Leakage Resilience against Concurrent Cache Attacks

Gilles Barthe[1], Boris Köpf[1], Laurent Mauborgne[2], and Martín Ochoa[3]

[1] IMDEA Software Institute, Spain
[2] AbsInt GmbH, Germany
[3] TU Munich, Germany

Abstract. In this paper we show how to engineer proofs of security for software implementations of leakage-resilient cryptosystems on execution platforms with concurrency and caches. The proofs we derive are based on binary executables of the cryptosystem and on simple but realistic models of microprocessors.

1 Introduction

The sharing of hardware resources is fundamental for the cost-effective implementation of concurrency in processors, operating systems, and the cloud. Unfortunately, sharing of hardware between conflicting parties introduces side channels that breach the isolation between processes and virtual machines. Typical goals of side-channel attacks are the recovery of cryptographic keys [20] and private information about users [22]; shared resources that have been exploited to this end are processor caches [7], branch prediction units [3], and main memory [14].

Leakage resilient cryptosystems [10, 26] offer formal security guarantees even if the underlying hardware reveals partial information about the internal state of the computation. In today's leakage resilient cryptosystems, the modeling of leakage focuses on physical characteristics such as power consumption or electromagnetic radiation. So far, there has been little focus on applying leakage resilient cryptography to other forms of leakage that arise in the setting of modern computing platforms, and in particular to leakage through cache.

In this paper we show how to engineer proofs of security for software implementations of leakage-resilient cryptosystems on execution platforms with concurrency and caches. Our proofs are based on binary executables of the cryptosystem and on simple but realistic models of microprocessors. We obtain them by tackling the following technical challenges:

– We propose a novel notion of leakage that caters for concurrent access-based adversaries [13, 20]. This notion of leakage characterizes an adversary that can choose an inital cache state and observe the final cache state, for each time slice of a concurrently running computation. We specialize this notion of leakage to pseudorandom generators, and propose a new security definition in which the adversary can freely interleave request queries, that leak information about keys, with test queries, that output a real or random output according to a secret bit b.

Then, we prove leakage-resilience of a PRF-based PRG in this model. Our proof goes beyond the one from [26] in that it allows leakage functions to be adaptively chosen and makes weaker assumptions on locality of leakage. These relaxations are essential for dealing with concurrent access-based adversaries as considered in this paper. We cast our proof in terms of games, for future certification using automated tools [5].

– We propose a novel program analysis technique that allows us to statically derive upper bounds on the range of *all* leakage functions that a concurrent access-based adversary can apply to the state of the cipher. These upper bounds can be used for instantiating the parameters of the cryptographic proof. Our technique is based on an algorithm that efficiently maintains a compact representation of a superset of the set of observations that any concurrent cache adversary can make. We cast this algorithm as an abstract domain [8], which we plug into CacheAudit [9], a framework for the automatic, static analysis of cache side-channels of binary executables.

We perform a case study where we use our analysis techniques for certifying the security of a binary executable of a leakage-resilient pseudorandom generator that is based on a library implementation of the AES block cipher. Using our novel abstract domain, we derive bounds for the side-channel leakage of this implementation to concurrent cache adversaries, which we use to instantiate the cryptographic proof. For example, we can show that the advantage of an adversary for distinguishing the output of the PRG from random is upper-bounded by $\frac{1}{2^{94}}$ for an 8KB cache with 128B line size for AES-256. We stress that on several modern CPUs, AES is either implemented in hardware [1] or can be implemented in software without cache side channels [15]. Here, we use AES as a simple yet realistic example for demonstrating the feasibility of platform-based security proofs.

In summary, our contributions are to show how existing cryptosystems can be connected to a notion of leakage that captures caches and concurrency, and to develop program analysis techniques that enable us to statically deliver leakage bounds based on executable code.

Related Work Leakage resilient cryptography (e.g. [10, 19]) provides models for expressing the security of cryptosystems against adversaries that can obtain partial information about the internal state of the computation. Yu et al. [26] specialize these models to match engineering experience in power analysis attacks. In particular, they account for an adversary who chooses the leakage functions a priori, i.e. before the attack. Moreover, their model requires that each leakage function is applied only to the inputs and outputs of a particular round. Based on these assumptions, they prove the security of a simple pseudorandom generator.

Our leakage model is inspired by that of Yu et al. [26], but it differs in the following two aspects. First, we allow the adversary to adaptively choose leakage functions between rounds, from a fixed set of leakage functions. This accounts for the fact that, in concurrent cache attacks, the adversary can partially influence the leakage during the attack by interacting with the cache and the scheduler.

Second, instead of applying a leakage function to the inputs and outputs of a round, we apply it to *all* previously sampled keys. This accounts for the fact that, in cache attacks, keys might persist in the cache beyond the rounds in which they are used.

On the static analysis side, we base our work on CacheAudit [9], a framework for the automatic, static analysis of cache-based side-channels. CacheAudit makes use of the fact that one can obtain upper bounds for the information leaked through the cache by abstract interpretation and model counting [17, 18].

An alternative, language-based approach by Zhang et al. [27] is to mitigate timing side channels based on systematic addition of delays. Another approach by Stefan et al. [24] uses typing and restrictive scheduling to close cache timing leaks. Our adversary model differs from theirs in that we consider access-based adversaries, i.e. those that can probe the cache. Finally, two recent approaches rely on the operating system making sure that caches are flushed upon context switches [4] or that security-relevant blocks are never evicted from the cache [16]. In contrast, our approach focuses on the security of the client program and makes only weak assumptions on the operating system.

Organization of this paper In Section 2 we formalize a leakage model for concurrent cache adversaries and in Section 3 we present a proof of cryptographic security against this leakage model. In Section 4 we present algorithms to compute bounds on the leakage based on binary code, which we put to work in a case study in Section 5. We conclude in Section 6.

2 Leakage to Concurrent Cache Adversaries

In this section we express the information that is leaked to a cache side-channel adversary in terms of program semantics, where we consider a scenario in which the adversary and the victim are concurrent processes that share the same processor cache. Upon a context switch the adversary partially *observes* the final cache state of the victim's computation.[4] The adversary can further *choose* the initial state of the cache of the victim's subsequent time slice. Early instances of this kind of attack against AES can be found in [7, 20], a more recent and highly effective one is [13].

2.1 Programs, Computations, Caches

A *program* $P = (\Sigma, I, \mathcal{T})$ consists of the following components

- Σ - a set of *states*
- $I \subseteq \Sigma$ - a set of *initial* states
- $\mathcal{T} \subseteq \Sigma \times \Sigma$ - a *transition relation*

[4] In practice, the adversary performs memory accesses and measures the corresponding latencies, thereby learning which memory blocks are loaded in, or have been evicted from, the cache (but not their content).

