Security of Cache Replacement Policies under Side-Channel Attacks

Felix Schröder

March 6, 2018

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Ausführungen, die anderen veröffentlichten oder nicht veröffentlichten Schriften wörtlich oder sinngemäß entnommen wurden, habe ich kenntlich gemacht.

Die Arbeit hat in gleicher oder ähnlicher Fassung noch keiner anderen Prüfungsbehörde vorgelegen.

Ort, Datum

Unterschrift

Zusammenfassung (German Abstract)

Ein Computer Cache ist ein Teil des Speichers, auf den schnell zugegriffen werden kann. Um dies zu gewährleisten, kann der Cache allerdings nicht besonders groß sein. Die Entscheidungen, welche Daten im Cache gespeichert bleiben und welche ersetzt (engl. "replaced") werden, entscheiden sogenannte "Cache Replacement Policies". Diese Algorithmen werden in den allermeisten Fällen aufgrund ihrer Zeiteffizienz ausgewählt. In dieser Arbeit wird es allerdings um ihre Sicherheit gehen. Denn spätestens seit 2014 ([YF14]) ist klar, dass sogenannte Cache Side Channel Attacks eine Bedrohung für die Sicherheit moderner Systeme bedeuten. Bei solchen Angriffen geht es darum, geheime, z.B. kryptographische Informationen von einem Programm mithilfe eines anderen auszuspähen. Dafür müssen das ausgespähte Programm V und das Spähprogramm A auf den gleichen Cache zugreifen können. Dadurch, dass V den Cache benutzt, kann A, indem es auf den gleichen Cache zugreift, Informationen über den Cache und damit über V erhalten, z.B. wie viele Daten V benutzt oder unter Umständen sogar welche. Ist V ein Verschlüsselungsprogramm und das Cache-Verhalten vom geheimen Schlüssel ("key") abhängig, so lassen sich dadurch Rückschlüsse auf ihn ableiten. Spätestens die Entdeckung der SPECTRE+MELTDOWN Angriffe hat das Thema in den Fokus der Öffentlichkeit gerückt. Diese Masterarbeit beschäftigt sich im Folgenden hauptsächlich mit der Fragestellung:

Gibt es weitläufig genutzte Cache Replacement Policies, die sicher oder zumindest sicherer sind als andere?

Um diese Fragestellung zu beantworten, bedarf es zunächst einer mathematisch prägnanten Formulierung, insbesondere einer sinnvollen Definition von Sicherheit. Inspiriert vom sehr umfangreichen Wissen über die Zeiteffizienz dieser Algorithmen, entwickelt diese Arbeit ein Konzept, um Cache Replacement Policies zu vergleichen.

Unglücklicherweise konnten in dieser Arbeit nur Unvergleichbarkeitsresultate bewiesen werden. Diese sind allerdings umfangreich und illustrieren damit die Komplexität, die dieses Thema nach wie vor innehat.

Contents

1 Introduction

In a computer, a cache is a part of the memory which is designed to be accessed very fast. Due to the design constraints on computer memory, in order for this to be possible, caches are required to be somewhat small. So it is impossible to store all the data in the cache. The decision, when to store which data in the cache is done by a so-called "Cache Replacement Policy". It is called like this because the hard decisions, which it has to take, is which data to replace if it decides to store new data in a full cache. Caches are useful whenever one and the same data is needed multiple times in a rather short time frame. Most real-world programs exhibit such behavior, a phenomenon known as "temporal locality of reference". Instead of loading the data multiple times from the slow memory, it is copied into the cache and then loaded from there, when used again. A good cache replacement policy is therefore one that identifies correctly, when some block will not be used anymore (or for a long time, at least). However, data accesses of programs are not known in advance for the most part, they are "online".

Somewhat recently, a way to abuse this behavior of the cache to retrieve secret information from it has been developed (see e.g. [Pag02]). This way of gaining information is called a "side-channel" since the cache is not intended to convey any information at all (as opposed to other channels which are supposed to send information). It is just a tool to increase time efficiency of modern computers. In order for the attack to work, you need two programs, say A for "attacker" and V for "victim", which share a common cache. There are multiple ways for this to be possible. Either both are executed quasi-parallelly on the same processing unit, then the cache of this processing unit is used, or they are executed parallelly on a processor with multiple processing units. Those processors typically have some levels of caches which are proper to the processing units and one level (the so-called Last-Level-Cache or LLC, sometimes also L3 for level 3) that is common to all processing units. Then this LLC is used. Now, when program V uses the cache, it will load the blocks it uses to the cache. Since A uses the same cache, it can then observe, that the cache was changed, and, to a certain extent, in which way. Recent attacks like "Spectre" and "Meltdown" ([CS18]) show that cache side-channel attacks are still a prevalent threat to computer security to be prepared against. But how?

An easy approach to shield against cache side-channel attacks would be to not cache at all. But this negates all the advantages one gets from caching in the first place. The performance loss using this kind of measure is large. Therefore, as of recently, most cache side-channel attacks have been answered one by one by specific counter-measures. This is somewhat unsatisfying though. A more general kind of protecting against this type of attack would be better. In this thesis, we are taking a look at the role of cache replacement policies in this regard.

The question to be discussed is going to be:

Are there concurrent cache replacement policies which are preferable to others from the standpoint of security against side-channel attacks?

To answer that question, we will come up with a model which tries to measure the security of a cache replacement policy. This is not easy at all though. Unlike in the case of performance, it is not possible to answer the question if some cache replacement policy is secure, when confronted with a certain sequence of data accesses. The context of the program has to be taken into account. Therefore security is called a hyper-property of the cache replacement policy, instead of a property like performance. This will be explained in more detail when more terms have been introduced (TODO: reference when exactly). Since comparing concurrent policies with policies which do not cache to obtain higher security would be futile (the latter ones are not relevant for real-world caches because of the performance deficit), some performance constraints for policies are going to be formulated. Unfortunately, all of the results are negative, i.e. proofs that policies are not comparable in general. This illustrates the complexity of the subject as well as the small amount of knowledge we have about general protection against cache side-channel attacks.

The outline of this thesis is the following: First, the terms and their correspondent mathematical entities are introduced. Secondly, information from the much better known field of performance of cache replacement policies is gathered in order to analyze its relevance for the field of security of cache replacement policies. Afterwards, a measure of security is defined and motivated. After talking a little bit about the way the attacker is assumed to observe what the victim does, the performance requirements are formulated, backed up by theoretical results in the area. This makes it possible to discuss the results presented in this thesis. Lastly, I will try to do a short outlook on the current state of the subject and future work which could be done based on the results.

2 Nomenclature

At this point, it should be specified that real-world computers work with multiple cache sets, which means that there are separate cache memory spaces to load blocks from different parts of the memory. We will however only consider caches with 1 cache set, because the set-associative structure of caches is something that can be exploited on its own, without interference of cache replacement policies. To be specific, irrespective of the used cache replacement policies, attacks like Prime+Probe or Evict+Time described in [OST06], as well as Flush+Reload ([YF14]) will still work[1]. In order to analyze caches in a mathematical way, we will introduce the following notions: A *block* is a part of the memory which can

[1]For an overview, how these techniques work, see Appendix (Chapter 9)

be loaded into the cache. In real-world computers, blocks have a specific size defined by the caching algorithms, such that if data inside this block is needed, the whole block will be loaded. This is because real-world programs normally work on data which is saved next to each other. This phenomenon is called "spacial locality of reference". So loading the whole block will probably load some data which will be needed soon and it comes with only a small overhead in comparison to loading specific parts of the block only if they are accessed. We will denote the set of possible blocks with $\mathcal{B}$. The existence of such blocks makes defining, what a cache replacement policy does, a lot easier to grasp as well. A cache, equipped with a cache replacement policy, consists of a certain set of *cache states* $\mathcal{C}$ that encode not only which blocks are currently cached, but also some additional information needed for the cache replacement policy, for example time stamps when some blocks were added to the cache (FIFO: block which entered the cache first is evicted) or when they were last used (LRU: least recently used block is evicted) or numbers how often they were used (LFU: least frequently used block is evicted). The cache replacement policy now knows a certain *update function* $u : \mathcal{C} \times \mathcal{B} \to \mathcal{C}$ which will model the transition of the cache state, given that a certain block is accessed. It is, however, tedious to write down this update function for the cache replacement policies explicitly, which is why what the different replacement policies do exactly will be described in words in this thesis (similarly, but in more detail than above). The number of blocks a given cache can keep in its memory at the same time is called *associativity* of the cache (set). When executing a program, the only relevant information for the cache is the sequence of accessed blocks, the *access sequence*. This sequence may, however, depend on the program input. This is why we will define *programs* as random variables on the space of access sequences $\mathcal{B}^*$, even if they are deterministic. More about this will be specified in chapter 4, when talking about security. When analyzing cache replacement policies, regardless of whether we are dealing with performance or security, the most relevant part is if the accessed blocks are in the cache or not. Accessing a block in the cache will be called a *cache hit*, accessing a block which is not currently in the cache will result in a *cache miss*. Of course, from a performance standpoint cache hits are desirable whereas misses are not. From a security point of view, it depends on the program which one is desirable in which spot.

3 Inspiration from Previous Insights

As mentioned above, the development of cache side-channel attacks is a rather recent one. For the exploration of ways to shield against them, this is both a challenge and a chance. It is a chance, because it is most likely possible to find out new things nobody ever thought about, more likely than in older, more elaborated parts of mathematics or computer science. But it is also a challenge, since there are few things to build upon. It is harder to just take known concepts and elaborate them further, because there are so few of them. This is why this thesis is based on assertions made from the (by far) better

known field of performance of cache replacement policies. This section will present some of the results in that area, together with an explanation, what expectations arise for the work on security of those policies.

3.1 An optimal algorithm (?)

When talking about performance, the fact that algorithms are online is important. The idea about being "online" is that cache replacement policies normally do not know what blocks are going to be requested later, when deciding whether to evict some block or not. Obviously though, it would be very useful for them to know. To be precise, among the offline algorithms, there is an optimal algorithm, which is therefore called the *MIN-algorithm*, but sometimes goes by the name *Bélády's algorithm* as well, since it was found by the hungarian mathematician László Bélády in 1966 ([Bél66]). The algorithm works as follows:

On a cache hit, the requested block is loaded from the cache. No block is loaded into the cache, no block is evicted. On a cache miss, the requested block is loaded from memory. If the cache is already full, some block has to be evicted. The block to be evicted is chosen by calculating the next time every block in the cache will be needed in the sequence. If there is some block which will never be needed again, one of these blocks is evicted (which one is irrelevant, for the purpose of well-definition of the algorithm, let's choose the next block in FIFO order). If there is no such block, the one not needed for the longest time is evicted (This block is unique since in each request, only one block is requested). Let's prove optimality for this algorithm:

Theorem 3.1. *Let P be a cache replacement policy with associativity A. Let B be the Bélády's algorithm on a cache with associativity A. Let's assume further that both algorithms start with the same set of blocks β stored in their cache. Let s be an access sequence and let k be the number of cache misses of B on s. Then P loads at least k blocks from memory during s.*

Proof. For the purpose of this proof, if ever any algorithm loads a block and does not save it in the cache, this is considered both a load-into-cache and cache-evict action (of the same block).
The idea is that at any point, where P does not behave like B, we can change Ps behavior to the behavior of B while preserving or even lowering the number of misses. Let b be the first block evicted by B, which is still in the cache of P. Since P still has b in its cache, some other block in the current cache of B is not in the cache of P. Let us call this block b'. There are two cases, when b' was evicted last by P, either at the same time as b was evicted by B, or earlier. In case b' was evicted earlier, no other block was evicted by B at that time by construction. Assume B did evict b' as well. Then apparently B reloaded it into the cache later, but by definition of B, this can only be if b' was accessed. In that case, P would have had to load it as well which is a contradiction, because we considered the last time b' was evicted. So if P evicted b' earlier, then B did not evict any block at this time, which means P did not have to evict any

block at all and could just have kept b' in its cache until it has to evict some block. There is no reason not to keep b' in the cache a little longer. So we can assume that P or an algorithm at least as good as P evicted b' at the same time B evicted b. Now, due to the definition of B, this block b' is accessed, before b is accessed for the next time. Consider the algorithm P' which works just like P, with the only difference, that it evicts b instead of b' at this point. This change does not alter the cache behavior for blocks other than b and b'. This means, until P would have loaded b', because this event occurs before b is needed next. At this point P' should now load b instead of b'. Now the cache of P' looks the same as the one of P again, and P' acts the same as P for the rest of the sequence. Obviously, P' incurs the exact same amount of memory loads as P. If we alter the algorithm for all blocks which B evicts and which are in the cache of P, then in the end, we will end up with an algorithm P^*, that is not worse than P, with the additional property, that B evicts no blocks that P^* has in its cache. Since both start with the same cache, that also means that it is impossible that P^*s cache contains blocks, which Bs cache does not contain, except when P^* loads blocks into the cache before they are accessed. This is unnecessary though, because this way, they take up space in the cache before they have to. Loading it later takes the same time, so B has a miss, only if P^* has a miss or has preloaded the block earlier. So

$$k = \mathit{Misses}(B) \leq \mathit{Loads}(P^*) = \mathit{Loads}(P)$$

□

So now we know the optimally performant algorithm. Unfortunately, this algorithm is impracticable. In order to implement it, one would have to know the full sequence of data requests in advance, an impossible challenge on a computer with input elements. One might precompute the whole sequence of some program and then apply this algorithm, but the precomputing would take too much time, such that even this is not helpful. In terms of security, we have a similar problem. The optimal algorithm is impracticable as well, since it does not cache at all, which leads to a large performance loss. Now there are other optimal offline algorithms which do cache. For example, to shield against time-driven attacks, a program might precompute an over-approximation on its worst-case execution time and then wait for it, when it is done. This way, no observation can be made due to execution time. The quality of this kind of measure depends on the program though, since it is not always obvious to over-approximate worst-case execution time. For example, it is possible that the precomputation (which is always needed in the offline case) takes too long or is too imprecise, or that the worst-case execution time is very large in comparison with average-case execution time. Nevertheless, this measure can be good enough for some programs. It gave rise to a new field of cryptography, *constant-time* cryptography (an example is [Por17]). For example, there are constant-time variants of RSA and AES. But we would like to have program-independent security measures and this is not possible with this approach.

