cdot P(t_j, t_{j\to k}),\\ Q(\tau_m) &= Q(t_j) + m \cdot Q(t_j, t_{j\to k}),\\ P(\tau_m) - Q(\tau_m) &= P(t_j) - Q(t_j) + m \cdot (P(t_j, t_{j\to k}) - Q(t_j, t_{j\to k})),\\ &= P(t_j) - Q(t_j) + m \cdot (k - j).\end{aligned}$$

Let $f = \frac{k-j}{|t_{j\to k}|+1} \geq \frac{1}{|t_{j\to k}|+1} > 0$. For large enough m, $|P(\tau_m) - Q(\tau_m)| = P(t_j) - Q(t_j) + m \cdot (k - j)$ is greater than $f \cdot |\tau_m| = \frac{k-j}{|t_{j\to k}|+1} \cdot (|\tau_0| + m \cdot |t_{j\to k}|)$, which proves the theorem. □

In other words, if the difference in misses between two finite-control-state algorithms is unbounded, then it actually grows linearly in the length of the trace.

Theorem 7.4. *The leak ratio between two finite-control-state cache algorithms P and Q grows linearly in the length of the trace if and only if the difference in misses between P and Q is unbounded:*

$$r_{P,Q}(l), r_{Q,P}(l) \in \Omega(l) \quad \Leftrightarrow \quad \forall m \in \mathbb{N} : \exists t \in B^* : |P(t) - Q(t)| > m.$$

Proof. Direction "⇒": Assume for a contradiction that there is an $m_{max} \in \mathbb{N}$, such that for all traces $t \in B^*$: $|P(t) - Q(t)| \leq m_{max}$. As $r_{P,Q}(l) \in \Omega(l)$ there must be an l^*, such that $r_{P,Q}(l^*) > 2 \cdot m_{max} + 1$. By the second part of Corollary 6.2, there is a trace t such that

$$m_{max} < \frac{r_{P,Q}(l^*) - 1}{2} \leq |P(t) - Q(t)|,$$

which contradicts our assumption.

Direction "⇐": We will prove that $r_{P,Q}(l) \in \Omega(l)$. The fact that $r_{Q,P}(l) \in \Omega(l)$ follows by simply exchanging P and Q because $|P(t) - Q(t)| = |Q(t) - P(t)|$.

To prove that $r_{P,Q}(l) \in \Omega(l)$, we have to show that there is a $k > 0$ and an $m_0 \in \mathbb{N}$, such that $\forall m \in \mathbb{N}, m > m_0 : r_{P,Q}(m) \geq k \cdot m$.

By Theorem 7.3, we can conclude that there is an $f > 0$ and an $m'_0 \in \mathbb{N}$, such that $\forall m \in \mathbb{N}, m > m'_0 : \exists t \in B^m : |P(t) - Q(t)| > f \cdot |t|$. Pick k to be f and m_0 to be m'_0. To prove the theorem it then remains to show that $\exists t \in B^m : |P(t) - Q(t)| > f \cdot |t|$ implies $r_{P,Q}(m) > f \cdot m$.

To this end, let $t \in B^m$ be a trace that satisfies $|P(t) - Q(t)| > f \cdot |t|$. Applying the first part of Corollary 6.2 yields $f \cdot |t| < |P(t) - Q(t)| < r_{P,Q}(m)$. □

8. Related Work

Leak competitiveness is inspired by work on competitive performance analysis. The notion of competitive analysis was first introduced in [ST85], where the authors bound the number of misses an online algorithm does on a trace of memory blocks in terms of the number of misses of an optimal offline algorithm [Bel66]. In contrast to performance, there is no clear candidate for an optimal offline cache algorithm for security, because the best option would be not to cache memory blocks and be trivially non-interferent. This is why we base leak competitiveness on relative competitiveness [RG08]. Here, the number of misses one cache algorithm produces is compared with the number of misses of another algorithm, none of them necessarily optimal.

The notion of leakage we use is based on concepts from quantitative information-flow analysis [CHM07, ACPS12, Smi09]. They have been successfully used for detecting and quantifying side channels of program code [DKMR15, HM10, KB07, NMS09].

Concepts from quantitative information-flow analysis have been applied to the analysis of cache algorithms [CKR17]. Our work goes beyond this in two crucial aspects. First, we consider adversaries that measure overall execution time of a victim, whereas [CKR17] consider so-called *access-based adversaries* that gain information by probing the state of a shared cache after the victim's computation terminates. Second, our analysis is based on a comparison of cache algorithms on each program, whereas [CKR17] identifies the worst possible program for each.

