CacheAudit: A Tool for the Static Analysis of Cache Side Channels

Goran Doychev[1], Dominik Feld[2], Boris Köpf[1], Laurent Mauborgne[1], and Jan Reineke[2]

[1]IMDEA Software Institute
[2]Saarland University

Abstract

We present CacheAudit, a versatile framework for the automatic, static analysis of cache side channels. CacheAudit takes as input a program binary and a cache configuration, and it derives formal, quantitative security guarantees for a comprehensive set of side-channel adversaries, namely those based on observing cache states, traces of hits and misses, and execution times.

Our technical contributions include novel abstractions to efficiently compute precise overapproximations of the possible side-channel observations for each of these adversaries. These approximations then yield upper bounds on the information that is revealed. In case studies we apply CacheAudit to binary executables of algorithms for symmetric encryption and sorting, obtaining the first formal proofs of security for implementations with countermeasures such as preloading and data-independent memory access patterns.

1 Introduction

Side-channel attacks recover secret inputs to programs from non-functional characteristics of computations, such as time [31], power [32], or memory consumption [27]. Typical goals of side-channel attacks are the recovery of cryptographic keys and private information about users.

Processor caches are a particularly rich source of side-channels because their behavior can be monitored in various ways, which is demonstrated by three documented classes of side-channel attacks: (1) In *time-based attacks* [31, 10] the adversary monitors the overall execution time of a victim, which is correlated with the number of cache hits and misses during execution. Time-based attacks are especially daunting because they can be carried out remotely over the network [6]. (2) In *access-based attacks* [40, 39, 23] the adversary probes the cache state by timing its own accesses to memory. Access-based attacks require that attacker and victim share the same hardware platform, which is common in the cloud and has already been exploited [41, 49]. (3) In *trace-based attacks* [5] the adversary monitors the sequence of cache hits and misses. This can be achieved, e.g., by monitoring the CPU's power consumption and is particularly relevant for embedded systems.

A number of proposals have been made for countering cache-based side-channel attacks. Some proposals focus entirely on modifications of the hardware platform; they either solve the problem for specific algorithms such as AES [2] or require modifications to the platform [46] that are so significant that their rapid adoption seems unlikely. The bulk of proposals rely on controlling the interactions between the software and the hardware layers, either through the operating system [23, 48], the client application [10, 39, 15], or both [29]. Reasoning about these interactions can be tricky and error-prone because it relies on the specifics of the binary code and the microarchitecture.

In this paper we present CacheAudit, a tool for the automatic, static exploration of the interactions of a program with the cache. CacheAudit takes as input a program binary and a cache configuration and delivers formal security guarantees that cover all possible executions of the corresponding system. The security guarantees are quantitative upper bounds on the amount of information that is contained in the side-channel observations of timing-, access-, and trace-based adversaries, respectively. CacheAudit can be used to formally analyze the effect on the leakage of software countermeasures and cache configurations, such as preloading of tables or increasing the cache's line size. The design of CacheAudit is modular and facilitates extension with any cache model for which efficient abstractions are in place. The current implementation of CacheAudit supports caches with LRU, FIFO, and PLRU replacement strategies.

We demonstrate the scope of CacheAudit in case studies where we analyze the side-channel leakage of representative algorithms for symmetric encryption and sorting. We highlight the following two results: (1) For the reference implementation of the Salsa20 [11] stream cipher (which was designed to be resilient to cache side-channel attacks) CacheAudit can formally prove non-leakage on the basis of the binary executable, for all adversary models and replacement strategies. (2) For a library implementation of AES 128 [3], CacheAudit confirms that the preloading of tables significantly improves the security of the executable: for most adversary models and replacement strategies, we can in fact prove non-leakage of the executable, whenever the tables fit entirely in the cache. However, for access-based adversaries and LRU caches, CacheAudit reports small, non-zero bounds. And indeed, with LRU (in contrast to, e.g., FIFO), the *ordering* of blocks within a cache set reveals information about the victim's final memory accesses.