In the following section, the way the studies about performance handled the online/offline problem is introduced.

3.2 Competitiveness

The concept of competitiveness was introduced by Sleator and Tarjan in their paper about amortized efficiency of paging rules ([ST85]), even though they did not call it like this yet. The idea is, that online algorithms can not be as good as offline ones such as the MIN-algorithm. But if the algorithm is good enough, perhaps there is a constant $k \in \mathbb{R}$, such that an algorithm is not more than k times worse than MIN on all possible sequences of accesses. In that case, this algorithm is called k-competitive. A more formal way to state this intuitive property is this:

Definition 3.2. A cache replacement policy A is called k-competitive, if its number of misses m_A fulfills the following relation with the number of misses of MIN, denoted by m_{MIN}:

$$\exists c \in \mathbb{R} : \forall \sigma \in \mathcal{B}^* : m_A(\sigma) \leq k \cdot m_{MIN}(\sigma) + c \tag{1}$$

The following theorem gives a lower bound on the factor k for deterministic online policies:

Theorem 3.3. *Let P be an online deterministic k-competitive cache replacement policy with associativity A. Then*

$$k \geq A$$

Proof. Since P is online, its cache content can only depend on the accesses which have been done before and on the blocks in its starting cache. Therefore we can simulate the algorithm up to a certain point and then choose the next requested block dependent on the cache content at that time. Let $b_1, ..., b_A$ be the blocks in the starting cache of MIN and b_0 a block not in the starting cache of neither P nor MIN. The first requested block shall be b_0. Since not all of the blocks $b_0, .., b_A$ can be in the cache of P at once, it is always possible to choose the next block in a way, such that it is one of the b_i, which is not in the cache of P. We shall access blocks this way $A-1$ times. This means that P has now missed A times. MIN has only missed once though, since among those $A-1$ accesses after the access to b_0, at least one of the b_i was not accessed at all yet. One of these is removed by the MIN-algorithm at the time b_0 is accessed, because the other ones are re-accessed earlier. You can repeat this arbitrarily often. By always accessing one of the b_i which is not in the cache of P, P will miss the whole time. After a miss of MIN though, it can remove any of the blocks $b_0, ..., b_A$ except the one just accessed, which are A blocks in total, and it will never evict one of the next $A-1$ accessed blocks, because there is a better choice, namely to evict one of the ones that will not be needed for the next $A-1$ accesses. Therefore MIN misses at most one out of A times in the

sequence. Therefore, if σ is the sequence of accesses and $s = |\sigma|$ is its length , the definition of competitiveness (1) becomes:

$$\begin{aligned}\exists c \in \mathbb{R} : \forall s \in \mathbb{N} :\ s = m_P(\sigma) &\leq k \cdot m_{MIN}(\sigma) + c \leq k \cdot \left(\frac{s}{A} + 1\right) + c \\ &= \frac{k}{A} \cdot s + k + c \\ \Rightarrow \exists c \in \mathbb{R} : \forall s \in \mathbb{N} :\ c \geq \left(1 - \frac{k}{A}\right) s - k &\overset{s \to \infty}{\Longrightarrow} 1 - \frac{k}{A} \leq 0 \Rightarrow k \geq A\end{aligned}$$

□

Sleator and Tarjan ([ST85]) also showed that LRU and FIFO are best possible in the way that the inequality of the last theorem is strict. A way to prove this for FIFO is the following

Proposition 3.4. FIFO *with associativity A is A-competitive.*

Proof. There is one thing the two policies (MIN and FIFO) have in common: The last accessed block is part of both caches. Let $s \in \mathcal{B}^*$ be an arbitrary sequence of block accesses.

- Part 1:

  Consider some part s' of s, on which FIFO misses exactly A times, and which starts immediately after a miss for FIFO incurred by a block b. Now, due to the way FIFO works, none of the A misses is due to b, since b is only evicted at the Ath miss after it was loaded into the cache. Also, no block can be responsible for more than one miss, because again, between two misses of the same block for FIFO, there have to be A misses of other blocks. That means, that at the beginning of s', MIN has $A - 1$ blocks other than b in its cache, which are not all the blocks which incur misses for FIFO. Therefore one of the blocks which incurs a miss for FIFO has to be loaded into the cache of MIN as well and induces a miss, when it is.

- Part 2:

  Consider the following partitioning of our sequence s: First, we rewrite the number of misses that FIFO does on this sequence as $m_{\text{FIFO}} = n \cdot A + k$, where $n \in \mathbb{N}_0, k \in \{1, ..., A\}$. This is always possible on sequences on which FIFO has at least 1 miss, the other sequences cannot possibly be used as a counterexample for the competitiveness of FIFO, so we can just ignore them. Second, we partition the sequence into sequences $s_0, ..., s_n$, where s_0 is the part of the sequence up until the kth miss, and each of the other $s_i (i \in \mathbb{N}_{\leq n})$ is the next part of the sequence, up until the $i \cdot A + k$-th miss (for FIFO), with the exception that the last part s_n (Note that this part might be identical to s_0, because $n = 0$ is possible) also includes the rest of the sequence, when FIFO does not miss anymore, if there is such a

part. Then any of the subsequences $s_i (i \in \mathbb{N}_{\leq n})$ fulfills the requirements of part 1. Therefore MIN has a miss on any of these subsequences, i.e. $m_{MIN} \geq n$. That yields

$$m_{\text{FIFO}} = n \cdot A + k \leq A \cdot n + A \leq A \cdot m_{MIN} + A.$$

So FIFO is A-competitive with constant A.

□

Remark 3.5. The exact same proof also works for LRU. In doubt, read the original paper which features the proof for LRU ([ST85]).

While the concept of competitiveness works pretty well in the case of online algorithms designed for performance, since MIN actually still misses from time to time, the optimal offline algorithm for security does not reveal any information at all, so comparing with it would be equivalent to checking if only finitely many bits of the key are revealed, regardless of the length of the sequence. Since this length, as well as the number of key bits, can go to infinity, this seems unlikely. As already mentioned, the comparison with an algorithm which is not performant enough seems futile anyway, and this fact only strengthens this assumption. But it is possible to take competitiveness to the next level, which we will do in the next section.

3.3 Relative Competitiveness

Relative competitiveness is just one step further from competitiveness. Instead of comparing every algorithm to the known best, we can compare those algorithms between each other, using the same techniques. This allows us to give bounds on their quality in terms of another algorithms quality.

Definition 3.6. A cache replacement policy A is called *k-competitive* with respect to a cache replacement policy B, if its number of misses m_A and the number of misses of B denoted by m_B, fulfill:

$$\exists c \in \mathbb{R} : \forall \sigma \in \mathcal{B}^* : m_A(\sigma) \leq k \cdot m_B(\sigma) + c \tag{2}$$

A is called *non-competitive* with respect to B if there is no $k \in \mathbb{N}$, such that the above inequality holds for all possible sequences $\sigma \in \mathcal{B}^*$

Remark 3.7. The intuition behind this notion is that asymptotically, A is not more than c times worse than B.

Remark 3.8. For $k > l$, k-competitiveness implies l-competitiveness. This is why we are mostly interested of the infimum of all k, such that A is k-competitive w.r.t. B. Consequently, in the paper of Reineke et al.([RG08]), which contains a good overview on relative competitiveness for concurrent policies, sometimes $\infty = \inf \emptyset$ is used as a competitiveness factor to express non-competitiveness.

Next up, we want to prove such a competitiveness result in order to highlight the method, which is used to obtain such results in [RG08]. To that end, we define a new algorithm, the PLRU (Pseudo Least Recently Used) algorithm.

Definition 3.9. The *PLRU*-algorithm is an algorithm, which, additionally to the blocks stored in the cache, maintains a full binary tree, whose leafs contain exactly those blocks. So the associativity A of a cache (set) equipped by this policy has to be a power of 2. We define $k := \log_2 A \in \mathbb{N}$. For each non-leaf node of this tree, it saves a left-or-right bit which is interpreted to point at one of the children of that node. So

$$\mathcal{C} = \mathcal{B}^{2^k} \times \{0,1\}^{2^k-1}.$$

It acts in the following way:

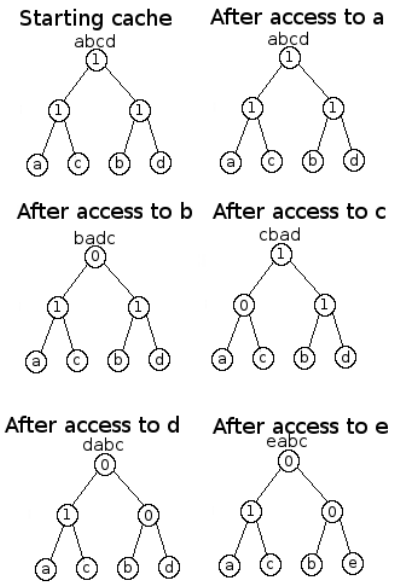

Figure 1: The 5 different kinds of accesses to a PLRU(4)-cache. Above the trees, the new eviction order is indicated, with the next block to evict written at the right.

- On a hit, the requested block is spotted as a leaf of the tree. Now, on the way from the root to this leaf, all pointers are set to face away from the way.
- On a miss, the algorithm follows the pointers from the root to the leaf, so at every non-leaf node, it will go right, if the bit is 1 and left, if the bit is 0. Then this block (the block all pointers point to) is evicted and replaced by the just evicted block. Similarly to the hit case, while on the way, all pointers are set to face away from the way we just took.

Remark 3.10. While the pointers are used to determine which block to evict, the most recently accessed block will have no pointer which points at it. It is the only block with this property (you could go the way from the root always not following the pointers and find this block). This property makes it the last block to be evicted if the next $A = 2^k$ accesses are cache misses. This property is why this algorithm is seen as a variant of least recently used. Due to the inherent structure as a binary tree though, some routines can be done in logarithmic instead of linear time. This is why this algorithm is sometimes used instead of straight LRU.

Proposition 3.11. *LRU is* 2*-competitive w.r.t. PLRU on a* 4*-associative cache. PLRU is non-competitive w.r.t. LRU on this cache, i.e. there is no* $k \in \mathbb{N}$*, such that PLRU is* k*-competitive w.r.t. LRU.*

Proof. First, an illustration of how PLRU works on a 4-associative cache might be appropriate, because its behavior is not quite as intuitive as the behavior of LRU. With four blocks in it, named a, b, c, d with d being the next one to evict, the following accesses are possible: An access to a, b, c, d or to an entirely new block, say e. Figure 1 illustrates the behavior of PLRU in those cases.