We can, however, interpret some of the results of [CKR17] in terms of leak-competitiveness w.r.t. to an access-based adversary. The bound that governs leakage in this scenario is not the length of the trace but rather the number of memory blocks used by (i.e. the *footprint* of) the victim program. With this, one can read Propositions 6 and 7 of [CKR17] as follows:

- for FIFO and LRU, the number of observations of an access-based adversary is bounded by a constant. This implies that the leak ratios of FIFO relative to LRU, and of LRU relative to FIFO, are in $\mathcal{O}(1)$;
- for PLRU, the number of observations grows at least linearly with the footprint. This implies that the leak ratio of PLRU relative to FIFO and LRU, respectively, is in $\Omega(n)$, whereas the leak ratio of FIFO and LRU relative to PLRU is in $\mathcal{O}(1)$.

Overall, these examples show that, unlike for time based adversaries, there are dominance relations for the security of cache algorithms with respect to access-based adversaries. We leave a detailed investigation of this case to future work.

Finally, a line of work focuses on secure cache architectures [HL17, ZL14]. They consider different architectures, either introducing some sort of partition on the cache or randomness in the replacement of memory blocks, and study their resilience against different kinds of cache side-channel attacks.

When it comes to timing attacks, they mention that, introducing some sort of randomness is the only way to reduce the vulnerability to leak information in this cases. This is because with deterministic cache algorithms, the attacker knows that the observation he obtains only depends on the victim's accesses to memory. Our work acknowledges that this dependence is unavoidable for deterministic cache algorithms but tries to quantify how specific cache algorithms make the dependance less dangerous.

9. Conclusions

We presented a novel approach to compare cache algorithms in terms of their vulnerability to side-channel attacks. Our core insight is that for leakage, as opposed to performance, there is no dominance relationship between any two cache algorithms, in the sense that one algorithm would outperform the other on all programs.

Acknowledgments. We thank Pierre Ganty and the anonymous reviewers for their constructive feedback.

This work was supported by Microsoft Research through its PhD scholarship programme, a gift from Intel Corporation, Ramón y Cajal grant RYC-2014-16766, Spanish projects TIN2015-70713-R DEDETIS and TIN2015-67522-C3-1-R TRACES, and Madrid regional project S2013/ICE-2731 N-GREENS.

References

[ACPS12] Mário Alvim, Kostas Chatzikokolakis, Catuscia Palamidessi, and Geoffrey Smith. Measuring information leakage using generalized gain functions. In *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th*, pages 265–279. IEEE, 2012.

[AK06] Onur Acıiçmez and Çetin Koç. Trace-driven cache attacks on AES. *Information and Communications Security*, pages 112–121, 2006.

[ASK07] Onur Acıiçmez, Werner Schindler, and Çetin K. Koç. Cache based remote timing attack on the AES. In *Cryptographers Track at the RSA Conference*, pages 271–286. Springer, 2007.

[AZMM04] Hussein Al-Zoubi, Aleksandar Milenkovic, and Milena Milenkovic. Performance evaluation of cache replacement policies for the spec cpu2000 benchmark suite. In *Proceedings of the 42nd annual Southeast regional conference*, pages 267–272. ACM, 2004.

[Bel66] Laszlo A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal*, 5(2):78–101, 1966.

[Ber05] Daniel Bernstein. Cache-timing attacks on AES. `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`, 2005.

[CHM07] David Clark, Sebastian Hunt, and Pasquale Malacaria. A static analysis for quantifying information flow in a simple imperative language. *JCS*, 15(3):321–371, 2007.

[CKR17] Pablo Cañones, Boris Köpf, and Jan Reineke. Security analysis of cache replacement policies. In *POST*, pages 189–209. Springer, 2017.

[CS10] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

[DKMR15] Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. CacheAudit: a tool for the static analysis of cache side channels. *ACM Transactions on Information and System Security (TISSEC)*, 18(1):4:1–4:32, 2015.

[Dor10] Reza Dorrigiv. Alternative measures for the analysis of online algorithms, 2010.

[FKL+91] Amos Fiat, Richard M. Karp, Michael Luby, Lyle A. McGeoch, Daniel D. Sleator, and Neal E. Young. Competitive paging algorithms. *Journal of Algorithms*, 12(4):685–699, 1991.

[GBK11] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games - bringing access-based cache attacks on AES to practice. In *SSP*, pages 490–505. IEEE, 2011.