For reasoning about cache side channels, we consider a program state that consists of logical memories (representing values of memory locations and registers) in $\mathcal{M}$ and a cache state in $\mathcal{C}$ (representing the memory blocks that are currently loaded, but *not* their content), i.e., $\Sigma = \mathcal{M} \times \mathcal{C}$. The *memory update* $upd_{\mathcal{M}}$ is a function $upd_{\mathcal{M}} \colon \mathcal{M} \to \mathcal{M}$ that is determined by the instruction set semantics. The *cache update* is a function $upd_{\mathcal{C}} \colon \mathcal{M} \times \mathcal{C} \to \mathcal{C}$ that is determined by the cache replacement strategy. For a formal description of the LRU replacement strategy, see Appendix A.1. We obtain the global transition relation $\mathcal{T} \subseteq \Sigma \times \Sigma$ as

$$\mathcal{T} = \{((m_1, c_1), (m_2, c_2)) \mid m_2 = upd_{\mathcal{M}}(m_1) \ \wedge \ c_2 = upd_{\mathcal{C}}(c_1, m_1)\}$$

which formally captures the asymmetric relationship between logical memories and caches.

A *computation* of P is a sequence of states and $\sigma_0 \sigma_1 \dots \sigma_n \in \Sigma^*$ such that $\sigma_0 \in I$ and that for all $i \in \{0, \dots, n-1\}$, $(\sigma_i, \sigma_{i+1}) \in \mathcal{T}$. The set of all computations is the *trace collecting semantics* $Col(P)$. We further denote the projection of all computations to logical memories by $Col^{\mathcal{M}}(P)$.

2.2 Leakage to Concurrent Adversaries

We assume that our program runs concurrently with the adversary, where we make the worst-case assumption that the adversary can probe and set the cache state at each context switch. For formalizing this adversary we assume a given set of *context switches* $A \subseteq \mathbb{N}$. A *concurrent computation for* A is a sequence of states $(m_0, c_0), \dots, (m_n, c_n) \in \Sigma^*$ such that

1. for all $i \in \{0, \dots, n-1\} \colon m_{i+1} = upd_{\mathcal{M}}(m_i)$, i.e. the logical memory is always updated according to the program semantics;
2. for all $i \in \{0, \dots, n-1\} \setminus A \colon c_{i+1} = upd_{\mathcal{C}}(c_i, m_i)$, i.e. without a context switch the cache is updated according to the program semantics;
3. for all $i \in A \cup \{0\} \colon c_{i+1} = upd_{\mathcal{C}}(c_i^*, m_i)$, i.e. at each context switch and initially, the cache can be set to an arbitrary state c_i^* by the adversary.

That is, the adversary's *choices* can be expressed as a tuple $a \in \mathcal{C}^{A \cup \{0\}}$ of cache states. They define a mapping

$$adv_{A,a} \colon Col^{\mathcal{M}}(P) \to Col_A(P)$$

where $Col_A(P)$ denotes the set of all concurrent computations for A.

Likewise, we can express the *observations* an adversary can make at context switches in A as a function $\pi_A \colon Col_A(P) \to \mathcal{C}^A$ that projects concurrent computations to sub-sequences of cache states with indices in A. The composition of both functions defines a *leakage function*

$$\Lambda^{A,a} = \pi_A \circ adv_{A,a}$$

that maps internal states of the computation to cache observations at A. Notice that cache states in our model only track *which* memory blocks are loaded,

but not their content. Observing a cache state models leakage about accesses to memory space that is shared between victim and adversary, as in [13]. For modeling disjoint memory spaces, we consider observations that only reveal *how many* memory blocks are loaded in each cache set [17].

Leakage about a Key For expressing the leakage about keys in round-based constructions, we assume that the key of round j can only affect the cache between positions $\alpha(j)$ and $\omega(j)$ of each computation. As a consequence, information about the key is observable at context switches between those positions. Moreover, information about the key may also persist in the cache state beyond $\omega(j)$ and be observable at the subsequent context switch. We account for this by defining $A_{\alpha(j),\omega(j)} = A \cap \{\alpha(j), \ldots, \omega(j)\} \cup \min\{i \in A \mid i \geq \omega(j)\}$. For given (A, a), the leakage about the key of round j can then be over-approximated by the following function:

$$\Lambda_j^{A,a} = \pi_{A_{\alpha(j),\omega(j)}} \circ adv_{A,a}$$

Schedulers Without any restrictions on when context switches can occur, we cannot hope to obtain meaningful security guarantees.[5] To model such restrictions, we introduce the notion of a *scheduler* $S \subset \mathcal{P}(\mathbb{N})$ that describes all permitted sets of context switches A. For a given scheduler, we can completely characterize the set of functions $\mathcal{L}_j$ the adversary can apply to a round key by

$$\mathcal{L}_j = \bigcup_{\substack{A \in S \\ a}} \Lambda_j^{A,a}$$

This class of leakage function will provide the interface between the cryptographic proofs and the guarantees derived by the static analysis.

3 Leakage Resilient PRG

Stateful pseudo-random number generators (PRGs) that depend on a secret key can be used as the basis for stream ciphers. Such constructions have been proposed as a means to provide leakage resilient cryptographic primitives [10, 21, 25, 26]. In this section, we prove the security of a stateful pseudo-random number generator based on a pseudo-random function (PRF), assuming partial leakages $\mathcal{L}_j$ on round keys as discussed in the previous section. Our proof is given in terms of bounds on the advantage of distinguishing the PRG from a truly random generator, depending on the computational power of the adversary and the maximal leakage per key.

[5] This is demonstrated by an attack that successfully recovers the secret key in such a setting [13]

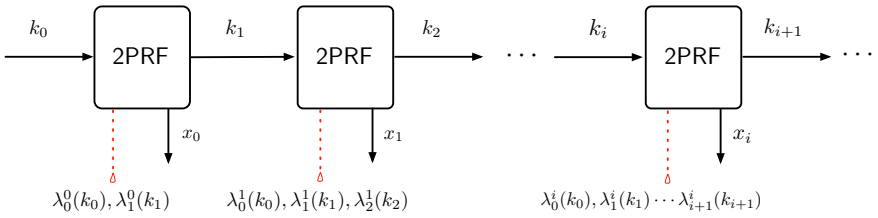

Fig. 1. Stateful PRG. Leakage is depicted by dotted arrows.

3.1 A Leakage Resilient PRG

The construction is depicted in Figure 1, and is based on a pseudorandom function $\mathsf{2PRF}\colon \{0,1\}^n \to \{0,1\}^{n+m}$ that takes as input a round key k_i of n bits and returns as output a pseudorandom string of $n+m$ bits [6]. The first n bits of this string are used as a key k_{i+1} for the next round, and the last m bits are output. The sequence $x_0 x_1 \ldots$ is the output of the pseudorandom generator.

Our security proof is inspired from [26], and follows the spirit of so-called *practical* leakage-resilient cryptography, where bounds are obtained assuming leakage functions that match engineering practice. In particular, we make the assumption that leakage does not reveal information about future computations; for concurrent access-based cache attacks, this assumption is perfectly natural, since caches only hold information about *past* computations made by the victim. Our proof is based on the random oracle model; extending the proof to the standard model as done e.g. in [26] is left for further work.