On a technical level, we build on the fact that the amount of leaked information corresponds to the number of possible side-channel observations, which can be over-approximated by abstract interpretation[1] and counting techniques [35, 34]. To realize CacheAudit based on this insight, we propose three novel abstract domains (i.e. data structures that approximate properties of the program semantics) that keep track of the observations of access-based, time-based, and trace-based adversaries, respectively. In particular:

1. We propose an abstract domain that tracks relational information about the memory blocks that may be cached. In contrast to existing abstract domains used in worst-case execution time analysis [21], our novel domain can retain analysis precision in the presence of array accesses to unknown positions.

2. We propose an abstract domain that tracks the traces of cache hits and misses that may occur during execution. We use a technique based on prefix trees and hash consing to compactly represent such sets of traces, and to count their number.

3. We propose an abstract domain that tracks the possible execution times of a program. This domain captures timing variations due to control flow and caches by associating hits and misses with their respective latencies and adding the execution time of the respective commands.

We formalize the connection of these domains in an abstract interpretation framework that captures the relationship between microarchitectural state and program code. We use this framework to formally prove the correctness of the derived upper bounds on the leakage to the corresponding side-channel adversaries.

In summary, our main contributions are both theoretical and practical: On a theoretical level, we define novel abstract domains that are suitable for the analysis of cache side channels, for a comprehensive set of adversaries. On a practical level, we build CacheAudit, the first tool for the automatic, quantitative information-flow analysis of cache side-channels, and we show how it can be used to derive formal security guarantees from binary executables of sorting algorithms and state-of-the-art cryptosystems.

[1]A theory of sound approximation of program semantics [16]

Outline The remainder of the paper is structured as follows. In Section 2, we illustrate the power of CacheAudit on a simple example program. In Section 3 we define the semantics and side channels of programs. We describe the analysis framework, the design of CacheAudit, and the novel abstract domains in Sections 4, 5 and 6, respectively. We present experimental results in Section 7, before we discuss prior work and conclude in Sections 8 and 9. The source code and documentation of CacheAudit are available at

`http://software.imdea.org/cacheaudit`

2 Illustrative Example

In this section, we illustrate on a simple example program the kind of guarantees CacheAudit can derive. Namely, we consider an implementation of BubbleSort that receives its input in an array `a` of length `n`. We assume that the contents of `a` are secret and we aim to deduce how much information a cache side-channel adversary can learn about the relative ordering of the elements of `a`.

```
1   void BubbleSort(int a[], int n)
2   {
3    int i, j, temp;
4    for (i = 0; i < n - 1; ++i)
5       for (j = 0; j < n - 1 - i; ++j)
6          if (a[j] > a[j+1])
7          {
8             temp = a[j+1];
9             a[j+1] = a[j];
10            a[j] = temp;
11         }
12  }
```

To begin with, observe that the conditional swap in lines 6–11 is executed exactly $\frac{n(n-1)}{2}$ times. A *trace-based* adversary that can observe, for each instruction, whether it corresponds to a cache hit or a miss is likely to be able to distinguish between the two alternative paths in the conditional swap, hence we expect this adversary to be able to distinguish between $2^{\frac{n(n-1)}{2}}$ execution traces. A *timing-based* adversary who can observe the overall execution time is likely to be able to distinguish between $\frac{n(n-1)}{2}+1$ possible execution times, corresponding to the number of times the swap has been carried out. For an *access-based* adversary who can probe the final cache state upon termination, the situation is more subtle: evaluating the guard in line 6 requires accessing both `a[j]` and `a[j+1]`, which implies that both will be present in the cache when the swap in lines 8–10 is carried out. Assuming we begin with an empty cache, we expect that there is only one possible final cache state.

CacheAudit enables us to perform such analyses (for a particular `n`) formally and automatically, based on actual x86 binary executables and different cache types. CacheAudit achieves this by tracking compact representations of supersets of possible cache states and traces of hits and misses, and by counting the corresponding number of elements. For the above example, CacheAudit was able to precisely confirm the intuitive bounds, for a selection of several `n` in $\{2,\dots,64\}$.