[HL17] Zecheng He and Ruby B. Lee. How secure is your cache against side-channel attacks? In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 341–353. ACM, 2017.

[HM10] Jonathan Heusser and Pasquale Malacaria. Quantifying information leaks in software. In *ACSAC*, pages 261–269. ACM, 2010.

[KB07] Boris Köpf and David Basin. An Information-Theoretic Model for Adaptive Side-Channel Attacks. In *Proc. 14th ACM Conference on Computer and Communications Security (CCS '07)*, pages 286–296, New York, NY, USA, 2007. ACM.

[KGG+18] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *arXiv preprint arXiv:1801.01203*, 2018.

[LSG+18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *arXiv preprint arXiv:1801.01207*, 2018.

[LYG+15] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *SSP*, pages 605–622. IEEE, 2015.

[NMS09] James Newsome, Stephen McCamant, and Dawn Song. Measuring channel capacity to distinguish undue influence. In *PLAS*, pages 73–85. ACM, 2009.

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In *CT-RSA*, pages 1–20. Springer, 2006.

[RG08] Jan Reineke and Daniel Grund. Relative competitive analysis of cache replacement policies. In *LCTES*, pages 51–60, New York, NY, USA, June 2008. ACM.

[Sch18] Felix Schröder. Security of cache replacement policies under side-channel attacks. 2018.

[Smi09] Geoffrey Smith. On the foundations of quantitative information flow. In *FoSSaCS*, pages 288–302. Springer, 2009.

[ST85] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[TOL+11] Mohit Tiwari, Jason Oberg, Xun Li, Jonathan Valamehr, Timothy E. Levin, Ben Hardekopf, Ryan Kastner, Frederic T. Chong, and Timothy Sherwood. Crafting a usable microkernel, processor, and I/O system with strict and provable information flow security. In *ISCA*, pages 189–200. ACM, 2011.

[YF14] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *USENIX*, pages 719–732. USENIX Association, 2014.

[ZL14] Tianwei Zhang and Ruby B. Lee. New models of cache architectures characterizing information leakage from cache side channels. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 96–105. ACM, 2014.

[ZWSM15] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. A hardware design language for timing-sensitive information-flow security. In *ASPLOS*, pages 503–516. ACM, 2015.

$$\begin{array}{lccccccccccccccccccc}
\text{LRU} & \{\bot,\bot\} & \xrightarrow[(1,1)]{\text{A}} & \{\text{A},\bot\} & \xrightarrow[(1,1)]{\text{B}} & \{\text{B},\mathbf{\text{A}}\} & \xrightarrow[(0,0)]{\text{A}} & \{\text{A},\mathbf{\text{B}}\} & \xrightarrow[(1,1)]{\text{C}} & \{\text{C},\mathbf{\text{A}}\} & \xrightarrow[(0,1)]{\text{A}} & \{\text{A},\mathbf{\text{C}}\} & \xrightarrow[(0,0)]{\text{C}} & \{\text{C},\mathbf{\text{A}}\} & \xrightarrow[(1,1)]{\text{B}} & \{\text{B},\mathbf{\text{C}}\} & \xrightarrow[(0,0)]{\text{B}} & \{\text{B},\mathbf{\text{C}}\} & \xrightarrow[(0,0)]{\text{B}} & \{\text{B},\mathbf{\text{C}}\} & \\
\text{FIFO} & \{\bot,\bot\} & & \{\text{A},\bot\} & & \{\text{B},\mathbf{\text{A}}\} & & \{\text{B},\mathbf{\text{A}}\} & & \{\text{C},\mathbf{\text{B}}\} & & \{\text{A},\mathbf{\text{C}}\} & & \{\text{A},\mathbf{\text{C}}\} & & \{\text{B},\mathbf{\text{A}}\} & & \{\text{B},\mathbf{\text{A}}\} & & \{\text{B},\mathbf{\text{A}}\} & : (4,5)
\end{array}$$