The security of the pseudorandom generator is expressed in terms of a cryptographic game where, in each round i, the adversary can do one of the following:

- choose a list of leakage functions $\lambda_0^i, \ldots, \lambda_{i+1}^i$ and observe the values of $\lambda_0^i(k_0), \ldots, \lambda_{i+1}^i(k_{i+1})$ together with the legitimate output x_i of the $\mathsf{2PRF}$ at round i. This is called a $\mathsf{request}$ query.
- test the round, and get the legitimate output x_i or a random output, according to a secret bit b sampled uniformly at the onset of the game. This is called a test query.

Moreover, the adversary has access to an oracle $\mathsf{2PRF}_{\mathrm{adv}}$ which he can query for the output of the $\mathsf{2PRF}$, for a chosen key. After p rounds of this game, the adversary is asked to guess the bit b. The adversary wins if his guess $\bar{b}$ is correct, i.e. $b = \bar{b}$. In summary, this game captures the notion that outputs of the $\mathsf{2PRF}$ should be indistinguishable from random.

The game is formally defined in Figure 2. We use K_{adv} to store the keys queried by the adversary to the $\mathsf{2PRF}_{\mathrm{adv}}$ oracle and K_{reqtest} to store the round

[6] Such a function can be constructed for instance by choosing an $IV \in \{0,1\}^{n-1}$ and defining $\mathsf{2PRF}(k) = (\mathsf{BC}_k(0||IV), \mathsf{BC}_k(1||IV))$ given an n-bit block-cipher BC as we will do in Sect. 5.

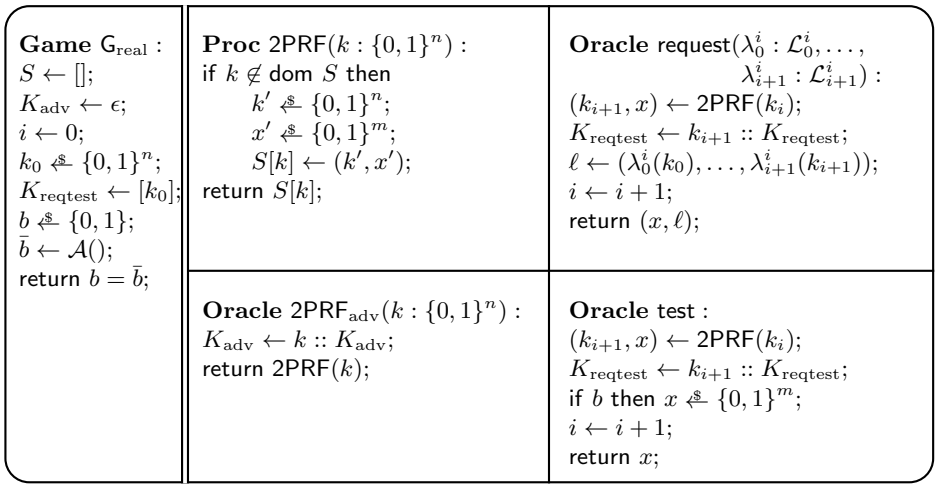

Fig. 2. Initial game G_{real}

keys of the stateful PRG. We store the values sampled by 2PRF in the array S. Note that, at each round i, the leakage functions are chosen from sets $\mathcal{L}_0^i, \ldots, \mathcal{L}_{i+1}^i$. Then $\mathcal{L}_j = \prod_i \mathcal{L}_j^i$ is the set of functions that the adversary can apply to the key of round j.

We now present the main theorem of this section, which quantifies the advantage of any adversary from distinguishing the PRG from a truly random number generator, given that he makes at most q queries to the $\mathsf{2PRF}_{\text{adv}}$ oracle, sees at most p outputs of the PRG and that the total leakage per key is bounded by a constant d.

Theorem 1. *Let $\mathcal{A}$ be an adversary that makes at most p queries to* request *or* test*, and at most q queries to $\mathsf{2PRF}_{\text{adv}}$. If for all $\Lambda_i \in \mathcal{L}_i$, $0 \leq i < p$, it holds $|\mathsf{ran}(\Lambda_i)| \leq d$ then:*

$$\Pr[\mathsf{G}_{\text{real}} : b = \bar{b}] \leq \frac{1}{2} + \frac{p\ (p-1) + q\ p\ (d+1)}{2^n}$$

Proof. The idea of the proof is to bound the adversary's advantage by the probability of the following events: 1) there is a cycle in the PRG, due to a repetition of a round key, 2) the adversary guesses a round key that was already used in a previous round and 3) the adversary guesses a round key before it is used. These are precisely the cases in which an adversary could distinguish the PRG from a truly random generator, as we show using a game reduction and Shoup's Lemma [23]: Starting from the original game as depicted in Figure 2, we defined a transformed version G_1, where we modify the oracles 2PRF and $\mathsf{2PRF}_{\text{adv}}$ so that only adversary queries are stored in the map S, whereas queries originating from request and test are always answered with fresh random values. This only

makes a difference if there is a collision between secret keys, i.e. $k_i = k_{i'}$ for distinct i and i' (we call this event bad_{RR}), or if the adversary calls the $\mathsf{2PRF}_{\mathrm{adv}}$ oracle with a secret key, i.e. $k_i \in K_{\mathrm{adv}}$ for some i; we distinguish the case where the adversary queries $\mathsf{2PRF}_{\mathrm{adv}}$ with k_i before the i^{th} round (we call this event bad_{AR}) from the case where the adversary query occurs after the i^{th} round (and call this event bad_{RA}). We introduce bad flags to capture these events and modify the code of the oracles as follows:

Proc $\mathsf{2PRF}(k : \{0,1\}^n)$:	**Oracle** $\mathsf{2PRF}_{\mathrm{adv}}(k : \{0,1\}^n)$:	**Oracle** test :
if $k \in K_{\mathrm{reqtest}}$ then	if $k \in K_{\mathrm{reqtest}}$ then	$k_{i+1} \xleftarrow{\$} \{0,1\}^n$;
$\quad \mathsf{bad}_{RR} \leftarrow \mathsf{true}$;	$\quad \mathsf{bad}_{RA} \leftarrow \mathsf{true}$;	$x \xleftarrow{\$} \{0,1\}^m$;
if $k \in K_{\mathrm{adv}}$ then	$K_{\mathrm{adv}} \leftarrow k :: K_{\mathrm{adv}}$;	$K_{\mathrm{reqtest}} \leftarrow k_{i+1} :: K_{\mathrm{reqtest}}$;
$\quad \mathsf{bad}_{AR} \leftarrow \mathsf{true}$;	if $k \notin \mathsf{dom}\ S$ then	$i \leftarrow i + 1$;
$k' \xleftarrow{\$} \{0,1\}^n$;	$\quad k' \xleftarrow{\$} \{0,1\}^n$;	return x;
$x' \xleftarrow{\$} \{0,1\}^m$;	$\quad x' \xleftarrow{\$} \{0,1\}^m$;	
return (k', x');	$\quad S[k] \leftarrow (k', x')$;	
	return $S[k]$;	