In terms of security, the number of possible observations corresponds to the factor by which the cache observation increases the probability of correctly guessing the secret ordering of inputs. Hence, for $\mathtt{n} = 32$ and a uniform distribution on this order (i.e. an initial probability of $\frac{1}{32!} = 3.8 \cdot 10^{-36}$), the bounds derived by CacheAudit imply that the probability of determining the correct input order from the side-channel observation is 1 for a trace-based adversary, $3.7 \cdot 10^{-33}$ for a time-based adversary, and remains $\frac{1}{32!}$ for an access-based adversary.

3 Caches, Programs, and Side Channels

3.1 A Primer on Caches

Caches are fast but small memories that store a subset of the main memory's contents to bridge the latency gap between the CPU and main memory. To profit from spatial locality and to reduce management overhead, main memory is logically partitioned into a set of *memory blocks* $\mathcal{B}$. Each block is cached as a whole in a cache line of the same size.

When accessing a memory block, the cache logic has to determine whether the block is stored in the cache ("cache hit") or not ("cache miss"). To enable an efficient look-up, each block can only be stored in a small number of cache lines. For this purpose, caches are partitioned into equally-sized *cache sets*. The size of a cache set is called the *associativity* k of the cache. There is a function *set* that determines the cache set a memory block maps to.

Since the cache is much smaller than main memory, a *replacement policy* must decide which memory block to replace upon a cache miss. Usually, replacement policies treat sets independently, so that accesses to one set do not influence replacement decisions in other sets. Well-known replacement policies in this class are least-recently used (LRU), used in various Freescale processors such as the MPC603E and the TriCore17xx; pseudo-LRU (PLRU), a cost-efficient variant of LRU, used in the Freescale MPC750 family and multiple Intel microarchitectures; and first-in first-out (FIFO), also known as ROUND ROBIN, used in several ARM and Freescale processors such as the ARM922 and the Freescale MPC7450 family. A more comprehensive overview can be found in [22].

3.2 Programs and Computations

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

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

A *computation* of P is an alternating sequence of states and events $\sigma_0 e_0 \sigma_1 e_1 \dots \sigma_n$ such that $\sigma_0 \in I$ and that for all $i \in \{0,\dots,n-1\}$, $(\sigma_i, e_i, \sigma_{i+1}) \in \mathcal{T}$. The set of all computations of P is its *trace collecting semantics* $Col(P) \subseteq Traces$, where *Traces* denotes the set of all alternating sequences of states and events. When considering terminating programs, the trace collecting semantics can be formally defined as the least fixpoint of the *next* operator containing I:

$$Col(P) = I \cup next(I) \cup next^2(I) \cup \dots ,$$

where *next* describes the effect of one computation step:

$$next(S) = \{t.\sigma_n e_n \sigma_{n+1} \mid t.\sigma_n \in S \wedge (\sigma_n, e_n, \sigma_{n+1}) \in \mathcal{T}\}$$

In the rest of the paper, we assume that P is fixed and abbreviate its trace collecting semantics by *Col*.

3.3 Cache Updates and Cache Effects

For reasoning about cache side channels, we consider a semantics in which the cache is part of the program state. Namely, the state will consist of logical memories in $\mathcal{M}$ (representing the values of main memory locations and CPU registers, including the program counter) and a cache state in $\mathcal{C}$, i.e., $\Sigma = \mathcal{M} \times \mathcal{C}$.

The *memory update* $upd_{\mathcal{M}}$ is a function $upd_{\mathcal{M}} : \mathcal{M} \to \mathcal{M}$ that is determined solely by the instruction set semantics. The memory update has effects on the cache that are described by a function $eff_{\mathcal{M}} : \mathcal{M} \to \mathcal{E}_{\mathcal{M}}$. The memory effect is an argument to the *cache update* function $upd_{\mathcal{C}} : \mathcal{C} \times \mathcal{E}_{\mathcal{M}} \to \mathcal{C}$.