$$\begin{array}{lccccccccccccccccccc}
\text{LRU} & \{\bot,\bot\} & \xrightarrow[(1,1)]{\text{A}} & \{\text{A},\bot\} & \xrightarrow[(1,1)]{\text{B}} & \{\text{B},\mathbf{\text{A}}\} & \xrightarrow[(0,0)]{\text{A}} & \{\text{A},\mathbf{\text{B}}\} & \xrightarrow[(1,1)]{\text{C}} & \{\text{C},\mathbf{\text{A}}\} & \xrightarrow[(1,1)]{\text{D}} & \{\text{D},\mathbf{\text{C}}\} & \xrightarrow[(1,1)]{\text{A}} & \{\text{A},\mathbf{\text{D}}\} & \xrightarrow[(0,0)]{\text{A}} & \{\text{A},\mathbf{\text{D}}\} & \xrightarrow[(0,0)]{\text{A}} & \{\text{A},\mathbf{\text{D}}\} & \xrightarrow[(0,0)]{\text{A}} & \{\text{A},\mathbf{\text{D}}\} & \\
\text{FIFO} & \{\bot,\bot\} & & \{\text{A},\bot\} & & \{\text{B},\mathbf{\text{A}}\} & & \{\text{B},\mathbf{\text{A}}\} & & \{\text{C},\mathbf{\text{B}}\} & & \{\text{D},\mathbf{\text{C}}\} & & \{\text{A},\mathbf{\text{D}}\} & & \{\text{A},\mathbf{\text{D}}\} & & \{\text{A},\mathbf{\text{D}}\} & & \{\text{A},\mathbf{\text{D}}\} & : (5,5)
\end{array}$$

$$\begin{array}{lccccccccccccccccccc}
\text{LRU} & \{\bot,\bot\} & \xrightarrow[(1,1)]{\text{A}} & \{\text{A},\bot\} & \xrightarrow[(1,1)]{\text{B}} & \{\text{B},\mathbf{\text{A}}\} & \xrightarrow[(0,0)]{\text{A}} & \{\text{A},\mathbf{\text{B}}\} & \xrightarrow[(1,1)]{\text{C}} & \{\text{C},\mathbf{\text{A}}\} & \xrightarrow[(1,0)]{\text{B}} & \{\text{B},\mathbf{\text{C}}\} & \xrightarrow[(1,1)]{\text{A}} & \{\text{A},\mathbf{\text{B}}\} & \xrightarrow[(1,1)]{\text{D}} & \{\text{D},\mathbf{\text{A}}\} & \xrightarrow[(0,0)]{\text{D}} & \{\text{D},\mathbf{\text{A}}\} & \xrightarrow[(0,0)]{\text{D}} & \{\text{D},\mathbf{\text{A}}\} & \\
\text{FIFO} & \{\bot,\bot\} & & \{\text{A},\bot\} & & \{\text{B},\mathbf{\text{A}}\} & & \{\text{B},\mathbf{\text{A}}\} & & \{\text{C},\mathbf{\text{B}}\} & & \{\text{C},\mathbf{\text{B}}\} & & \{\text{A},\mathbf{\text{C}}\} & & \{\text{D},\mathbf{\text{A}}\} & & \{\text{D},\mathbf{\text{A}}\} & & \{\text{D},\mathbf{\text{A}}\} & : (6,5)
\end{array}$$

$$\begin{array}{lccccccccccccccccccc}
\text{LRU} & \{\bot,\bot\} & \xrightarrow[(1,1)]{\text{A}} & \{\text{A},\bot\} & \xrightarrow[(1,1)]{\text{B}} & \{\text{B},\mathbf{\text{A}}\} & \xrightarrow[(0,0)]{\text{A}} & \{\text{A},\mathbf{\text{B}}\} & \xrightarrow[(1,1)]{\text{C}} & \{\text{C},\mathbf{\text{A}}\} & \xrightarrow[(1,0)]{\text{B}} & \{\text{B},\mathbf{\text{C}}\} & \xrightarrow[(1,1)]{\text{A}} & \{\text{A},\mathbf{\text{B}}\} & \xrightarrow[(1,0)]{\text{C}} & \{\text{C},\mathbf{\text{A}}\} & \xrightarrow[(1,1)]{\text{B}} & \{\text{B},\mathbf{\text{C}}\} & \xrightarrow[(0,0)]{\text{B}} & \{\text{B},\mathbf{\text{C}}\} & \\
\text{FIFO} & \{\bot,\bot\} & & \{\text{A},\bot\} & & \{\text{B},\mathbf{\text{A}}\} & & \{\text{B},\mathbf{\text{A}}\} & & \{\text{C},\mathbf{\text{B}}\} & & \{\text{C},\mathbf{\text{B}}\} & & \{\text{A},\mathbf{\text{C}}\} & & \{\text{A},\mathbf{\text{C}}\} & & \{\text{B},\mathbf{\text{A}}\} & & \{\text{B},\mathbf{\text{A}}\} & : (7,5)
\end{array}$$