The two games are equivalent up to bad. It follows from Shoup's Lemma [23] that

$$\left|\Pr[\mathsf{G}_{\mathsf{real}} : b = \bar{b}] - \Pr[\mathsf{G}_1 : b = \bar{b}]\right| \leq \Pr[\mathsf{G}_1 : \mathsf{bad}_{RR}] + \Pr[\mathsf{G}_1 : \mathsf{bad}_{AR}] + \Pr[\mathsf{G}_1 : \mathsf{bad}_{RA}]$$

Moreover, the key k_i is always a freshly sampled value and $|K_{\mathrm{reqtest}}| \leq p$, therefore the event $k_i \in K_{\mathrm{reqtest}}$ (bad_{RR}) is a standard birthday event and has a probability of at most $\frac{p\ (p-1)}{2^n}$. Also, the probability of the event $k_i \in K_{\mathrm{adv}}$ (bad_{AR}) for a given freshly uniformly sampled k_i key is upper bounded by $\frac{q}{2^n}$. There are at most p rounds, i.e. $0 \leq i < p$, thus the probability of a collision between k_i and a key in K_{adv} is upper bounded by $\frac{q\ p}{2^n}$. Finally, the value x output by the test oracle is a fresh uniformly sampled value for each round, and hence the probability of the adversary $\mathcal{A}$ guessing the bit b correctly in G_1 is $\frac{1}{2}$. Summarizing, we have

$$\left|\Pr[\mathsf{G}_{\mathsf{real}} : b = \bar{b}] - \frac{1}{2}\right| \leq \frac{p\ (p-1)}{2^n} + \frac{q\ p}{2^n} + \Pr[\mathsf{G}_1 : \mathsf{bad}_{RA}]$$

Next, we introduce a game G_2 in which a fresh key k is sampled uniformly at the onset of the game, and an adversary $\mathcal{A}'$ can observe at each round i the value of $\lambda^i(k)$, for a leakage function λ^i drawn from a set $\mathcal{L}^i$. The adversary wins if he guesses correctly the key. The game is formalized as follows:

Game G_2 :	**Oracle** $\mathsf{leak}(\lambda^i : \mathcal{L}^i)$:
$i \leftarrow 0$;	$\ell \leftarrow \lambda^i(k)$;
$k \xleftarrow{\$} \{0,1\}^n$;	$i \leftarrow i + 1$;
$\bar{k} \leftarrow \mathcal{A}'()$;	return ℓ;
return $k = \bar{k}$;	

One can prove that for every adversary $\mathcal{A}$ against G_1 making at most p queries to the $\mathsf{request}$ oracle and q queries to the $\mathsf{2PRF}_{\mathrm{adv}}$ oracle, there exists an adversary $\mathcal{A}'$ against G_2 such that $\Pr[\mathsf{G}_1 : \mathsf{bad}_{RA}] \leq q\ p\ \Pr[\mathsf{G}_2 : k = \overline{k}]$.

Finally, $\Pr[\mathsf{G}_2 : k = \overline{k}]$ is upper-bounded by $\frac{d}{2^n}$, given the assumption that the total range of his observations is upper bounded by d, i.e. $\prod_{i=0}^{p} |\mathsf{ran}(\lambda^i)| \leq d$. This last step follows from Theorem 1 of [9], which we reproduce in Appendix A.3.

4 Computing Bounds on the Leakage

For computing the range of the leakage functions that a concurrent cache-based adversary can apply to the internal state of a concrete program, one needs to consider *all* possible computations, which is infeasible in most cases. Abstract interpretation [8] overcomes this fundamental problem by resorting to an approximation of the state space and the transition relation. In this section, we present corresponding approximations for concurrent cache-based adversaries. We proceed by reducing the problem in two steps to the problem of computing numbers of reachable cache states, for which static analysis techniques are in place [9, 17]. Throughout the section we rely on the notation introduced in Section 2.

4.1 Reduction to empty initial cache states

In the first reduction step, we show how to soundly abstract from the adversary's choices of cache states. The result is a generalization of a result from [9] to concurrent computations.

Let $a_\emptyset \in \mathcal{C}^{A \cup \{0\}}$ be the mapping that takes each $i \in A \cup \{0\}$ to the empty cache state.

Lemma 1. *For the LRU replacement strategy and all A and $a \in \mathcal{C}^{A \cup \{0\}}$,*

$$\left|\mathsf{ran}(\Lambda^{A,a})\right| \leq \left|\mathsf{ran}(\Lambda^{A,a_\emptyset})\right|$$

Proof. With LRU replacement, each cache set (seen as a list of memory blocks) of the final cache state of the time slice of a computation with initial cache state $a_\emptyset(i)$ is a prefix of the corresponding cache set of the same computation, performed with initial cache state $a(i)$. The remaining lines of each set are determined by $a(i)$. This correspondence defines, for each a, a surjective mapping from $\mathsf{ran}(\Lambda^{A,a_\emptyset})$ to $\mathsf{ran}(\Lambda^{A,a})$, from which the assertion follows.

4.2 Abstract Interpretation

Reachability problems on programs can be cast as finding fixpoints of the transition relation, because reaching a fixpoint means that no new states can be discovered. Abstract interpretation [8] computes such fixpoints based on approximations of the statespace and the transition relation. The relationship between

the abstract and the concrete statespace is given by a *concretization function* γ that maps abstract states to sets of concrete states. A static analysis is *(globally) sound* if the concretization of an abstract fixpoint contains a concrete fixpoint.

In our case, the goal is to define such an abstract domain $\mathcal{T}^\sharp$ whose fixpoints $t^* \in \mathcal{T}^\sharp$ satisfy

$$Col_S^{\mathcal{C}}(P) \subseteq \gamma_{\mathcal{T}}(t^*) \tag{1}$$

CacheAudit [9] is an abstract interpretation framework that enables computing such abstract fixpoints t^* based on binary executables and concrete cache models. For framing a sound cache analysis within CacheAudit, an abstract domain needs to satisfy the following local soundness condition, where $\mathcal{B}$ denotes the set of all memory blocks:

$$\forall t^\sharp \in \mathcal{T}^\sharp, M \subseteq \mathcal{B} : \ upd_{\mathcal{T}}(\gamma_{\mathcal{T}}(t^\sharp), M) \subseteq \gamma_{\mathcal{T}}(upd_{\mathcal{T}^\sharp}(t^\sharp, M)) \ , \tag{2}$$

This statement captures that the abstract cache update function $upd_{\mathcal{T}^\sharp}$ computes a superset of the concrete cache update function $upd_{\mathcal{T}}$. Computing the set of reachable observations w.r.t. $upd_{\mathcal{T}^\sharp}$ is hence necessarily a superset of $upd_{\mathcal{T}}$. The global soundness then follows from the fact that CacheAudit updates the abstract cache with a superset of the set of possible memory blocks that are accessed at each program point.