In order to analyze the behavior of PLRU and LRU on the same sequence, we will design a network which simulates all possible sequences of accesses. We will use the same starting cache as before, but for both policies. Figure 2 illustrates the possibilities here. The nodes of the network contain the current cache state for both policies, i.e. the cached blocks as well as their eviction order in case only misses will occur the next $A = 4$ times. Again, the rightmost block is evicted next. The cache state left of the placeholder | is the PLRU cache state, whereas the right one is the LRU cache state. The arcs of the network will be labeled by a block name. There will be an arc labeled with $b \in \mathcal{B}$ coming from a node with cache state $c_{PLRU}|c_{LRU}$ and going to a node with cache state $c'_{PLRU}|c'_{LRU}$, if and only if

$$u_{PLRU}(c_{PLRU}, b) = c'_{PLRU} \text{ and } u_{LRU}(c_{LRU}, b) = c'_{LRU}.$$

The interpretation of such a network is that, when a sequence of block accesses is read by the two policies simultaneously, we can consider it as a walk through the network. In figure 2 on the left, apparently one of the nodes contains the exact same cache states as before. So we do not need to distinguish that node from the original one as the behavior of the two policies only depends on the cache state and the accessed block. Also, in two other cases, the cache state changes, but both policy still have the same cache state, i.e. the same blocks are stored in it and they have the same eviction order. The naming of a, b, c, d was arbitrary though. Both policies do not differentiate their behavior as a function of the accessed block but only as a function of the position the block has in the cache at the moment. So after each access, we can rename the blocks arbitrarily. We would prefer to do this in an orderly manner though, so we will rename the blocks always in a way, such that the eviction order for LRU is *abcd*. The renaming has to be consistent within both cache states, so if we decide to rename b to a and a to b, so that LRU eviction order is *abcd* again, we need to do this for PLRU as well. If there are any blocks in the cache of PLRU, which LRU does not contain, we will name them e, f, g, h in eviction order. This will reduce the size of the network significantly. If we would differ between different nodes of exact same cache states, the network would be infinite. Even after merging those nodes, when we do not rename blocks at all, the size of the network depends on the number of blocks $|\mathcal{B}|$ available, which is very large in comparison to A. We will call this method network folding, since all that is

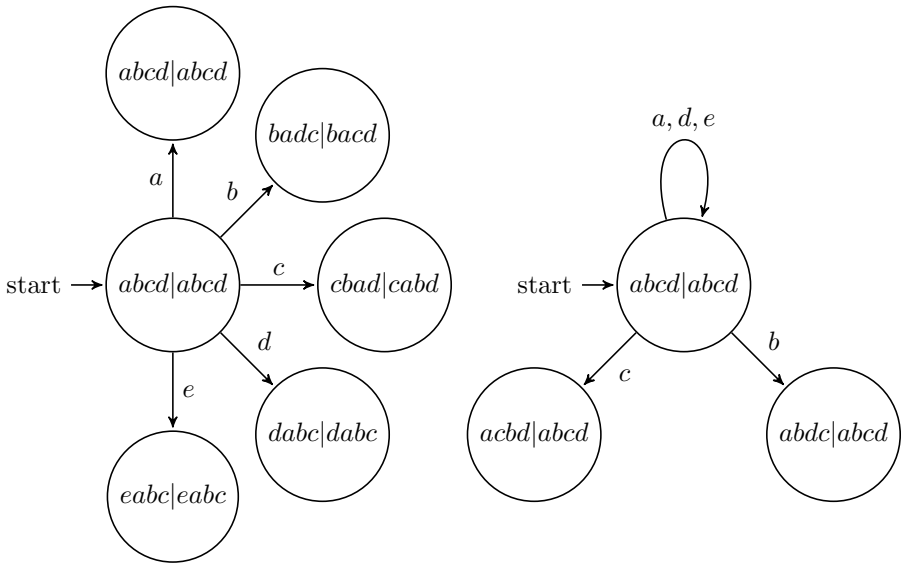

Figure 2: Start of the PLRU(4) vs. LRU(4) network, before and after network folding simplification

done is merging nodes, which can be visualized by folding the network in a way such nodes are folded onto each other. Figure 2 also illustrates how the part of the network around the node *abcd*|*abcd* looks like. After this simplification, the cache state of LRU is always *abcd*, so we can just omit it. The whole network looks like in figure 3.

Since we keep renaming our blocks, the labeling of the edges becomes counterintuitive. For example, when going through the cycle of the three rightmost states, it seems possible that we always access the same block c. Accessing c means accessing the third block in the cache of LRU though. Since this block changes at every access if we access it, we will never access the same block twice in a row here. In fact, accessing the most recently accessed block does not change the cache at all for both policies. Also, if we want to proceed proving competitiveness of LRU w.r.t. PLRU and non-competitiveness vice-versa, we will need to include something indicating when the different policies will have their misses and hits. So we will update these labels. For example, when we are in the cache state *abec*, an access to a, b or c will yield a hit for both policies, which we will denote by hh in the labeling (remember the cache state of LRU is always *abcd*). Accessing d will yield a hit for LRU, but a miss for PLRU, so we will call that mh. Similarly, an access to e yields hm and an access to the block f which is not contained in either cache results in mm. Using this kind of labeling we can also merge some of the block accesses such as b and c in the case of *abec*, because both yield the same resulting cache state and the same hits or misses for both policies. The resulting and final network is presented in

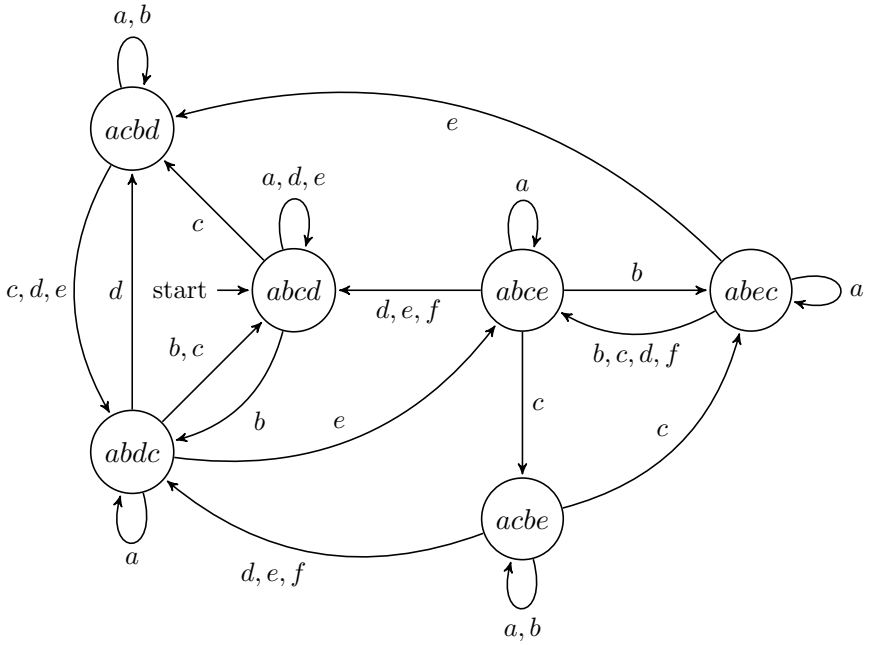

Figure 3: The full PLRU(4) vs. LRU(4) network

figure 4. With this network, it is possible to evaluate competitiveness of the two policies w.r.t. each other.

So when a sequence of block accesses is given to the two policies, this is equivalent to a walk through the network in figure 4. It is then possible to read the sequence of hits and misses for the two policies. The non-competitiveness result is now easy to obtain. It is possible to go from the start to the node *abdc* below on the left, then to the upper right to the node *abce*, incurring a hit and a miss for both policies. If we then go from *abce* to *abec* and back, we are able to incur a miss and a hit for PLRU while only incurring hits for LRU. So if we do this n times, the total number of misses of PLRU is raised up to $n+1$ while the number of misses of LRU stays at 1. So

$$\forall n \in \mathbb{N} : \exists \sigma \in \mathcal{B}^* : m_{PLRU}(\sigma) > n \cdot m_{LRU}(\sigma).$$

Assume PLRU is k-competitive w.r.t. LRU for some k, then with σ_n being the sequence described above:

$$\begin{aligned} &\exists k, c \in \mathbb{R} : \forall \sigma \in \mathcal{B}^* : m_{PLRU}(\sigma) \leq k \cdot m_{LRU}(\sigma) + c \\ \Rightarrow &\exists k, c \in \mathbb{R} : \forall n \in \mathbb{N} : n < m_{PLRU}(\sigma_n) \leq k \cdot m_{LRU}(\sigma_n) + c = k + c \end{aligned}$$

This is impossible. The number $k + c$ cannot be bigger than all naturals $\lightning$.

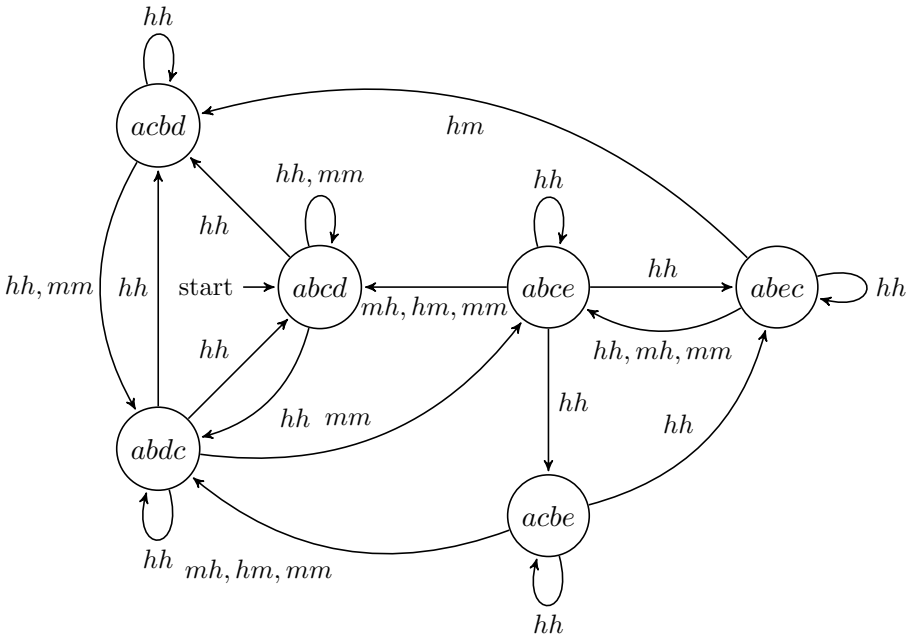

Figure 4: The full PLRU(4) vs. LRU(4) network with the new labels

The other result, the 2-competitiveness of LRU w.r.t. PLRU, is a little bit harder to obtain. It is not possible to find a similar loop in the network in figure 4, which incurs misses for LRU without incurring them for PLRU. Nevertheless, PLRU can have hits, when LRU has misses and there are some cycles with more misses for LRU than PLRU as well. An example is going from the start to the bottom left (*abdc*) then to the upper right (*abce*), then back to the start again, incurring two misses and one hit for LRU, while leaving PLRU with only one miss and two hits. Similarly to the proof above, it is now possible to prove that LRU is not k-competitive w.r.t. PLRU for any $k < 2$. We claim that this example can be generalized to all cycles.

Claim 3.11.1. *Let n_C be the number of misses of some replacement policy A, and m_C be the number of misses of some other replacement policy B on a certain cycle C in their common cache state network. Then the "cycle ratio" $\frac{n_C}{m_C}$ describes a lower bound on the competitiveness of A w.r.t. B.*

Proof. Assume A is k-competitive w.r.t. B and $k < \frac{n_C}{m_C}$. Let further

$$\epsilon := n_C - km_C > 0.$$

Again, we will proceed by constructing a sequence of access sequences $(\sigma_n)_n$, which produces too many misses for A such that this can be true. We will do so by first accessing blocks such that we go to some node of the cycle. This will

take us at most N accesses, if N is the number of nodes in the network (so a constant w.r.t. n). We will name this first part of all sequences σ_0. Then for σ_n, we will go around the cycle n times. This yields the following:

$$\begin{aligned} m_A(\sigma_n) &= m_A(\sigma_0) + nn_C \geq nkm_C + n\epsilon \\ m_B(\sigma_n) &= m_B(\sigma_0) + nm_C \leq nm_C + N \\ nkm_C + n\epsilon \leq m_A(\sigma_n) &\leq km_B(\sigma_n) + c \leq nkm_C + kN + c \\ \Rightarrow n &\leq \frac{kN + c}{\epsilon} \quad \lightning \end{aligned}$$

Once again, no constant may be bigger than all naturals. □

We are therefore interested in the maximum cycle ratio of any cycle in our network. It turns out, that it is indeed 2, which can be checked by hand or by computer still rather easily. Table 5 contains all cycles with at least one miss for any policy and their ratios for comprehensibility.

$abcd$	1
$abcd - abdc - abce$	$\frac{1}{2}$
$abcd - abdc - abce$	1
$abcd - abdc - abce$	2
$abcd - acbd - adbc$	1
$abcd - acbd - adbc - abce$	$\frac{1}{2}$
$abcd - acbd - adbc - abce$	1
$abcd - acbd - adbc - abce$	2
$abcd - acbd - adbc - abce$	$\frac{2}{3}$
$abcd - acbd - adbc - abce$	1
$abcd - acbd - adbc - abce$	$\frac{3}{2}$
$abdc - acbd$	1
$abdc - abce - acbe$	$\frac{1}{2}$
$abdc - abce - acbe$	1
$abdc - abce - acbe$	2
$abdc - abce - abec - acbd$	$\frac{1}{2}$
$abdc - abce - abec - acbd$	$\frac{2}{3}$
$abce - abec$	0
$abce - abec$	1
$abce - acbe - abec$	0
$abce - acbe - abec$	1

Table 5: All cycles and their ratios for LRU-competitiveness w.r.t. PLRU

Of course, when comparing policies with a larger cache or more complicated transition procedures, the network becomes larger and the maximum cycle ratio is harder to determine. Nevertheless, this approach gives the problem a concept of duality which makes it an even more interesting algorithmic problem. We even have strong duality here, i.e.