$$\begin{array}{lccccccccccccccccccc}
\text{LRU} & \{\bot,\bot\} & \xrightarrow[(1,1)]{\text{A}} & \{\text{A},\bot\} & \xrightarrow[(1,1)]{\text{B}} & \{\text{B},\mathbf{\text{A}}\} & \xrightarrow[(0,0)]{\text{A}} & \{\text{A},\mathbf{\text{B}}\} & \xrightarrow[(1,1)]{\text{C}} & \{\text{C},\mathbf{\text{A}}\} & \xrightarrow[(1,0)]{\text{B}} & \{\text{B},\mathbf{\text{C}}\} & \xrightarrow[(1,1)]{\text{A}} & \{\text{A},\mathbf{\text{B}}\} & \xrightarrow[(1,0)]{\text{C}} & \{\text{C},\mathbf{\text{A}}\} & \xrightarrow[(1,1)]{\text{B}} & \{\text{B},\mathbf{\text{C}}\} & \xrightarrow[(1,0)]{\text{A}} & \{\text{A},\mathbf{\text{B}}\} & \\
\text{FIFO} & \{\bot,\bot\} & & \{\text{A},\bot\} & & \{\text{B},\mathbf{\text{A}}\} & & \{\text{B},\mathbf{\text{A}}\} & & \{\text{C},\mathbf{\text{B}}\} & & \{\text{C},\mathbf{\text{B}}\} & & \{\text{A},\mathbf{\text{C}}\} & & \{\text{A},\mathbf{\text{C}}\} & & \{\text{B},\mathbf{\text{A}}\} & & \{\text{B},\mathbf{\text{A}}\} & : (8,5)
\end{array}$$

(A) Example of a set of traces where LRU produces 5 observations and FIFO produces 1.

$$\begin{array}{lccccccccccccccccccc}
\text{LRU} & \{\bot,\bot\} & \xrightarrow[(1,1)]{\text{A}} & \{\text{A},\bot\} & \xrightarrow[(1,1)]{\text{B}} & \{\text{B},\mathbf{\text{A}}\} & \xrightarrow[(0,0)]{\text{A}} & \{\text{A},\mathbf{\text{B}}\} & \xrightarrow[(1,1)]{\text{C}} & \{\text{C},\mathbf{\text{A}}\} & \xrightarrow[(1,0)]{\text{B}} & \{\text{B},\mathbf{\text{C}}\} & \xrightarrow[(1,1)]{\text{A}} & \{\text{A},\mathbf{\text{B}}\} & \xrightarrow[(0,0)]{\text{A}} & \{\text{A},\mathbf{\text{B}}\} & \xrightarrow[(0,0)]{\text{A}} & \{\text{A},\mathbf{\text{B}}\} & \xrightarrow[(0,0)]{\text{A}} & \{\text{A},\mathbf{\text{B}}\} & \\
\text{FIFO} & \{\bot,\bot\} & & \{\text{A},\bot\} & & \{\text{B},\mathbf{\text{A}}\} & & \{\text{B},\mathbf{\text{A}}\} & & \{\text{C},\mathbf{\text{B}}\} & & \{\text{C},\mathbf{\text{B}}\} & & \{\text{A},\mathbf{\text{C}}\} & & \{\text{A},\mathbf{\text{C}}\} & & \{\text{A},\mathbf{\text{C}}\} & & \{\text{A},\mathbf{\text{C}}\} & : (5,4)
\end{array}$$

$$\begin{array}{lccccccccccccccccccc}
\text{LRU} & \{\bot,\bot\} & \xrightarrow[(1,1)]{\text{A}} & \{\text{A},\bot\} & \xrightarrow[(1,1)]{\text{B}} & \{\text{B},\mathbf{\text{A}}\} & \xrightarrow[(0,0)]{\text{A}} & \{\text{A},\mathbf{\text{B}}\} & \xrightarrow[(1,1)]{\text{C}} & \{\text{C},\mathbf{\text{A}}\} & \xrightarrow[(1,1)]{\text{D}} & \{\text{D},\mathbf{\text{C}}\} & \xrightarrow[(1,1)]{\text{A}} & \{\text{A},\mathbf{\text{D}}\} & \xrightarrow[(0,0)]{\text{A}} & \{\text{A},\mathbf{\text{D}}\} & \xrightarrow[(0,0)]{\text{A}} & \{\text{A},\mathbf{\text{D}}\} & \xrightarrow[(0,0)]{\text{A}} & \{\text{A},\mathbf{\text{D}}\} & \\
\text{FIFO} & \{\bot,\bot\} & & \{\text{A},\bot\} & & \{\text{B},\mathbf{\text{A}}\} & & \{\text{B},\mathbf{\text{A}}\} & & \{\text{C},\mathbf{\text{B}}\} & & \{\text{D},\mathbf{\text{C}}\} & & \{\text{A},\mathbf{\text{D}}\} & & \{\text{A},\mathbf{\text{D}}\} & & \{\text{A},\mathbf{\text{D}}\} & & \{\text{A},\mathbf{\text{D}}\} & : (5,5)
\end{array}$$