Theorem 2. *Local soundness implies global soundness, i.e.* (2) $\Rightarrow$ (1)

This theorem from [9] is a specialization of a result of [8] to the way in which abstract domains are combined in CacheAudit.

We present our new abstract domain in two steps: in the first step we abstract sets of cache traces by traces of sets of cache states, while keeping enough information about the history of computation to obtain reasonably precise bounds. In the second step we further abstract to obtain finite representations. Since in abstract interpretation, abstract domains compose, it is enough to prove soundness of each step to have the soundness of the whole abstraction process.

4.3 An Abstract Domain for Concurrent Computations

One of the main reason for the intractability of computing leakage functions is the need to keep track of sets of traces. The first step we propose is to abstract such sets into a single (possibly infinite) trace of abstract states abstracting sets of caches and possible interruption choices of the adversary. For that purpose, we assume a given abstract domain for cache *states* such as the one from [11], whose elements we denote by $c^\sharp$. The concretization of an abstract cache is a set of caches, i.e.

$$\gamma_{\mathcal{C}}(c^\sharp) \subseteq \mathcal{C} \ .$$

In addition, we assume that this abstract domain is equipped with a join $\sqcup$ that soundly over-approximates unions of sets of caches.

We define the abstract domain $\mathcal{T}^\sharp$ that groups together cache states with the same concurrent access history as follows: Each element $t^\sharp \in \mathcal{T}^\sharp$ consists

of a partial map of pairs of nonnegative integers to abstract cache states. We denote this map by $t^\sharp = (c^\sharp_{i,j})$. Here $(c^\sharp_{i,j})$ represents the set of possible cache states at position j when the last context switch happened i steps ago. During the analysis we will maintain $(c^\sharp_{i,j})$ for $1 \leq i \leq j \leq n$, where n is the current program point. That is, our abstract state is of the form

$$\begin{matrix} c^\sharp_{1,1} & c^\sharp_{1,2} & \cdots & c^\sharp_{1,n} \\ & c^\sharp_{2,2} & \cdots & c^\sharp_{2,n} \\ & & \ddots & \vdots \\ & & & c^\sharp_{n,n} \end{matrix}$$

For convenience of notation we further define $c^\sharp_{0,j} = c^\sharp_\emptyset$ (with $\gamma_\mathcal{C}(c^\sharp_\emptyset) = \{c_\emptyset\}$) for all $j \geq 0$.

Concretization Function We first define the concretization of an abstract state $(c^\sharp_{i,j})$, with $0 \leq i \leq j \leq n$, w.r.t. given (ordered) set of context switches $A = \{i_1, \ldots, i_k\}$ with $i_k \leq n$ by:

$$\begin{matrix} \gamma_A(c^\sharp_{i,j}) = & \gamma_\mathcal{C}(c^\sharp_{0,0}) & \cdots & \gamma_\mathcal{C}(c^\sharp_{i_1,i_1}) \\ & \cdot\, \gamma_\mathcal{C}(c^\sharp_{1,i_1+1}) & \cdots & \gamma_\mathcal{C}(c^\sharp_{i_2-i_1,i_2}) \\ & \vdots & & \vdots \\ & \cdot\, \gamma_\mathcal{C}(c^\sharp_{1,i_k+1}) & \cdots & \gamma_\mathcal{C}(c^\sharp_{n-i_k,n}) \end{matrix}$$

The result is a set of traces built from a trace of sets using the Cartesian product $\cdot$, where the end of each line corresponds to the states that can be observed at a context switch.

For a scheduler S, we then define the concretization of an abstract state as the union of the concretizations w.r.t. all sets of context switches in S, i.e.

$$\gamma_\mathcal{T}(c^\sharp_{i,j}) = \bigcup_{A \in S_n} \gamma_A(c^\sharp_{i,j})$$

where $S_n = \{A \cap \{1, \ldots n\} \mid A \in S\}$ denotes the sets of context switches that are truncated at position n.

Abstract Transition Function We define an abstract transition function $upd_{\mathcal{T}^\sharp}(c^\sharp_{0 \leq i \leq j \leq n}, M) = (c^\sharp_{0 \leq i \leq j \leq n+1})$ that maps a set of memory blocks M and a state representing traces of length n to a state representing traces of length $n+1$, where

$$c^\sharp_{i+1,n+1} = upd_{\mathcal{C}^\sharp}(c^\sharp_{i,n}, M) \quad \text{for } 0 \leq i \leq n$$

The case $i > 1$ describes the scenario in which *no* context switch happens at position n. The case $i = 0$ describes the scenario in which a context switch happens, where we only need to consider the empty initial state due to Lemma 1 (recall that $c^\sharp_{0,n} = c^\sharp_\emptyset$). The entries $c^\sharp_{i,j}$ with $j \leq n$ remain unchanged.

The following lemma states the local soundness of our abstract domain. Together with Theorem 2, it ensures the global soundness of our analysis.

Lemma 2. $upd_{\mathcal{T}}(\gamma_{\mathcal{T}}(t^\sharp), M) \subseteq \gamma_{\mathcal{T}}(upd_{\mathcal{T}^\sharp}(t^\sharp, M))$

The proof of Lemma 2 is given in Appendix A.2; it proceeds by a simple unfolding of definitions and a reduction to the soundness of $upd_{\mathcal{C}^\sharp}$.

4.4 Compact Representations of Infinite Computations

With the abstraction described above we can represent cache observations up to a moment n in time. We now propose a further abstraction that enables us to finitely represent and compute cache observations for *all* points in time, using fixpoint techniques of abstract interpretation.

Our current abstract domain for concurrent computations grows in two directions, both of which have to be bounded in a meaningful way: (1) the number of instructions since the beginning of the computation, and (2) the number of instructions since the last context switch happenend.

For bounding (1) we assume that the computation of the individual rounds of the PRG is performed in one main loop whose body takes exactly ℓ steps to execute. We leverage this knowledge to fold the abstract states corresponding to the same number of instructions inside the loop body. Technically, we will abstract each program point $n \in \mathbb{N}$ with the unique $j \in \{0, \dots \ell - 1\}$ such that $j \equiv n \mod \ell$, and we write $j = [n]$. In this way we only need to maintain elements $c^\sharp_{i,[n]}$ instead of all $c^\sharp_{i,n}$.

For bounding (2), we impose a threshold s on the length of the history we track about the last context switch. To achieve soundness, we modify the update function such that the last element $c^\sharp_{s,j}$ aggregates all possible cache states $c^\sharp_{i,j}$ with $i \geq s$.

Our new abstract state, which we denote by $(d^\sharp_{i,j})$, for $0 \leq i \leq s$ and $0 \leq j < \ell$ is updated using the following transition function

$$
\begin{aligned}
d^\sharp_{i+1,[n+1]} &= upd_{\mathcal{C}^\sharp}\left(d^\sharp_{i,[n]}, M\right) && \text{if } 0 \leq i < s-1 \\
d^\sharp_{s,[n+1]} &= upd_{\mathcal{C}^\sharp}\left(d^\sharp_{s,[n]}, M\right) \sqcup d^\sharp_{s-1,[n+1]}
\end{aligned}
$$

Since the second subscript is an equivalence class, the update functions define a set of fixpoint equations. The transfer function for this set of equations is obviously monotonic, so we can iterate from a matrix entirely filled with empty caches to compute this fixpoint. In addition, we can use program points to store columns of this matrix and use the fixpoint iteration techniques already developed for CacheAudit.