Claim 3.11.2. *The maximum cycle ratio coincides with the infimum of all possible competitiveness factors. For non-competitiveness, a cycle with a ratio $\frac{n}{0}$ for some $n \in \mathbb{N}$ is needed.*

Proof. Consider a set of sequences which proves a lower bound l on this competitiveness. Then this sequence does so by containing a sequence $(\sigma_n)_n$ of access sequences for any number k lower than this bound, such that $m_A(\sigma_n) - km_B(\sigma_n)$ goes to infinity. This set of sequences can be seen as a set of walks through our network. Now let k be less than this lower bound, then we can find such a sequence $(\sigma_n)_n$. Now, within these sequences, the number of misses of A certainly has to go to infinity, so the length of the sequences has to go to infinity as well. Since the only thing we are interested in is the total misses for any of the policies on the walks through the network, we can reorder the walks,

if we want. Therefore, we will do a path-cycle-decomposition[2] of the walks. Basicly, at the first point, where a node is visited which was visited before, we will cut off the cycle, since he last visited the node, out of the walk and keep it separate from the walk. If we do this to any walk multiple times, until he does not visit any node twice, we will come up with a rest-walk, which does not visit any node twice (path) and some cut-off cycles. Since the length of the sequences and thus the length of the paths goes to infinity and a path (that does not visit any node twice) has limited length, there has to be an unbounded amount of the walk going into the cycles. Now assume all cycles have cycle ratios $r < k$. Then, within the cycles, $m_A - rm_B$ is non-positive, thus it is bounded. Also, the path is finite, so here $m_A - rm_B$ is bounded as well. This is a contradiction to the assumptions about the sequence (σ_n). So there has to be a cycle with ratio $r \geq k$. But since $k < l$ was arbitrary, and there are only finitely many cycles, $r \geq l$ as well. □

Table 5 thus proves our proposition.

□

For performance, we know some algorithms competitive w.r.t. the best possible algorithm (MIN), so this implies competitiveness w.r.t. any other algorithm as well, so there is a natural upper bound on the competitiveness factor. Above, we found out, that w.r.t. PLRU, LRU is even more competitive, so the competitiveness factor went down from 4 to 2. For PLRU, proposition 3.11 proves that it is not competitive w.r.t. LRU, therefore not to all policies, in particular also not to MIN. For security, we do not have an upper bound on the competitiveness factor as well, so the results could be anything. They might be non-existent as well. Also, some algorithm might be comparable to a similar one and yielding better results than our a-priori bound like LRU did. Finding out such competitiveness factors is the main subject of this thesis.

3.4 Random Policies

Up until this point, the discussion was only about deterministic policies. We showed that, using deterministic, online policies, no competitiveness factor lower than the associativity A is possible. The question one might ask is:

Can we do better by including randomness in our policies?

And the result is: Yes!

To show this result, we will need to introduce a new policy, the *marking algorithm* of Fiat et al.([FKL+91]). It is called *marking*, because it maintains a list of marked blocks M, which stay in the cache for the next time.

Definition 3.12. The *marking algorithm* on a cache (set) of size A is defined as a cache replacement policy which retains a list of marked blocks in addition

[2] Example for path-cycle-decomposition in our network: $abcd - abdc - abdc - abce - abec - acbd - acbd - abdc - abce - acbe$ reduces to the path $abcd - abdc - abce - acbe$ and the cycles $abdc$,$acbd$,$abdc - abce - abec - acbd$

to the list of cached blocks, formalized:

$$\mathcal{C} = \mathcal{B}^A \times \{0,1\}^A$$

The binary variable is an indicator, if the corresponding block at the same index of the cache is currently marked.

- When a block incurs a hit, it is marked.
- When a block b incurs a miss, the following things happen:
  1. An unmarked block u already in the cache is determined uniformly at random. If there is none, then all the blocks are unmarked and u is chosen uniformly at random among all blocks in the cache.
  2. u is evicted.
  3. b is loaded into the cache.
  4. b is marked.

One can see the marking algorithm as a randomized version of LRU, since the most recently accessed blocks are most probably marked and therefore safe from eviction. It is possible though, that the most recently accessed block was the last non-marked block in the cache, and that it is evicted now, since the list is reset. So in some cases, the algorithm works a lot differently than LRU, but these cases are rather improbable. The reason we defined this algorithm is, that it has a (asymptotically) significantly lower (expected) competitiveness factor than any deterministic algorithm:

Proposition 3.13. *The marking algorithm M, working on an A-associative cache (set), is expected $2H_A$-competitive, where H_n is the n-th harmonic number, defined by:*

$$H_n = \sum_{i=1}^{n} \frac{1}{i}$$

Proof. Let $s \in \mathcal{B}^*$ be arbitrary. Then we can partition s by considering the marking phases of M. Let s_1 be the largest subsequence of s containing the first request, such that no blocks are unmarked. The next request after s_1 (if it ends) has to be a miss, because it leads to an unmarking of blocks. The subsequence s_2 is then defined as being the longest subsequence containing this request, such that no blocks are unmarked in any other step of the sequence. The further subsequences $s_3, ..., s_k$ are defined in the same way, such that s is fully partitioned. Note that the fact, that there is just one unmarking of exactly A blocks in each phase (except the first), together with the property that at the start and end of each phase, all blocks are marked (except for the first and perhaps for the last one), means, that during $s_i, i \in \{2, ..., k-1\}$, exactly A different blocks are marked. Since a block is marked if and only if it is accessed, during each phase (except the first and last one), exactly A different blocks are accessed.

Now let $i \in \{2, ..., k-1\}$ be arbitrary. We will call a block *clean*, if it has not been requested in neither s_{i-1} nor s_i until this point and *stale*, if it has been accessed in s_{i-1}, but also only as long as it has not been requested in s_i. This yields that initially in s_i, the set of marked blocks B_m, the set of stale blocks B_s and the set of blocks in the cache of M, B_M, coincide. The set of clean blocks B_c is their complement.

Let l_i be the number of requests to clean blocks in $s_i, d_i = |B_{MIN} \setminus B_M|$ be the number of blocks in the cache of MIN not in the cache of M at the beginning of phase s_i and $d'_i = d_{i+1}$ the number of such blocks at the end of s_i. Now MIN misses at least $l_i - d_i$ times on s_i, since MIN only has at most d_i of the accessed clean blocks in its cache at the start, it has to load the other ones (if there are any, i.e. if $l_i > d_i$). Also, since all requested blocks in a phase are marked, and all marked blocks at the end of the phase were accessed in that phase at some point, the fact that d'_i blocks in MINs cache are not marked in the end, means, that they were not accessed in this phase. Since MIN does not load blocks into the cache which are not accessed, those blocks had to be in the cache for the whole phase. But, since A different blocks (the marked blocks at the end of the phase) have been accessed, and MIN only used $A - d'_i$ slots of its cache, it has to miss at least d'_i times. We conclude:

$$m_{MIN} \geq \sum_{i=2}^{k-1} \max(l_i - d_i, d'_i) \geq \sum_{i=2}^{k-1} \frac{l_i - d_i + d_{i+1}}{2} = \frac{d_k - d_2 + \sum_{i=2}^{k-1} l_i}{2} \geq \frac{l}{2} - \frac{A}{2}$$

Here $l = \sum_{i=2}^{k-1} l_i$ denotes the total number of requests to clean blocks in the phases $s_2, ..., s_{k-1}$. Note that this proves, that the amortized cost of MIN is at least linear in the total number l of requests to clean blocks.

We now want to bound the amortized cost of M from above by lH_A. Doing this would yield

$$m_M \leq lH_A + c \leq 2H_A \cdot m_{MIN} + AH_A + c.$$

Consequently, this is a way of showing $2H_A$-competitiveness of M.

Consider some phase $s_i, i \in \{2, ..., k-1\}$ again. Since l_i of the requests are requests to clean blocks, there have to be exactly $A - l_i$ requests to stale blocks. This is because we know by construction that exactly A different blocks are accessed, and all of them either have been accessed in s_{i-1} (stale case), or they have not (clean case). At any point after the first reqeust to any of these blocks, the block is marked, and M will therefore incur a hit. We would like to calculate the probability, that these stale blocks are still in B_M. This probability depends on the number of requested clean blocks until this point, c, and the number of stale vertices at the moment, $|B_s|$. During any of these c requests, one of the stale blocks was evicted, so the probability that any of these is not in the cache is $\frac{c}{|B_s|}$. Since all requests to clean blocks will incur a miss for M anyway, to do a worst case analysis, we assume, that the clean blocks are accessed first in

order to obtain the maximum miss probability for the stale blocks. Remember we start the phase with A stale blocks. However, every requested stale block lowers the number of such blocks by 1, so the expected number of misses on requested stale blocks is at most (j is the number of accessed and therefore non-stale vertices which started the phase marked.)

$$\sum_{j=0}^{A-l_i-1} \frac{l_i}{A-j} = \sum_{n=l_i+1}^{A} \frac{l_i}{n} = l_i(H_A - H_{l_i}).$$

If $l_i \geq 1$, the total number of expected misses on s_i is at most (otherwise it would be 0)

$$l_i + l_i(H_A - H_{l_i}) = l_i(H_A - H_{l_i} + 1) \leq l_i H_A.$$

So we obtain the wanted result, since in both phases s_1, s_k, not more than A blocks are marked, such that M incurs not more than $2A$ misses here.

$$m_M \leq 2A + \sum_{i=2}^{k-1} l_i H_A = l H_A + 2A$$

□

So it is possible to get (significantly) better results when using randomized algorithms. In part, the proof also shows the intuition why this is possible. Consider the performance competitiveness of an algorithm as a game in a game theoretic sense, where one of the players, say A, chooses an algorithm, then the other player, say B, has to choose an access sequence. A's goal would be, that his algorithm performs well on the given sequence and B's goal would be the opposite. In some sense, the fact that A chooses a randomized algorithm denies information to B, because if A would choose a deterministic algorithm, B would know exactly how it would react to any sequence. Given a randomized algorithm though, multiple cache configurations are possible with different probabilities, so it is a lot harder to find a block which is not hit for sure, sometimes you have to settle for the block which is just improbable to be hit. But still, applying a randomized algorithm to a sequence will give you exact times, when the algorithm misses, and you can then show, that the number of these misses still has to be at least as big as the number of misses of Bélády's algorithm. The proof of optimality still works. So the optimal algorithm is still optimal. Furthermore, there is still a lower bound on competitiveness of random algorithms, it is H_A. So the marking algorithm is best possible up to a constant. This is also proven in the same paper([FKL$^+$91])

However, the more interesting question for us is, if randomized policies can also be used to improve security of cache replacement policies. This question will be addressed in the "Summary" of our results in section 7.3.

4 Measuring Security

There are certainly different ways to define security of cache replacement policies. It should firstly be noted that security is a property too non-linear and complex to just map it to the real axis. The security of a cache replacement policy depends largely on the chosen program. Nevertheless, we will try to analyze the security of replacement policies in general. But however, what is security? We would consider a cache replacement policy "secure", if it disables the possibility for an attacker, which uses the same cache, to derive information about other programs. This gain of information involves the information theoretic notion of entropy. The entropy of a random variable is the amount of information gained, when the outcome of the random experiment is known. There are different ways to measure this amount of information. In our case, we decided to use MIN-entropy because of a worst-case analysis. It describes the probability of an attacker guessing the key of a cryptographic procedure in one guess. Now this entropy is important because there is no way of distinguishing such an attacker, guessing correctly, from a legitimized user who knows the key. In some instances, it might be possible for an attacker to guess the key some more times, because even users who know the key might have an error while transmitting it. We will omit that case though, because it can be fought against using other methods. In formulas, for a given random variable $X : \Omega \to \mathcal{X}$, where $(\Omega, \mathbb{P})$ is the probability space underlying X:

$$H_\infty(X) = -\log \max_{x \in \mathcal{X}} \mathbb{P}(X = x)$$

It is to be noted, that the information gain, when the outcome of a random experiment is known, is the higher, the lower the a-priori (without information) percentage of the attackers guess is. This is reflected by the $-\log$ in front of this percentage. Another useful property and reason to use the logarithm is, that the entropy is a non-negative real number, which is 0, if and only if no information is gained at all and which can be arbitrarily big, if the key space is appropriately big. It also lines up well with other entropy measures this way.