$$\begin{array}{lccccccccccccccccccc}
\text{LRU} & \{\bot,\bot\} & \xrightarrow[(1,1)]{\text{A}} & \{\text{A},\bot\} & \xrightarrow[(1,1)]{\text{B}} & \{\text{B},\mathbf{\text{A}}\} & \xrightarrow[(0,0)]{\text{A}} & \{\text{A},\mathbf{\text{B}}\} & \xrightarrow[(1,1)]{\text{C}} & \{\text{C},\mathbf{\text{A}}\} & \xrightarrow[(0,1)]{\text{A}} & \{\text{A},\mathbf{\text{C}}\} & \xrightarrow[(1,1)]{\text{B}} & \{\text{B},\mathbf{\text{A}}\} & \xrightarrow[(1,1)]{\text{C}} & \{\text{C},\mathbf{\text{B}}\} & \xrightarrow[(0,0)]{\text{C}} & \{\text{C},\mathbf{\text{B}}\} & \xrightarrow[(1,1)]{\text{C}} & \{\text{C},\mathbf{\text{B}}\} & \\
\text{FIFO} & \{\bot,\bot\} & & \{\text{A},\bot\} & & \{\text{B},\mathbf{\text{A}}\} & & \{\text{B},\mathbf{\text{A}}\} & & \{\text{C},\mathbf{\text{B}}\} & & \{\text{A},\mathbf{\text{C}}\} & & \{\text{B},\mathbf{\text{A}}\} & & \{\text{C},\mathbf{\text{B}}\} & & \{\text{C},\mathbf{\text{B}}\} & & \{\text{C},\mathbf{\text{B}}\} & : (5,6)
\end{array}$$

$$\begin{array}{lccccccccccccccccccc}
\text{LRU} & \{\bot,\bot\} & \xrightarrow[(1,1)]{\text{A}} & \{\text{A},\bot\} & \xrightarrow[(1,1)]{\text{B}} & \{\text{B},\mathbf{\text{A}}\} & \xrightarrow[(0,0)]{\text{A}} & \{\text{A},\mathbf{\text{B}}\} & \xrightarrow[(1,1)]{\text{C}} & \{\text{C},\mathbf{\text{A}}\} & \xrightarrow[(0,1)]{\text{A}} & \{\text{A},\mathbf{\text{C}}\} & \xrightarrow[(0,0)]{\text{C}} & \{\text{C},\mathbf{\text{A}}\} & \xrightarrow[(1,1)]{\text{B}} & \{\text{B},\mathbf{\text{C}}\} & \xrightarrow[(0,1)]{\text{C}} & \{\text{C},\mathbf{\text{B}}\} & \xrightarrow[(1,1)]{\text{A}} & \{\text{A},\mathbf{\text{C}}\} & \\
\text{FIFO} & \{\bot,\bot\} & & \{\text{A},\bot\} & & \{\text{B},\mathbf{\text{A}}\} & & \{\text{B},\mathbf{\text{A}}\} & & \{\text{C},\mathbf{\text{B}}\} & & \{\text{A},\mathbf{\text{C}}\} & & \{\text{B},\mathbf{\text{A}}\} & & \{\text{C},\mathbf{\text{B}}\} & & \{\text{C},\mathbf{\text{B}}\} & & \{\text{A},\mathbf{\text{C}}\} & : (5,7)
\end{array}$$

(B) Example of a set of traces where FIFO produces 4 observations and LRU produces 1.

FIGURE 2. Example of two sets of traces of memory blocks updating empty caches under cache algorithms LRU and FIFO. Each configuration of the cache shows the blocks cached between brackets, the symbol $\bot$ represents a line with no memory block cached in it and the bold face memory block is the next one to be evicted (the least recently used for LRU and the first to be cached for FIFO). Each update of the cache configuration is labeled with the accessed memory block (over the arrow) and the number of misses it produces on LRU and FIFO respectively (under the arrow). At the end of each trace we have the total number of misses the trace produces on LRU and FIFO respectively.