Even though $c^\sharp_{i,j}$ with $i > s$ of the abstract state defined in Section 4.3 are not explicitly represented in the above state, we define their concretization by

$$\gamma_{\mathcal{T}}(c^\sharp_{i,n}) = \gamma_{\mathcal{T}}(d^\sharp_{\min\{i,s\},[n]}) \tag{3}$$

The definition of $(d^\sharp_{i,j})$ and the corresponding update function ensures that the thus defined $\gamma_{\mathcal{T}}(c^\sharp_{i,n})$ is always a superset of the concretization of the explicit

representation. We obtain the following corollary for the compactly represented abstract state.

Corollary 1. $upd_{\mathcal{T}}(\gamma_{\mathcal{T}}(t^\sharp), M) \subseteq \gamma_{\mathcal{T}}(upd_{\mathcal{T}^\sharp}(t^\sharp, M))$

Proof. Follows from the proof of the previous lemma and by monotonicity of the new update function.

4.5 Computing Bounds on the Leakage

We now present an algorithm for upper-bounding counting the range of the leakage function based on a fixpoint $(d^\sharp_{i,j})$ with $i \leq s$ and $j \leq \ell - 1$ of the abstract domain described above. For convenience of notation we describe the algorithm in terms of $(c^\sharp_{i,j})$ and explicit indices, which can be immediately translated using Equation (3).

Recall from Section 2 that $A_{\alpha,\omega} = A \cap \{\alpha, \ldots, \omega\} \cup \min\{i \in A \mid i \geq \omega\}$ is the set of context switches at which the adversary can make observations about a secret that is present in logical memory from position α to position ω. The leakage about such secrets can hence be described by the function $\Lambda^{A,a}_{\alpha,\omega} = \pi_{A_{\alpha,\omega}} \circ adv_{A,a}$. We obtain an upper bound on the size of the range of $\Lambda^{A,a}_{\alpha,\omega}$ as follows

$$\begin{aligned}
\left|\Lambda^{A,a}_{\alpha,\omega}(Col^{\mathcal{M}}(P))\right| &\overset{(*)}{\leq} \left|\Lambda^{A,c_\emptyset}_{\alpha,\omega}(Col^{\mathcal{M}}(P))\right| \\
&= \left|\pi_{A_{\alpha,\omega}}(adv_{A,c_\emptyset}(Col^{\mathcal{M}}(P))\right| \\
&\overset{(**)}{\leq} \left|\pi_{A_{\alpha,\omega}}(\gamma_A(c^\sharp_{i,j}))\right| \\
&= \left|\gamma_{\mathcal{C}}(c^\sharp_{i_1-\alpha,i_1})\right| \left|\gamma_{\mathcal{C}}(c^\sharp_{i_2-i_1,i_2})\right| \ldots \left|\gamma_{\mathcal{C}}(c^\sharp_{i_{k+1}-i_k,i_{k+1}})\right|
\end{aligned}$$

where $(*)$ follows from Lemma 1, $(**)$ follows from Theorem 2, and $A_{\alpha,\omega} = \{i_1, \ldots i_{k+1}\}$. We can instantiate this upper bound by applying existing procedures for counting concretizations of abstract cache states [17] to the elements of $(c^\sharp_{i,j})$.

For upper-bounding the leakage for a given scheduler S, we need to maximize the above expression over all $A \in S$. We show how this can be done for a very general class of schedulers, whose only requirement is a lower bound f on the number of instructions processed by the victim between two interruptions by the adversary.

$$S_f = \{A \subseteq \mathbb{N} \mid i = j \vee |i - j| \geq f\} \tag{4}$$

For bounding the leakage w.r.t. to S_f, we first give a recursive formula that expresses the maximal number of observation that an adversary can make at and between context switches at positions x and y

$$R_{x,y} = \max_{x \leq j \leq y-f} R_{x,j} \left|\gamma_{\mathcal{C}}\left(c^\sharp_{y-j,y}\right)\right| \quad \text{if } y \geq x + f \tag{5}$$

where we adopt the convention that $R_{x,y} = 1$ whenever $y < x + f$.

Observe that it is *not* sufficient to use $R_{\alpha,\omega}$ for upper-bounding the leakage of a secret that is present in logical memory between positions α and ω because (1) α does not necessarily coincide with a context switch, and (2) the final context switch may happen an indefinite number of steps after ω. To account for this fact we define

$$L_{\alpha,\omega} = \max \left\{ \max_{\substack{r \leq \alpha \\ \omega \leq t}} \left| \gamma_{\mathcal{C}}(c^{\sharp}_{t-r,t}) \right|, \max_{\substack{r \leq \alpha < x \\ y < \omega \leq t}} \left| \gamma_{\mathcal{C}}(c^{\sharp}_{x-r,x}) \right| R_{x,y} \left| \gamma_{\mathcal{C}}(c^{\sharp}_{t-y,t}) \right| \right\}$$

The left term in the definition of $L_{\alpha,\omega}$ captures the case where no context switch happens between α and ω. The second term captures the case where at least one context switch happens; in this case x is the position of the first, and y is the position of the last context switch between α and ω. For readability we omit further constraints on the minimal distance f between each two context switches.

The following lemma states that $L_{\alpha,\omega}$ describes an upper bound on the information that is leaked about the logical memory between positions α and ω.

Lemma 3.

$$\max_{A \subseteq S} \left| \mathsf{ran}(\Lambda^{A,a}_{\alpha,\omega}) \right| \;\leq\; L_{\alpha,\omega}$$

The correctness of $R_{x,y}$ follows by a simple induction on y using the bound on $\Lambda^{A,a}_{\alpha,\omega}$ described above. The correctness of $L_{\alpha,\omega}$ follows by construction. Equation (5) immediately suggests an implementation of the algorithm using dynamic programming.

4.6 Leakage Per Key

For deriving bounds on the leakage *per key* for the PRG described in Section 3, we also assume that k_i is part of the internal state of rounds i and $i+1$. If each round can be computed in ℓ commands, the leakage about k_i is hence upper-bounded by $L_{i\ell,(i+2)\ell-1}$ As we identify all i modulo ℓ in (5), we immediately obtain that $L_{0,2\ell-1}$ is an upper bound for the leakage about each round key.

Corollary 2. *For all $j \in \mathbb{N}$ we have*

$$\forall \Lambda \in \mathcal{L}_j \colon \; |\mathsf{ran}(\Lambda)| \;\leq\; L_{0,2\ell-1}$$

5 Case Study

In this section we report on a case study where we use our techniques to derive formal security guarantees against concurrent cache-based adversaries for binary executable of the leakage-resilient pseudorandom number generator from [26].