To describe the resiliency of a cache replacement policy, we want to measure how much of the information about the key "leaks", i.e. is discovered by the attacker during the side-channel attack. Usually in information theory, the choice of the secret key is modeled as an assignment of a random variable to a certain value. Let $A \in \mathcal{P}$ be a cache replacement policy and K a random variable, which represents the key, chosen at random from a set $\mathcal{K}$, and let its underlying probability space be $(\Omega, \mathbb{P})$. Let further $O : \mathcal{P} \times \mathcal{K} \to \mathcal{O}$ be the observation an attacker can make from the cache, when leading a cache side-channel attack against a cache equipped by a policy $A \in \mathcal{P}$, assuming the key is $k \in \mathcal{K}$. Since K is a random variable on $\mathcal{K}$, $O(A, K)$ or O_A for short, is a random variable on $\mathcal{O}$. This means, we can measure the probability of guessing the key K in one guess without any observations on one hand (denoted by $H_\infty(K)$), and the estimated probability of guessing the key in one guess, if you have the side-channel observation O_A on the other hand (denoted by $H_\infty(K|O_A)$. A

formula for the maximum leakage of an algorithm could look like this:

$$\begin{aligned} L(A) &:= \sup_{\substack{K:\Omega\to\mathcal{K} \\ \mathcal{K} \text{ finite set} \\ \Omega \text{ probability space}}} I_\infty(K, O_A) \\ &:= \sup_{\substack{K:\Omega\to\mathcal{K} \\ \mathcal{K} \text{ finite set} \\ \Omega \text{ probability space}}} H_\infty(K) - H_\infty(K|O_A) \end{aligned} \tag{3}$$

Unfortunately, this measure is mostly infinite everywhere. Roughly, even if only 1% of key bits leak, you can make this measure arbitrarily big by making the number of key bits $\log|\mathcal{K}|$ to guess arbitrarily high, because this increases the number of leaked bits unboundedly as well. To put this into perspective, we will compare different policies using the measure. This comes down to the concept of relative competitiveness introduced in the last Chapter. Thus we will be comparing different, known to be efficient policies with each other, as mentioned, because comparing with a low-performing security-optimal policy would make no sense. An intuitive definition of (relative) competitiveness would be the following:

Let A, B be cache replacement policies and $k \in \mathbb{R}$. A is called k-competitive with respect to B, if

$$\exists c : \forall \text{ finite sets } \mathcal{K}, \forall K : \Omega \to \mathcal{K} : I_\infty(K, O_A) \leq k \cdot I_\infty(K, O_B) + c. \tag{4}$$

However, this measure is not perfect either. It is hard to calculate, especially quantifying over all possible key distributions is a mess. In cryptographic applications it is common to use equidistributed keys, since this makes the a-priori guess probability the lowest for any given number of possible keys. Since the original entropy is largest in this case, the leakage has the possibility to be big as well. An upper bound on leakage, which represents this intuition, by being an equality, if the key is equidistributed, is this (see Theorem 1 of [KS10]):

$$\forall K : \Omega \to \mathcal{K} : I_\infty(K, O_A) \leq \log \sum_{o\in\mathcal{O}} \max_{k\in\mathcal{K}} \mathbb{P}(O_A = o|K = k) \tag{5}$$

This yields our final notion of security competitiveness.

Definition 4.1. A cache replacement policy A is *k-competitive w.r.t.* a policy B, if there is some c, such that for any finite set $\mathcal{K}$ and K equidistributed on $\mathcal{K}$:

$$\begin{aligned} &\log \sum_{o\in\mathcal{O}} \max_{k\in\mathcal{K}} \mathbb{P}(O_A = o|K = k) \leq k \log \sum_{o\in\mathcal{O}} \max_{k\in\mathcal{K}} \mathbb{P}(O_B = o|K = k) + c \quad \Leftrightarrow \\ &\sum_{o\in\mathcal{O}} \max_{k\in\mathcal{K}} \mathbb{P}(O_A = o|K = k) \leq c \left(\sum_{o\in\mathcal{O}} \max_{k\in\mathcal{K}} \mathbb{P}(O_B = o|K = k) \right)^k \end{aligned} \tag{6}$$

Non-competitiveness means, that there is no k, such that inequality (6) holds.

Remark 4.2. A commonly used counterexample for competitiveness will be a sequence of programs, which leak no information at all using policy B, while leaking more and more using policy A.

Remark 4.3. A special case of this competitiveness is the case, that one of the policies is deterministic. If O_A depends on K deterministically, then the probabilities $\mathbb{P}(O_A = o|K = k)$ can only be 1 or 0. We can assume, that every observation o can be made, otherwise it would not be necessary to include it in $\mathcal{O}$. Therefore there is some k such that the probability is 1 for each o, yielding:

$$\log \sum_{o \in \mathcal{O}} \max_{k \in \mathcal{K}} \mathbb{P}(O_A = o|K = k) = \log \sum_{o \in \mathcal{O}} 1 = \log |\mathcal{O}| \tag{7}$$

This measure is therefore our measure for deterministic policies. Note that it lines up well with the intuition. The more different observations attackers can make, the more information they can gain.

5 Attacker Model

In the last section, we just considered some abstract set $\mathcal{O}$ of observations. Now there are multiple models about what the attacker in a cache side-channel attack can actually observe. One of the original models of an attacker's observation is the *time-driven* model ([Koc96]). Since cache misses take a lot more time than cache hits, it is normally possible to derive the number of hits and misses from the execution time. As a lot of secret keys are used during authentification procedures and as those try to do their job as fast as possible, this is a realistic model. Sometimes though, especially in the case mentioned before where multiple programs are executed quasi-parallelly on the same core or where two programs are executed parallelly on two cores sharing the last-level-cache, it is possible for the attacker to measure time multiple times during the execution of a program, up to a maximum of information, when every hit and miss in succession is known. This is known as the *trace-driven* attacker model, because the attacker can observe the whole sequence of hits and misses. Another way to observe the whole sequence of hits and misses is to keep track of power consumption or of the traffic. On a miss, the power-consumption or traffic goes up. On a hit, power-consumption or traffic is comparatively low. A real-world example for this is a credit card reader, which measures the traffic, as a credit card is read by a cash machine. There is another attacker model as well (see e.g. [KB07] for a well-elaborated *access-driven* attacker model), but these ones are the ones which are directly used in this thesis, in part because some of the access-driven models require the set-associative structure of the cache, which we will be ignoring.

To be precise, we will use the time-driven model. It is very easy to apply our results to a sequence-driven attacker as well though. In order to do this, since we only achieve negative results, it is only necessary to make sure, that no information leaks even in this case. The fact, that the amount of information

obtained by a time-driven attacker is arbitrarily big, will imply immediately, that the same holds for a trace-driven attacker. This is done easily, one simply has to take a look at the first part of Lemma 7.5 and convince oneself, that indeed, no information leaks for such an attacker.

6 Minimum Standards

6.1 Two Performance Requirements

In the introduction, we mentioned, that an analysis purely based on security is futile. While analyzing how the disadvantages for security brought by caches can be minimized, we want to keep in mind the advantages. Caching can reduce the running time of real world programs by a lot. Therefore, when talking about cache replacement policies, we will make a few minimum requirements they have to fulfill. These are aimed at only considering policies, which might be competitive to concurrent policies from a performance stand point. We will add an explanation why this requirement makes a policy more performant than policies, which do not fulfill it.

Definition 6.1. A cache replacement policy is said to be *lazy*, if it does not load any blocks into the cache, which are not accessed.

This requirement covers the fact, that an algorithm might load the whole cache, whenever a block is accessed. This can not only decrease the performance to a level, where not caching might be comparable, but even further. If we just look for the accessed block in main memory, this takes about the time we load one block into the cache from main memory, but if at each access, we would load multiple blocks into the cache, performance can even get multiple times worse. There are some algorithms used in practice, which do that to a certain extent, but Manasse et al. ([MMS90]) proved, that any such algorithm has a lazy variant, which is at least as performant as the original one. While being lazy is a useful requirement and backed up by theoretical results, it does not require the policy to use the cache at all, since it only forbids loading blocks to the cache under certain circumstances. It does not force the algorithm to cache anything. To address this, we will need a second requirement:

Definition 6.2. A cache replacement policy is called *caching*, if it loads a block into the cache at any time it is requested and not already in the cache.

If we want to harness the advantages of caching, we would like to have a notion of what they are. The concept we are looking for is the so-called locality of reference. There are two different kinds of this locality, spacial locality and temporal locality of reference. Real world applications often have either of these properties, mostly even both. The former means, that data, which is not far from each other, is often used together or one after the other. Caches exploit that behavior of concurrent programs by caching not only the exact data requested, but a little bit around this data as well. This is why the data is stored in units of

blocks, not of bytes or even bits. The latter means, that one and the same data is often used more than once, and if so, the times, when it is used, are normally close to each other. Both of these properties are exploited by a caching cache replacement policy, because only by actually caching the blocks, you can reduce the access time to the data, if either the same data is used again after a short period of time or some other section of the data block is used just afterwards. This is a rather common principle of caching in some mathematical models also. The assumption is there, that you have to store the block somewhere close to the processor to work with it anyway (see e.g. [Alb97]). It is very practical, since multiple consecutive accesses to data from the same block are handled as though they were just one access to that block.

6.2 Consequences

Lemma 6.3. *Any lazy cache replacement policy P will incur a miss for every block b, which has not been in the access sequence before, if the starting cache state does not contain b.*

Proof. If b is not contained in the starting cache state, but a policy can incur a hit for b at a later point, there has to be a point in the access sequence, where it became possible for b to be in the cache. That means, at this access, it is possible that b was loaded to the cache, since it was not in the cache before. So this access has to be an access to b, otherwise P cannot be lazy by definition. □

Lemma 6.4. *Any caching cache replacement policy P will incur a hit for every block b, which has just been accessed before.*

Proof. Since b has been accessed just before, it has to be in the cache by definition of a caching policy. Since it is in the cache, it will incur a hit when accessed again. □

These consequences allow us to construct access sequences, where we know for some accessed blocks, that they will produce hits (or misses), regardless of the considered cache replacement policy.

7 Results

As a note for the reader: We are sometimes talking about *random* cache replacement policies. This only emphasizes the fact, that the results we are presenting are not limited to deterministic policies. It does not signalize a requirement, that randomness has to be involved in the choice of blocks to evict.

Let's start by defining a permutation-based policy, because most of our results are based on those. Examples for permutation-based policies are the already mentioned policies LRU, PLRU and FIFO. So this is a relevant class of policies, since it contains the most used policies in practice.

Definition 7.1. A permutation based policy is one, which tracks an order of the blocks currently in the cache. This order will determine, which of the blocks to evict first on a miss. There are two basic properties any permutation based policy must follow:

1. Upon a miss, the next block in order is evicted. Now every block is one place closer to being evicted. The accessed block is loaded into the cache and stored at the end of the evict list (it will be evicted last).

2. Upon a hit, no block is loaded into the cache. It is allowed to change the order of eviction. The permutation of blocks in the eviction list can only depend on the place of the accessed block in the eviction list (and on a random factor in the case of a random replacement policy).

Remark 7.2. Permutation based policies are lazy and caching.

Remark 7.3. A deterministic permutation based policy on an A-way cache is fully determined by its permutations if places $0, ..., A-1$ are hit. Therefore there are exactly $A!^A$ such policies.

Remark 7.4. The cache state of a permutation based policy, be it deterministic or not, after A misses, where A is its associativity, is deterministic. Moreover, if the accessed blocks $b_1, ..., b_A$, which produced the misses, are accessed in this order, exactly those blocks are in the cache and their eviction order is $b_A, ..., b_1$, with b_1 being evicted next.

7.1 The Main Lemma

Before we procede to proving the theorems, the general procedure we are using when constructing counterexamples will be described:

Lemma 7.5. *Let $(k_n) \subset \mathbb{N}, \lim_{n\to\infty} k_n = \infty$ be a sequence of natural numbers going to infinity, and $p \in]0, 1]$.*

Now let A and B be cache replacement policies, which fulfill our performance requirements.

Let $(s_n) \subset \mathcal{B}^*$ be a sequence of (finite) sequences of blocks with the following properties:

1. *(a) There are at least k_n blocks in s_n, s.t. B can have a hit with probability $p > 0$, but A cannot have a hit. OR*

   (b) There are at least k_n blocks in s_n, s.t. B can have a miss with probability $p > 0$, but A cannot have a miss.

2. *For the blocks not covered by the first point, both algorithms produce the same result (hit or miss) deterministically.*

Then B is non-competitive w.r.t. A.

Remark 7.6. The access sequences s_n must have length going to infinity.

Remark 7.7. Note that $p = 1$ is a valid possibility for the lemma. This reflects (amongst other things) the possibility, that B is a deterministic algorithm.

Proof. We will prove the lemma for the *(a)* variant of the first point. The *(b)* variant is equivalent to it, since we are only interested in the possible states and their probabilities, not if the most probable state has a lot of hits or a lot of misses. Basicly, reading the rest of the proof but replacing each instance of the word "hit" with "miss" and the other way around, yields a proof for the *(b)* variant.