Implementation of abstract domain and counting We implement a special case of the abstract domain presented in Section 4. Namely, we use a fixed history threshold of $s = 1$, thereby trading precision for efficiency of the analysis. We connect this abstract domain to the CacheAudit platform [9]. CacheAudit takes as input a (32 bit) x86 binary executable, reconstructs the control flow, and uses abstract interpretation to compute an over-approximation of the set of program states (which comprise the cache) that are reachable. The local soundness of our novel domain (stated in Lemma 2), together with the correctness of CacheAudit (stated in Theorem 2) ensures that our analysis soundly over-approximates the set of all concurrent computations.

We further choose f such that the scheduler interrupts *at most once* per round of the pseudorandom number generator. With Corollary 2 we see that the leakage *per key* exceeds the leakage *per round* by a factor of at most three. We can hence obtain a leakage bound by maximising over the number of concretizations of abstract cache states that appear in the fixpoint, which is how we avoid implementing the concurrent counting procedure from Section 4.5 in full generality. We perform the counting of individual cache states using the techniques described in [17].

Implementation of the PRG We implement the $\mathsf{2PRF}$ by concatenating two blocks produced by a block cipher $\mathsf{BC}\colon \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$. More precisely, for an initialization vector $IV \in \{0,1\}^{n-1}$ we compute

$$\mathsf{2PRF}(k) = (\mathsf{BC}_k(0||IV), \mathsf{BC}_k(1||IV)) \ .$$

For our implementation, we instantiate BC with the AES implementation from the PolarSSL library [2], where we use a keylength of 256 bits. We put the key schedule and the two calls to the block cipher in an infinite loop and use gcc to compile this program to a 32-bit x86 executable, which is the artifact we analyze using the techniques developed in this paper.

Experimental results We perform the analysis of the executable on a set-associative cache with LRU replacement strategy, where we consider different cache sizes, line sizes, and associativities. The results of our analysis are given in Table 1.

Columns 5 and 6, respectively, present bounds on the leakage to an concurrent cache adversary per round and per key. Our data show that leakage increases with the cache size and decreases with the line size. The first effect occurs because a larger cache size means that the table is spread out into more cache sets, which increases the resolution with which the adversary can observe the memory accesses of the victim. The second effect occurs because a larger line size decreases the adversary's resolution. Finally, our data shows that greater associativities lead to better bounds.

The entries of Column 6 can be used to instantiate the parameters in Theorem 1, where we consider an adversary with $q = 2^{50}$ and set the amount of

observable data with the same IV to be 1GB, thus $p \leq 2^{25}$. The cryptographic security guarantees we obtain with these parameters are given in column 7; they range from very strong (e.g. 126.1 bits for a 2KB cache with 64B line size) to non-existent (e.g. 8KB cache with 32B line size).

Discussion Column 4 presents an absolute, *program-independent* bound on the number of cache states an adversary can observe in each context switch. Throughout the case study, we consider an adversary whose memory space is disjoint from the victim's, i.e. one who can observe how many memory blocks are loaded in each cache set, but not which. For the example of an 8KB cache with 4 ways and lines of 64B, this number amounts to $(4+1)^{8192/(4*64)}$, where the basis denotes the number of observations per set (0-4 blocks have been loaded into that set by the victim) and the exponent denotes the number of (independent) cache sets.

A comparison between columns 4 and 5 sheds light on the scope of our technique. For caches up to 4KB, the entries in both columns almost coincide. This is due to the fact that the 4KB+256B of tables in the PolarSSL AES implementation entirely fill such small caches, and that the static analysis can only predict that each of the corresponding memory blocks can either be loaded or not. The small difference in leakage stems from the fact that the static analysis *can* predict that the memory blocks containing local variables will be loaded. For caches of 8KB or more the static analysis can moreover determine that the memory access patterns of the executable only affect the memory blocks in which the tables and the local variables reside, hence the bounds obtained by static analysis are significantly better than those obtained by pure combinatorics.

Finally, we remark that there are several timing-relevant features of hardware our approach does not cover (and make assertions about) yet, including out-of-order execution, pipelines, TLBs, and multiple levels of caches. Likewise, implementations of instruction-based scheduling [24] are not yet widely deployed. From a practical perspective, it is currently still wise to rely on implementations that entirely avoid secret-dependent memory lookups, e.g. [1, 6, 15].

6 Conclusions

We have presented the first proof of resilience against side-channel attacks by concurrent cache-based adversaries. To achieve this, we extended existing leakage-resilient cryptosystems to a notion of leakage that captures caches and concurrency, and we developed program analysis techniques for statically deriving formal security guarantees based on executable code.

Acknowledgments This work was partially funded by European Project FP7-256980 NESSoS, by the Spanish Project TIN2012-39391-C04-01 StrongSoft, and by the Madrid Regional Project S2009TIC-1465 PROMETIDOS. This work was partially developed while Martín Ochoa was affiliated with Siemens AG.

Cache size	Line size	Ways	Possible cache observations (bits)	Leakage per round (bits)	Leakage per key (bits)	Security (bits)
2KB	64B	4	18.6	18.3	54.9	126.1
4KB	64B	4	37.2	36.8	110.4	70.6
8KB	64B	4	74.3	55.2	165.6	15.4
16KB	64B	4	148.6	70.2	210.6	0
32KB	64B	4	297.2	73.1	219.3	0
64KB	64B	4	594.41	75.0	225	0
8KB	**32B**	4	148.6	109.5	328.5	0
8KB	**64B**	4	74.3	55.2	165.5	15.4
8KB	**128B**	4	37.2	28.8	86.4	94.6
8KB	64B	**2**	101.4	68.1	204.3	0
8KB	64B	**4**	74.3	55.2	165.6	15.4
8KB	64B	**8**	50.7	39.9	119.7	61.3

Table 1. Leakage about the 2PRF based on the PolarSSL AES 256 implementation, for different cache sizes, line sizes, and associativities. The entries of columns 5 and 6 represent the range of the leakage functions per round and per key, respectively. The entries of column 7 represent bits of security computed using Theorem 1.

References

1. Intel Advanced Encryption Standard (AES) Instructions Set. `http://software.intel.com/file/24917`.
2. PolarSSL. `http://polarssl.org/`.
3. O. Aciiçmez, Ç. K. Koç, and J.-P. Seifert. Predicting secret keys via branch prediction. In *CT-RSA*. Springer, 2007.
4. G. Barthe, G. Betarte, J. D. Campo, and C. Luna. Cache-Leakage Resilient OS Isolation in an Idealized Model of Virtualization. In *CSF*. IEEE, 2012.
5. G. Barthe, B. Grégoire, S. Heraud, and S. Z. Béguelin. Computer-aided security proofs for the working cryptographer. In *CRYPTO*. Springer, 2011.
6. D. Bernstein. Salsa20. `http://cr.yp.to/snuffle.html`.
7. D. J. Bernstein. Cache-timing attacks on AES. Technical report, 2005.
8. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction of approximation of fixpoints. In ACM, editor, *POPL*, 1977.
9. G. Doychev, D. Feld, B. Köpf, L. Mauborgne, and J. Reineke. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. In *USENIX Security Symposium*, 2013.
10. S. Dziembowski and K. Pietrzak. Leakage-Resilient Cryptography. In *FOCS*. IEEE, 2008.
11. C. Ferdinand, F. Martin, R. Wilhelm, and M. Alt. Cache behavior prediction by abstract interpretation. *Science of Computer Programming*, 35(2):163 – 189, 1999.
12. D. Grund. *Static Cache Analysis for Real-Time Systems – LRU, FIFO, PLRU*. PhD thesis, Saarland University, 2012.
13. D. Gullasch, E. Bangerter, and S. Krenn. Cache Games - Bringing Access-Based Cache Attacks on AES to Practice. In *S&P*. IEEE, 2011.