Let's consider the program P_n, whose possible memory access patterns, depending on a secret input, are represented by s_n and some other sequences constructed from it. The construction works as follows: The first sequence is $S_{k_n} = s_n$. The next sequence (S_{k-1}) is always constructed recursively from the last one (S_k) by following this procedure:

1. The last block b of S_k, where the two policies may differ is determined. If there is no such block, we are done with the construction of sequences and S_k is the last one.
2. Remove this block and all blocks following it.
3. For each block removed this way, starting at b and going forward, look at what algorithm A would have produced, when it came to that block.
4. If it was a hit, add to the sequence the block, which was just accessed (note that this cannot be the first block, since we start at an empty cache, so the first block will always incur a miss). If it was a miss, add to the sequence a block, which was never accessed in the sequence before.

Now that means the following: We constructed the other sequences in such a way, that

1. A produces the exact same, deterministic result (sequence of hits and misses) for all input sequences.
2. The number of blocks, where the policies differ, is k_n for s_n, so there are $k_n + 1$ different sequences, from S_{k_n} to S_0.
3. For the other sequences this number is smaller, to be precise: For S_{k_n}, it is k_n. A sequence with an index of exactly 1 less has also 1 less such opportunity by construction. Therefore the index of the sequence corresponds to the number of times B has the opportunity to have another result than A.

Let us call the number of hits of A during the sequences h_A. Then the number of hits of B on the sequence S_0 is h_A as well. On any other sequence though, the number of hits of B is non-deterministic (if $p < 1$). The general formula for

the probability to hit i out of k additional times with constant probability p is the binomial distribution:

$$\mathbb{P}(O(B, S_k) = h_A + i) = \binom{k}{i} p^i (1-p)^{k-i} \tag{8}$$

Now we want to prove non-competitiveness, i.e., that there exists no c, s.t. Equation 6 holds. We have

$$\log \sum_{o \in \mathcal{O}} \max_{k \in \mathcal{K}} \mathbb{P}(O(A, K) = o | K = k) = \log \sum_{o = h_A} \max_{k \in \{0, \dots, k_n\}} 1 = \log 1 = 0. \tag{9}$$

The only observation a time-driven adversary can make, is, that there are h_A hits and $|s_n| - h_A$ misses, so he cannot gain any information from this[3]. Further

$$\begin{aligned} \log \sum_{o \in \mathcal{O}} \max_{k \in \mathcal{K}} \mathbb{P}(O(B, K) = o | K = k) &= \log \sum_{i=0}^{k_n} \max_{k \in \{0, \dots, k_n\}} \mathbb{P}(O(B, S_k) = h_A + i) \\ &= \log \sum_{i=0}^{k_n} \max_{k \in \{0, \dots, k_n\}} \binom{k}{i} p^i (1-p)^{k-i}. \end{aligned}$$

The non-existence of c is now achieved, if we can show, that this measure is unbounded for $n \to \infty$. This is equivalent to showing the divergence of a series, since the logarithm is monotone. Since $k_n \to \infty$, the series to consider[4] is

$$\sum_{i=0}^{\infty} \max_{k \in \mathbb{N}_0} \binom{k}{i} p^i (1-p)^{k-i}.$$

We will prove the divergence using direct comparison, i.e. we will find a series with elements no greater than this, which diverges. The reasoning goes as follows:

First, if we use a specific term for k, it might be the maximum or smaller, so if we just choose $k := \lfloor \frac{i}{p} \rfloor$, every term of the series can only get smaller.

$$\sum_{i=0}^{\infty} \max_{k \in \mathbb{N}_0} \binom{k}{i} p^i (1-p)^{k-i} \geq \sum_{i=0}^{\infty} \binom{\lfloor \frac{i}{p} \rfloor}{i} p^i (1-p)^{\lfloor \frac{i}{p} \rfloor - i} \tag{10}$$

Second, we rewrite the binomial coefficient for $i \geq 1$ using the stirling formula, as bounded in [Rob55]:

[3] Note that even a trace-driven attacker has only a single observation, since the succession of hits and misses is the same in every sequence from S_{k_n} to S_0 when applying A

[4] Actually, the fact that we are able to extend this limit to the maximum and the sum is non-trivial, see Appendix (Chapter 9) for more information

$$
\begin{aligned}
\binom{\lfloor \frac{i}{p} \rfloor}{i} &= \frac{\lfloor \frac{i}{p} \rfloor}{(\lfloor \frac{i}{p} \rfloor - i)! i!} \\
&\geq \frac{\sqrt{2\pi \lfloor \frac{i}{p} \rfloor} \left(\frac{\lfloor \frac{i}{p} \rfloor}{e} \right)^{\lfloor \frac{i}{p} \rfloor} e^{\frac{1}{12 \lfloor \frac{i}{p} \rfloor + 1}}}{\sqrt{2\pi(\lfloor \frac{i}{p} \rfloor - i)} \left(\frac{\lfloor \frac{i}{p} \rfloor - i}{e} \right)^{\lfloor \frac{i}{p} \rfloor - i} e^{\frac{1}{12(\lfloor \frac{i}{p} \rfloor - i)}} \sqrt{2\pi i} \left(\frac{i}{e} \right)^{i} e^{\frac{1}{12i}}} \\
&= \sqrt{\frac{\lfloor \frac{i}{p} \rfloor}{2\pi(\lfloor \frac{i}{p} \rfloor - i) i}} \frac{\left(\frac{\lfloor \frac{i}{p} \rfloor}{e} \right)^{\lfloor \frac{i}{p} \rfloor - i} \left(\frac{\lfloor \frac{i}{p} \rfloor}{e} \right)^{i}}{\left(\frac{\lfloor \frac{i}{p} \rfloor - i}{e} \right)^{\lfloor \frac{i}{p} \rfloor - i} \left(\frac{i}{e} \right)^{i}} \frac{e^{\lfloor \frac{i}{p} \rfloor}}{e^{\lfloor \frac{i}{p} \rfloor}} e^{\frac{1}{12 \lfloor \frac{i}{p} \rfloor + 1} - \frac{1}{12(\lfloor \frac{i}{p} \rfloor - i)} - \frac{1}{12i}} \\
&\geq \sqrt{\frac{1}{2\pi \frac{i}{p} i}} \left(\frac{\lfloor \frac{i}{p} \rfloor}{\lfloor \frac{i}{p} \rfloor - i} \right)^{\lfloor \frac{i}{p} \rfloor - i} \left(\frac{\lfloor \frac{i}{p} \rfloor}{i} \right)^{i} e^{-\frac{1}{6}} \\
&= c(p) \frac{1}{i} \left(\frac{\lfloor \frac{i}{p} \rfloor}{\lfloor \frac{i}{p} \rfloor - i} \right)^{\lfloor \frac{i}{p} \rfloor - i} \left(\frac{\lfloor \frac{i}{p} \rfloor}{i} \right)^{i}
\end{aligned}
$$

In the last line, we defined a constant $c(p) := \sqrt{\frac{p}{2\pi}} e^{-\frac{1}{6}}$. It is constant, because p is independent of i. When we include this estimate in our result from (10), the last two terms almost cancel out. We get the expression:

$$
\begin{aligned}
1 + \sum_{i=1}^{\infty} \binom{\lfloor \frac{i}{p} \rfloor}{i} p^i (1-p)^{\lfloor \frac{i}{p} \rfloor - i} &\geq c(p) \sum_{i=1}^{\infty} \frac{1}{i} \left(\frac{\lfloor \frac{i}{p} \rfloor - p \lfloor \frac{i}{p} \rfloor}{\lfloor \frac{i}{p} \rfloor - i} \right)^{\lfloor \frac{i}{p} \rfloor - i} \left(\frac{p \lfloor \frac{i}{p} \rfloor}{i} \right)^{i} \\
&\geq c(p) \sum_{i=1}^{\infty} \frac{1}{i} \left(\frac{p \lfloor \frac{i}{p} \rfloor}{i} \right)^{i} \qquad (11)
\end{aligned}
$$

The first of these terms is easier to handle, since $p \lfloor \frac{i}{p} \rfloor \leq i$ means, that we have a power of base ≥ 1 and non-negative exponent ($p \leq 1$). We can just bound this from below with 1. The second one is a little more intricate. Let's define:

$$
q := \frac{1}{p}, q_i := \frac{\lfloor \frac{i}{p} \rfloor}{i}
$$

By definition of the floor function, these properties follow:

$$
q_i \leq q \leq q_i + \frac{1}{i} \Rightarrow \lim_{i \to \infty} q_i = q
$$

We can now bound the second term:

$$c(p)\sum_{i=1}^{\infty}\frac{1}{i}\left(\frac{p\lfloor\frac{i}{p}\rfloor}{i}\right)^{i}=c(p)\sum_{i=1}^{\infty}\frac{1}{i}\left(pq_i\right)^{i}\geq c(p)\sum_{i=1}^{\infty}\frac{1}{i}\left(1-\frac{p}{i}\right)^{i}\tag{12}$$

Now it is well-known, that

$$\lim_{i\to\infty}\left(1-\frac{p}{i}\right)^{i}=e^{-p}.$$

So we can find an index $I\in\mathbb{N}$ big enough, such that for $\bar{c}(p):=\sqrt{\frac{p}{2\pi}}e^{-p-1}$:

$$c(p)\sum_{i=1}^{\infty}\frac{1}{i}\left(1-\frac{p}{i}\right)^{i}\geq c(p)\sum_{i=I}^{\infty}\frac{1}{i}e^{-p-\frac{5}{6}}=\bar{c}(p)\sum_{i=I}^{\infty}\frac{1}{i}\tag{13}$$

This is the harmonic series. Its divergence is a basic theorem of analysis (see e.g. [For11]).

In summary, by applying inequalities (6) to (13), we get:

$$\begin{aligned}
\bar{c}(p)\sum_{i=I}^{\infty}\frac{1}{i}&\overset{(13)}{\leq}c(p)\sum_{i=1}^{\infty}\frac{1}{i}\left(1-\frac{p}{i}\right)^{i}\overset{(12)}{\leq}c(p)\sum_{i=1}^{\infty}\frac{1}{i}\left(\frac{p\lfloor\frac{i}{p}\rfloor}{i}\right)^{i}\\
&\overset{(11)}{\leq}\sum_{i=1}^{\infty}\binom{\lfloor\frac{i}{p}\rfloor}{i}p^{i}\left(1-p\right)^{\lfloor\frac{i}{p}\rfloor-i}\overset{(10)}{\leq}\sum_{i=1}^{\infty}\max_{k\in\mathbb{N}_0}\binom{k}{i}p^{i}(1-p)^{k-i}\\
&\leq\sum_{i=0}^{\infty}\max_{k\in\mathbb{N}_0}\binom{k}{i}p^{i}(1-p)^{k-i}\\
&\overset{(8)}{=}\lim_{n\to\infty}\sum_{i=0}^{k_n}\max_{k\in\{0,\dots,k_n\}}\mathbb{P}(O(B,S_k)=h_A+i)\\
&\overset{(6)}{\leq}\lim_{n\to\infty}c\left(\sum_{i=0}^{k_n}\max_{k\in\{0,\dots,k_n\}}\mathbb{P}(O(A,S_k)=h_A+i)\right)^{k}\overset{(9)}{=}c<\infty\qquad\lightning
\end{aligned}$$

This finishes the proof, that there cannot be any $k\in\mathbb{R}$, s.t. (6) holds for algorithms A and B, in other words B is non-competitive w.r.t. A. □

7.2 Resulting Theorems

We are now able to prove our first theorem:

Theorem 7.8. *Any permutation-based policy is non-competitive w.r.t. any other deterministic permutation-based policy.*

Remark 7.9. The "other" is not meant to imply, that the first policy has to be deterministic, it is just necessary, because obviously, any policy is 1-competitive to itself.

Proof. Let's call the random policy R and the deterministic policy D and their respective associativities A_R, A_D. Our goal will be to construct a sequence s_n for each $n \in \mathbb{N}$, such that Lemma 7.5 is applicable. We will first subject both policies to $m := \max(A_R, A_D)$ misses with some arbitrary, different blocks $b_1, ..., b_m$. This produces cache states we will call c_A, c_R. From here on, since the two policies are different, there has to be a sequence of accesses which brings you to two different cache states for the two policies. If $c_A \neq c_R$, we are done for instance. Since R is random and D is not, this might just mean, that after this sequence of accesses, multiple configurations of blocks are possible for R. If we assume this sequence of accesses to be shortest among all sequences of accesses, which produce different cache states, this yields the further properties:

1. Every access to a block produces a hit or a miss deterministically for R.

   This is the case, because if R does contain a block only with a certain probability $0 < p < 1$, its current cache state is already different from the deterministic one of D. So the sequence of accesses could be shortened to the subsequence before this access. ↯

2. Every access to a block produces a hit for both policies, or a miss for both policies.

   If there is some block in the cache of one of the algorithms which is not contained in the cache of the other algorithm, then the cache states are different and the subsequence before this access would be a viable sequence of accesses as well. ↯

Now the cache states are different. This means, that there is a block b, which the two caches do not contain with the same probability. We access b. Afterwards, we will add misses to restitute the starting cache state. Since both policies are replacement policies, this is fairly easy. First access m arbitrary blocks different from the currently possible jobs in the cache and different from $b_1, .., b_m$. Then the caches are filled with these blocks only, so you can access the blocks $b_1, .., b_m$ again to get back to the cache states c_A, c_R. To make yourself clear that this works, see Remark 7.4. You may repeat this procedure an arbitrary amount of times. Repeating this procedure n times produces a sequence in the sense of Lemma 7.5, satisfying $k_n = n$. Specifically, the k_n blocks, where the two policies differ, are the accesses to the block b. If R has the bigger cache, we can hit b for R with a non-zero probability p, but not for D (since the probability to hit has to be strictly smaller than p and is in $\{0, 1\}$), so the (a) variant of the lemma applies. If R has the smaller cache, D is sure to hit b, but R misses with a certain non-zero probability p (analogously, the probability to hit of D has to be bigger than p and in $\{0, 1\}$, so it is 1), so in this case, the (b) variant applies. The other wanted properties follow from the two properties of the sequence we followed from the minimality of the sequence of differentiation earlier. Note that both properties are also fulfilled for the misses we use to create the cache states c_A, c_R multiple times.

□

The second theorem works similarly, but uses a specific, easier sequence. It is about the RANDOM policy.

Definition 7.10. The RANDOM policy is a specific kind of cache replacement policy, whose cache state is solely determined by the *set* of blocks contained in the cache. It will behave as follows:

1. On a hit, the RANDOM policy does not alter the cache at all.
2. On a miss, the RANDOM policy loads the accessed block to the cache. The cache block to evict is chosen at random uniformly. If the cache is not full yet, an empty slot may be chosen to be evicted. In this case, no block is evicted.

Remark 7.11. Note that the RANDOM policy is lazy and caching. Also, the probability for any block in the cache to be evicted on a miss is always $\frac{1}{A}$, where A is the cache associativity.

Theorem 7.12. *The RANDOM policy, which evicts a block uniformly at random on a miss, is non-competitive w.r.t. any permutation-based policy.*

Proof. Let A be the associativity of the permutation-based policy. To begin to construct our sequence s_n for arbitrary $n \in \mathbb{N}$ we will access a block b , which is in neither of the starting caches. Then we will access A blocks $b_1, ..., b_A$, different from each other and b and the blocks in the starting caches. This yields s_0. To construct s_{n+1} from s_n we will always copy the sequence, then continue by accessing b, then again accessing A totally new blocks $b_{(n+1)A+1}, ..., b_{(n+2)A}$. This construction can define the whole sequence $(s_n)_{n\in\mathbb{N}}$ recursively. So s_n looks like this:

$$b, b_1, ..., b_A, b, b_{A+1}, ..., b_{2A}, b, ..., b, b_{nA+1}, ..., b_{(n+1)A} \tag{14}$$

Since both policies are lazy, they can not produce a hit for any of the b_i (See Lemma 6.3). Then again, after n misses for a permutation-based policy, the cache is filled with those blocks (Remark 7.4). So the permutation-based policy can not have a hit for b either. For RANDOM however, there can be a hit. If b is in the cache, on a miss, it is evicted with constant probability $\frac{1}{A_R}$, where A_R is the associativity of the RANDOM policy cache. Since RANDOM is caching, after each access to b, it is in the cache for sure. So with a constant probability of $p = \left(1 - \frac{1}{A_R}\right)^A$, b is still in the cache after A misses. This is regardless of which of the accesses to b (except for the first one) we are talking about. So again, we have constructed a sequence which fulfills variant (a) of Lemma 7.5, with $k_n = n$. Note that $p \neq 0$ as well. □

For the last few theorems, we abused the properties of permutation-based policies to gain information about the competitiveness of policies to them. For explicit policies, there are more properties to abuse. For example, when using LRU, which always evicts the least recently used element, if its associativity is

at least 2, the most recently used block will never get evicted. The property is also true for the Pseudo-LRU variant, which is even more common. We can use this property, when analyzing the following sequence:

$$s_k = (i_a, ..., i_1, n_1, i_1, n_2, i_1, n_3, ..., i_1, n_k, i_1)$$

In this sequence, $i_r, n_s (r \in \{1, .., a\}, s \in \{1, .., k\})$ are pairwise different jobs. Therefore, any lazy policy with associativity $\leq a$ will miss, when any block other than i_1 is accessed (Lemma 6.3). Now there is a multitude of policies which regularly misses some instance of i_1 for growing k, e.g. FIFO and RANDOM, and of course the policies, which will always miss, such as LIFO (Last in, first out) and MRU (Most recently used). Using this sequence, it is possible to show non-competitiveness of all of these policies w.r.t. LRU and PLRU. It comes down to the following

Proposition 7.13. *Let A be a cache replacement policy, which hits i_1 every time (such as LRU), B be a cache replacement policy, which hits i_1 arbitrarily many times deterministically as k goes to infinity and misses i_1 arbitrarily many times deterministically as k goes to infinity as well (such as FIFO), C be a cache replacement policy, which hits i_1 with constant probability (such as RANDOM) and D be a cache replacement policy, which never hits i_1 (such as MRU). If all aforementioned policies satisfy our performance constraints, then*

1. *A, B and D are pairwise non-competitive.*
2. *C is non-competitive w.r.t. A and D.*

Proof. Lemma 7.5 is applicable for the sequence (s_k) in any of the cases:

1. These are in fact 6 cases:
   (a) A is non-competitive w.r.t. B.
   Here all the times, that A hits i_1, while B does not, count for our k_n blocks. We are in the case (a) of the lemma.
   (b) B is non-competitive w.r.t. A.
   The same blocks as in the last case are the k_n blocks. We are in the case (b) of the lemma.
   (c) A is non-competitive w.r.t. D.
   Here all the accesses to i_1 except the first one belong to our k_n blocks. The case (a) of the lemma applies.
   (d) D is non-competitive w.r.t. A.
   Again, the same blocks are considered, but we are in case (b).
   (e) B is non-competitive w.r.t. D.
   Here the blocks, where B hits i_1, are considered our k_n blocks. It is the case (a) of the lemma.

(f) D is non-competitive w.r.t. B.
Another time, the same blocks are considered as above, but we are in case (b).

2. These are only two cases:

   (a) C is non-competitive w.r.t. A.
   We consider all accesses to i_1 except the first to be our k_n blocks for Lemma 7.5. The (a) case yields the wanted result.

   (b) C is non-competitive w.r.t. D.
   One last time, the same blocks are considered as in the last case, but we are in case (b) of the lemma now.

□

7.3 Summary

Our competitiveness results can be summarized as follows:

1. No permutation-based policy is competitive w.r.t. any other deterministic permutation-based policy.
2. The RANDOM policy is non-competitive w.r.t. any permutation-based policy.
3. LIFO and MRU are non-competitive w.r.t. LRU, PLRU and FIFO and vice-versa.
4. RANDOM is non-competitive w.r.t. LIFO and MRU.

The contribution of this paper is a method to construct non-competitiveness witness programs from sequences for arbitrary replacement policies w.r.t. some policy which acts deterministically and independent of the key when executing the program. Such a sequence is easier to find, if the policy to compare to is deterministic. This might be some indicator, that using random policies makes constructing a policy, which is competitive w.r.t. this policy easier. Thus, random policies would be (slightly) less advisable. Another interpretation is probably more realistic: Since deterministic policies are easier to deal with, it is easier to yield any results at all using them. The stronger argument against random policies is, that more randomness normally also means more observations for a potential attacker, which is very bad in the deterministic case. In that case, the number of observations is the exact same as the leaked information. This changes in the random case though. The information obtained by an attacker has to be divided into high-level and low-level information. Obtaining key bits would be high-level, while getting to know which side came up at the coin toss used during the algorithm is defined as low-level information. If all possible observations are possible irrespective of high-level information, then randomization might work. A way to do this is to introduce noise to make observations

harder to interprete. However, this only decreases the amount of information an attacker obtains in expectancy, it does not eliminate it ([OST06]).

The main takeaway from this thesis, proved by the afore-mentioned lemma, is that no permutation-based policy is competitive w.r.t. any other deterministic one in terms of security. This illustrates, how hard dealing with security issues in the most general mathematic way is. With no assumptions on the programs, it is impossible to compare a lot of concurrent policies. This also proves, that they are non-competitive w.r.t. any policy, which in turn is competitive w.r.t. another deterministic permutation-based policy. So even if a not optimal, but only competitive, but reasonably performant policy would be found, which we could use to define plain competitivity for security, our result would yield non-competitivity for a lot of currently used replacement policies.

8 Outlook

It would have been wonderful to include competitiveness results in this thesis. Unfortunately, we did not find any, neither do we know of someone who did. This would definitely be the next step, when building upon the results presented in this thesis.

Is it really impossible to achieve competitiveness in such a general setting? Intuitively, similar policies will achieve a better competitiveness factor. This is backed up by the result on LRU competitiveness w.r.t. PLRU in the performance case, which we highlighted earlier. A method to construct similar policies could be to construct a policy, that acts like LRU in most cases, but sometimes acts like another policy.

When researching in this direction, it seemed like introducing randomness to do this yielded non-competitiveness, irrespective of the probability to act just like LRU. So the question remains: Is it advisable to use randomness in algorithms in order to achieve higher security results? Considering that you have a lot more degrees of freedom in choosing the policy, it seems like it is. Our results seem to point in the opposite direction though.

More non-competitiveness results for randomized policies could be obtained by a similar method as in Lemma 7.5, if someone finds a way to compare

$$\sum_{i=0}^{\infty} \max_{k \in \mathbb{N}_0} \binom{k}{i} p^i (1-p)^{k-i}$$

for different $p \in (0, 1)$. In the lemma, one of the leakages is always 0, but this is not necessary. If for some p, the above series grows significantly faster than for some other p, it would be possible to apply a variant of the lemma to two randomized policies.

Also, there are ways to generalize the results to multiple cache sets, for example looking at only the memory of one cache set at a time and formulate

security results about this part of the memory or looking at the whole cache as one cache set with a cache replacement policy, which replaces blocks only with blocks from the same part of memory. Unfortunately, both of these methods have strong disadvantages. However, as mentioned in the first section of Chapter 9, there are methods to abuse the set-associative structure on its own, so in order to get a cache secure from side-channel attacks, this structure has to be revised. This is something researchers in the area should take a look at more closely.

9 Appendix

9.1 Attacks on Set-Associative Caches

There are multiple ways to attack a cache, such that the cache replacement policy can not interfere with the attack. As mentioned, caches are usually set-associative, this means, that the cache memory is divided into different parts, called the *cache sets*. Instead of loading all blocks from memory into the same cache set, and only letting the cache replacement policy decide, which blocks to evict, it will load certain blocks into certain cache sets, and then those cache sets have their cache replacement policy to decide, which block from this cache set to evict. More precisely, when some block is loaded into the cache, the most insignificant address bits (how many depends on the number of cache sets) determine, which cache set the block is mapped to. This system was set in place because of locality of reference, too. Since most programs use blocks, which share the more significant address bits, these are even less likely to share the most insignificant address bits. For example, blocks just next to each other will never share the most insignificant address bits, unless the number of blocks is bigger than the number of cache sets, in which case two would have to be mapped to the same cache set anyway. So if multiple blocks, which are needed for the same program, are mapped to different cache sets, they cannot evict each other from the cache, which is great, because the program then keeps its blocks in the cache and only evicts blocks from other programs (that are perhaps even not executed anymore). But unfortunately, this way of caching can give an attacker information. With a little bit of knowledge about the program, knowing the least significant address bits of an accessed block is sufficient to derive information about the key. For example, in an AES encryption protocol with lookup tables for faster computation, the location where to look in the lookup table is key dependent. This is what was done in the papers of Osvik et al.([OST06]), as well as Yarom et al. ([YF14]). Since the FLUSH+RELOAD algorithm described in the latter can be seen as a variant of one of the algorithms in the former, namely PRIME+PROBE, we will only be discussing the two algorithms presented there.

(a) PRIME+PROBE:
PRIME+PROBE works on the whole cache at the same time. In order to gain information, blocks are evicted from the cache first,

more precisely all blocks from all cache sets (PRIME). Then, after the victim program's access, accesses to the block, which were used to evict the other blocks from the cache are done and timed (PROBE). On a miss on any of those blocks, it means, that the victim program accessed a block mapped to the same cache set. This makes this attack a disjoint-memory access-driven attack, because the attacker's program is supposed to only be able to access blocks the victim program is not using. Not being able to access these secret blocks, which the program spied on uses, is common, since the attacker's program might (in some cases also should) not have the right to access these parts of the memory. However, with this attack, the attacker does not need the right to do so. This is part of the reason, why potent attacks like FLUSH+RELOAD ([YF14]) can be mounted using this kind of approach.