14. S. Jana and V. Shmatikov. Memento: Learning secrets from process footprints. In *S&P*. IEEE, 2012.
15. E. Käsper and P. Schwabe. Faster and timing-attack resistant AES-GCM. In *CHES*. Springer, 2009.
16. T. Kim, M. Peinado, and G. Mainar-Ruiz. StealthMem: System-level protection against cache-based side channel attacks in the cloud. In *19th USENIX Security Symposium*. USENIX, 2012.
17. B. Köpf, L. Mauborgne, and M. Ochoa. Automatic Quantification of Cache Side-Channels. In *CAV*. Springer, 2012.
18. B. Köpf and A. Rybalchenko. Approximation and Randomization for Quantitative Information-Flow Analysis. In *CSF*. IEEE, 2010.
19. S. Micali and L. Reyzin. Physically observable cryptography. In *TCC*. Springer, 2004.
20. D. A. Osvik, A. Shamir, and E. Tromer. Cache Attacks and Countermeasures: the Case of AES. In *CT-RSA*. Springer, 2006.
21. K. Pietrzak. A leakage-resilient mode of operation. In *EUROCRYPT*. Springer, 2009.
22. T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *CCS*. ACM, 2009.
23. V. Shoup. Sequences of games: a tool for taming complexity in security proofs. Cryptology ePrint Archive, Report 2004/332, 2004.
24. D. Stefan, P. Buiras, E. Z. Yang, A. Levy, D. Terei, A. Russo, and D. Mazières. Eliminating cache-based timing attacks with instruction-based scheduling. In *ESORICS*. Springer, 2013.
25. Y. Yu and F.-X. Standaert. Practical leakage-resilient pseudorandom objects with minimum public randomness. In *CT-RSA*. Springer, 2013.
26. Y. Yu, F.-X. Standaert, O. Pereira, and M. Yung. Practical leakage-resilient pseudorandom generators. In *CCS*. ACM, 2010.
27. D. Zhang, A. Askarov, and A. C. Myers. Language-based control and mitigation of timing channels. In *PLDI*. ACM, 2012.

A Appendix

A.1 Example Formalization Cache Update Function

We formally describe $upd_{\mathcal{C}}$ only for the LRU strategy and a single cache set; see [12] for formalizations of other replacement strategies. Upon a cache miss, LRU replaces the least-recently-used memory block. To this end, it tracks the ages of memory blocks within the cache, where the youngest block has age 0 and the oldest cached block has age $k-1$. Thus, the state of the cache can be modeled as a function that assigns an age to each memory block $b \in \mathcal{B}$, where non-cached blocks are assigned age k:

$$\mathcal{C} := \{c \in \mathcal{B} \to A \mid \forall a, b \in \mathcal{B}\colon c(a) \neq c(b) \vee c(a) = c(b) = k\},$$

where $A := \{0, ..., k-1, k\}$ is the set of ages. The constraint encodes that no two blocks can have the same age. For readability we omit the additional constraint

that blocks of non-zero age are preceded by other blocks, i.e. that caches do not contain "holes". The cache update for LRU is then given by

$$upd_{\mathcal{C}}(c,b) := \lambda b' \in \mathcal{B}.\begin{cases} 0 & : b' = b \\ c(b')+1 & : c(b') < c(b) \\ c(b') & : c(b') > c(b) \end{cases}$$

A.2 Local Soundness of Cache Trace Domain

Lemma 2.

$$upd_{\mathcal{T}}(\gamma_{\mathcal{T}}(t^\sharp), M) \subseteq \gamma_{\mathcal{T}}(upd_{\mathcal{T}^\sharp}(t^\sharp, M)),$$

Proof.

$$\begin{aligned}
&upd_{\mathcal{T}}(\gamma_{\mathcal{T}}(c^\sharp_{i,j}), M) = \\
&\quad = \bigcup_{A \in S_n} upd_{\mathcal{T}}(\gamma_A(c^\sharp_{i,j}, M)) \\
&\quad = \bigcup_{n \notin A} \gamma_{\mathcal{C}}(c^\sharp_{0,0}) \ldots \gamma_{\mathcal{C}}(c^\sharp_{n-i_k,n}) upd_{\mathcal{C}}(\gamma(c^\sharp_{n-i_k,n}), M) \\
&\qquad \cup \bigcup_{n \in A} \gamma_A(c^\sharp_{i,j}) upd_{\mathcal{C}}(c_\emptyset, M) \\
&\quad \overset{(*)}{\subseteq} \bigcup_{n \notin A} \gamma_{\mathcal{C}}(c^\sharp_{0,0}) \ldots \gamma_{\mathcal{C}}(c^\sharp_{n-i_k,n}) \gamma_{\mathcal{C}}(upd_{\mathcal{C}^\sharp}(c^\sharp_{n-i_k,n}), M) \\
&\qquad \cup \bigcup_{n+1 \in A} \gamma_A(c^\sharp_{i,j}) upd_{\mathcal{C}^\sharp}(c^\sharp_\emptyset, M) \\
&\quad = \gamma_{\mathcal{T}}(upd_{\mathcal{T}^\sharp}(t^\sharp, M)) \ ,
\end{aligned}$$

where $(*)$ follows from the local soundness of the cache state abstract domain $\mathcal{C}^\sharp$.

A.3 Bounded range leakage and guessing

We recall the following result (see [9] for a proof):

Lemma 4. *Let $X \to Y \to \hat{X}$ be a Markov Chain. Then*

$$\Pr[X = \hat{X}] \leq \max_x \Pr[X = x] \, |\mathsf{ran}(Y)|$$

In particular, if X returns uniformly random n-bit strings, $\Pr[X = \hat{X}] \leq \frac{1}{2^n} \, |\mathsf{ran}(Y)|$

In the context of G_2 in Theorem 1, the key k is a uniformly chosen random n-bit string, and $\Lambda(k)$ the vector of observations of k given to the adversary, corresponding to the variable Y. Therefore, the probability of an adversary of outputting k' such that $k = k'$ is upper bounded by $\frac{1}{2^n} \, |\mathsf{ran}(\Lambda)| \leq \frac{d}{2^n}$.