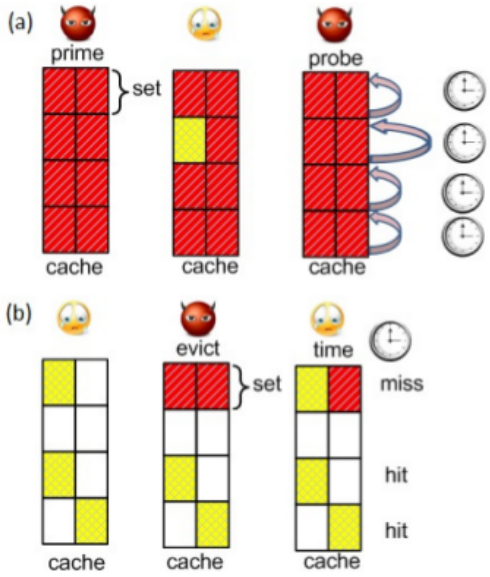

Figure 6: Prime+Probe and Evict+Time explained (by [LL13])

(b) EVICT+TIME:
An attacker following the EVICT+TIME strategy focuses on only one cache set. This is a time-driven attack, so the question, if the attacker is able or not to access the same blocks as the victim program, is not that relevant. First, the execution time of the program is measured. Then, by accessing blocks, which are mapped to this cache set exclusively, all the current blocks in this cache set are evicted (EVICT). Thirdly, the algorithm does another time measurement and compares with the original one (TIME). The additional misses

are due to the accesses to blocks, which are not in the targeted cache set anymore. So the attacker can learn the total number of accesses of the victim to blocks in the targeted cache set. Using a sequence-driven observation of the victim could even tell the attacker, when exactly blocks of the targeted cache set are used. By repeatedly executing the strategy targeting all different cache sets, a lot of information about the cache usage of the victim program can be obtained.

So we have seen multiple ways a cache can be attacked when equipped with whatever cache replacement policy there might be. It is futile to ask oneself, which cache replacement policy to use against these attacks, because it just does not matter. This is why, in this paper, we focus on caches with only a single cache set.

9.2 Interchangeability of Limits

For the proof of Lemma 7.5, there is an additional small lemma needed:

Lemma 9.1. *Let $(a_{nm})_{m\in\mathbb{N}}$ be a monotonely increasing, bounded sequence for each $n \in \mathbb{N}$, such that the sequence $(a_{nm})_{n\in\mathbb{N}}$ is monotonely increasing for each $m \in \mathbb{N}$. Then $(a_{nm})_{m\in\mathbb{N}}$ is convergent for all $n \in \mathbb{N}$ and the sequence of limits $(\lim_{m\to\infty} a_{nm}) =: (a^*_n)_{n\in\mathbb{N}}$ is monotonely increasing. Further*

$$\lim_{n\to\infty}\lim_{m\to\infty} a_{nm} = \lim_{n\to\infty} a^*_n = \lim_{n\to\infty} a_{nn}$$

Remark 9.2 (Well-Definition). Since the sequence of limits as well as the diagonal sequence are monotonely increasing, they are either convergent or unbounded from above. In both cases the limit of the sequence in $\mathbb{R} \cup \{\infty\}$ is unique.

Proof. The fact that (a_{nm}) is convergent for arbitrary n follows from the monotonicity and boundedness of the sequence (for further explanation consult standard analysis literature like [For11]).

For arbitrary $n \geq m \in \mathbb{N}, \epsilon > 0$, we now find $N = \max(N_n, N_m)$, such that

$$|a_{nN} - a^*_n| < \epsilon > |a_{mN} - a^*_m|$$

and by monotonicity of $(a_{nm})_{n\in\mathbb{N}}$, it follows

$$a^*_n > a_{nN} - \epsilon \geq a_{mN} - \epsilon > a^*_m - 2\epsilon \Rightarrow a^*_n \geq a^*_m,$$

where we used in the last step that ϵ can be chosen arbitrarily small.

The first part of the equation is just the definition of a^*_n, so it remains to show:

$$\lim_{n\to\infty} a^*_n = \lim_{n\to\infty} a_{nn} \tag{15}$$

So let $\epsilon > 0$ be arbitrary, then for all n, we can find $N_n \geq n$, such that:

$$
\begin{aligned}
&|a_{nm} - a_n^*| < \epsilon \forall m \geq N_n \\
\Rightarrow\ & a_{N_n}^* = \lim_{m\to\infty} a_{N_n m} \geq a_{N_n N_n} \geq a_{n N_n} > a_n^* - \epsilon
\end{aligned}
$$

Now if $n \to \infty$, then $N_n \to \infty$, since $N_n \geq n$. So by taking the limit $n \to \infty$, we will get

$$
\begin{aligned}
&\lim_{n\to\infty} a_n^* = \lim_{n\to\infty} a_{N_n}^* \geq \lim_{n\to\infty} a_{N_n N_n} \geq \lim_{n\to\infty} a_n^* - \epsilon \\
\overset{\epsilon\ arbitrary}{\Longrightarrow}\ & \lim_{n\to\infty} a_n^* = \lim_{n\to\infty} a_{N_n N_n} = \lim_{n\to\infty} a_{nn}
\end{aligned}
$$

In both steps, we used that the limit of a subsequence is also the limit of the whole sequence for monotone sequences, since their limit point is unique (see Remark 9.2).

Alternate proof of (15) by case distinction:

$$\textbf{Case 1}: \lim_{n\to\infty} a_n^* = \infty$$

In this case, given $R \in \mathbb{R}$, we will always find $N_1, N_2 \in \mathbb{N}$ such that (by monotonicity):

$$
\begin{aligned}
& a_{N_1}^* > R + 1, \left|a_{N_1 n} - a_{N_1}^*\right| < 1 \forall n > N_2, N := \max(N_1, N_2) \\
\Rightarrow\ & a_{NN} \geq a_{N_1 N} > a_{N_1}^* - 1 > R \\
\Rightarrow\ & \lim_{n\to\infty} a_{nn} = \infty
\end{aligned}
$$

$$\textbf{Case 2}: \lim_{n\to\infty} a_n^* =: a < \infty$$

In this case, given $\epsilon > 0$, we will always find $N_1, N_2 \in \mathbb{N}$ such that (by monotonicity):

$$
\begin{aligned}
& \left|a_{N_1}^* - a\right| < \epsilon, \left|a_{N_1 n} - a_{N_1}^*\right| < \epsilon \forall n > N_2, N := \max(N_1, N_2) \\
\Rightarrow\ & a = \lim_{n\to\infty} a_n^* \geq a_N^* = \lim_{n\to\infty} a_{Nn} \geq a_{NN} \geq a_{N_1 N} > a_{N_1}^* - \epsilon > a - 2\epsilon \\
\Rightarrow\ & \lim_{n\to\infty} a_{nn} = a
\end{aligned}
$$

□

Remark 9.3 (Generalization). The lemma also holds, if we exchange increasing with decreasing in the assumptions. It cannot, though, be generalized to monotone sequences altogether, since we need all the sequences $(a_{nm})_{n\in\mathbb{N}}$ to have the same kind of monotonicity to prove monotonicity of the limit sequence and we need $(a_{nm})_{n\in\mathbb{N}}$ and $(a_{nm})_{m\in\mathbb{N}}$ to have the same kind of monotonicity to prove

monotonicity of the diagonal sequence. A counterexample for this lemma, when only monotonicity is assumed, is

$$a_{nm} = \frac{1}{m-n+1} \forall m \geq n, a_{nm} = 1 \forall m < n$$
$$\Rightarrow \lim_{n\to\infty} a_{nn} = \lim_{n\to\infty} 1 = 1 \neq 0 = \lim_{n\to\infty} 0 = \lim_{n\to\infty} a_n^*.$$

Remark 9.4 (Applicability). In the proof of Lemma 7.5,

$$a_{nm} = \sum_{i=0}^{n} \max_{k\in\{0,\dots,m\}} \binom{k}{i} p^i (1-p)^{k-i}.$$

It is monotonely increasing in n and m, since both taking the maximum over more possibilities and taking the sum over more non-negative summands can only lead to an increase in value. Also, since the probabilities $\binom{k}{i}p^i(1-p)^{k-i} \leq 1$, it holds $a_{nm} \leq n$, so the sequence $(a_{nm})_{m\in\mathbb{N}}$ is bounded for all n. It remains:

$$\begin{aligned}
\lim_{n\to\infty} a_{nn} &= \lim_{n\to\infty} a_{k_n k_n} = \lim_{n\to\infty} \sum_{i=0}^{k_n} \max_{k\in\{0,\dots,k_n\}} \binom{k}{i} p^i (1-p)^{k-i} \\
= \lim_{n\to\infty} \lim_{m\to\infty} a_{nm} &= \lim_{n\to\infty} \lim_{m\to\infty} \sum_{i=0}^{n} \max_{k\in\{0,\dots,m\}} \binom{k}{i} p^i (1-p)^{k-i} \\
&= \lim_{n\to\infty} \sum_{i=0}^{n} \max_{k\in\mathbb{N}_0} \binom{k}{i} p^i (1-p)^{k-i} \\
&= \sum_{i=0}^{\infty} \max_{k\in\mathbb{N}_0} \binom{k}{i} p^i (1-p)^{k-i}
\end{aligned}$$

9.3 Acknowledgements

First and foremost, I would like to thank my supervisor Boris Köpf for not only the amount of work that I needed from his side to get me immersed into the subject of cache security with as little as I knew about server and processor architecture before, but also for helping me out during my first few weeks in Madrid not as a boss but rather as a friend. Also, I would like to thank the people at IMDEA in general for being such a welcoming community, especially but not limited to Avinash, Ignacio, Artem, Natalia, Borja, Joakim, Jesús, Anton, Pablo, Pepe, Álvaro, Vincent, Yuri, Luis, Paolo and Arianna. It was a pleasure to work with you guys!

Second, I give thanks to the supervisor of my master thesis, Prof. Skutella and his co-corrector Prof. Seifert who made possible this interdisciplinary project with an ease which should be, but as I feel, is not in general, the standard.

Last, but not least, I thank my parents for the support they give me in order to achieve my studying goals and my girlfriend for being sometimes even more mindful of them than me and for keeping me motivated.

References

[Alb97] Susanne Albers. Competitive online algorithms. *OPTIMA, Mathematical Programming Society Newsletter*, 54:2–8, June 1997.

[Bél66] László A. Bélády. A study of replacement algorithms for a virtual - storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[CS18] Brad Chacos and Michael Simon. Meltdown and Spectre FAQ: How the critical CPU flaws affect PCs and Macs, 2018. `https://www.pcworld.com/article/3245606/security/intel-x86-cpu-kernel-bug-faq-how-it-affects-pc-mac.html`,[Online; Accessed February 13th, 2018].

[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, December 1991.

[For11] Prof. Dr. Otto Forster. *Analysis 1*. Springer, 2011. [German].

[KB07] Boris Köpf and David Basin. An information-theoretic model for adaptive side-channel attacks. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 286–296. ACM, 2007.

[Koc96] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and other Systems. In *Advances in Cryptology — CRYPTO '96*, pages 104–113, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[KS10] Boris Köpf and Geoffrey Smith. Vulnerability bounds and leakage resilience of blinded cryptography under timing attacks. In *Computer Security Foundations Symposium (CSF)*, volume 23, pages 44–56, Edinburgh, United Kingdom, July 2010. IEEE.

[LL13] Fangfei Liu and Ruby B. Lee. Security Testing of a Secure Cache Design. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, pages 3:1–3:8, New York, NY, USA, 2013. ACM.

[MMS90] L.A. McGeoch M.S. Manasse and D.D. Sleator. Competitive algorithms for server problems. *Journal of Algorithms*, 11(2):208–230, 1990.

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of aes. In *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology*, CT-RSA'06, pages 1–20, Berlin, Heidelberg, 2006. Springer-Verlag.

[Pag02] Dan Page. Theoretical use of cache memory as a cryptanalytic side-channel. *IACR Cryptology ePrint Archive*, 2002.

[Por17] Thomas Pornin. Why constant time crypto?, 2017. `https://www.bearssl.org/constanttime.html`,[Online; Accessed March 5th, 2018].

[RG08] Jan Reineke and Daniel Grund. Relative competitive analysis of cache replacement policies. In *Proceedings of the 2008 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, volume 36, pages 431–432. ACM, 2008.

[Rob55] Herbert Robbins. A remark on stirling's formula. *The American Mathematical Monthly*, 62(1):26–29, 1955.

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

[YF14] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *Proceedings of the 23rd USENIX Security Symposium*, pages 718–732. USENIX Association, 2014.