In the setting of this paper, $eff_{\mathcal{M}}$ determines which block of main memory is accessed, which is required to compute the cache update $upd_{\mathcal{C}}$, i.e., $\mathcal{E}_{\mathcal{M}} = \mathcal{B} \cup \{\perp\}$, where $\perp$ denotes that no memory block is accessed.

We formally describe $upd_{\mathcal{C}}$ only for the LRU strategy. For formalizations of other strategies, see [22]. Upon a cache miss, LRU replaces the least-recently-used memory block. To this end, it tracks the ages of memory blocks within each cache set, 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, where non-cached blocks are assigned age k:

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

where $A := \{0,...,k-1,k\}$ is the set of ages. The constraint encodes that no two blocks in the same cache set 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 cache sets do not contain "holes".

The cache update for LRU is then given by

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

In the setting of this paper, the events $\mathcal{E}$ consist of cache hits and misses, which are described by the cache effect $\mathit{eff}_{\mathcal{C}} : \mathcal{C} \times \mathcal{B} \to \mathcal{E}$:

$$\mathit{eff}_{\mathcal{C}}(c,m) := \begin{cases} hit & : c(m) < k \\ miss & : \text{else} \end{cases}$$

Both $upd_{\mathcal{C}}$ and $\mathit{eff}_{\mathcal{C}}$ are naturally extended to the case where no memory access occurs. Then, the cache state remains unchanged and the cache effect is $\perp$, so $\mathcal{E} = \{hit, miss, \perp\}$.

With this, we can now connect the components and obtain the global transition relation $\mathcal{T} \subseteq \Sigma \times \mathcal{E} \times \Sigma$ by

$$\begin{aligned} \mathcal{T} = \{((m_1,c_1),\ e\ ,(m_2,c_2)) \mid\ & m_2 = upd_{\mathcal{M}}(m_1) \\ \wedge\ & c_2 = upd_{\mathcal{C}}(c_1, \mathit{eff}_{\mathcal{M}}(m_1)) \\ \wedge\ & e = \mathit{eff}_{\mathcal{C}}(c_1, \mathit{eff}_{\mathcal{M}}(m_1))\}\ , \end{aligned}$$

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

3.4 Side Channels

For a deterministic, terminating program P, the transition relation is a function, and the program can be modeled as a mapping $P \colon I \to Col$.

We model an adversary's view on the computations of P as a function $view \colon Col \to O$ that maps computations to a finite set of observations O. The composition

$$C = (view \circ P) \colon I \to O$$

defines a function from initial states to observations, which we call a *channel* of P. Whenever *view* is determined by the cache and event components of traces, we call C a *side channel* of P.

We next define views corresponding to the observations of access-based, trace-based, and timing-based side-channel adversaries.

The view of an *access-based* adversary that shares the memory space with the victim is defined by

$$view^{acc} \colon (m_0,c_0)e_0 \dots e_{n-1}(m_n,c_n) \mapsto c_n$$

and captures that the adversary can determine (by probing) which memory blocks are contained in the cache upon termination of the victim. An adversary that does *not* share the memory space with the victim can only observe how many blocks the victim has loaded in each cache set (by probing how many of its own blocks have been evicted), but not which. We denote this view by $view^{accd}$. The view of a *trace-based* adversary is defined by

$$view^{tr} \colon \sigma_0 e_0 \dots e_{n-1} \sigma_n \mapsto e_0 \dots e_{n-1}$$

and captures that the adversary can determine for each instruction whether it results in a hit, miss, or does not access memory. The view of a *time-based* adversary is defined by

$$\begin{aligned} view^{time} \colon\ & \sigma_0 e_0 \dots e_{n-1} \sigma_n \mapsto \\ & t_{hit} \cdot |\{i \mid e_i = hit\}| + t_{miss} \cdot |\{i \mid e_i = miss\}| + \\ & t_{\perp} \cdot |\{i \mid e_i = \perp\}| \end{aligned}$$

and captures that the adversary can determine the overall execution time of the program. Here, t_{hit}, t_{miss}, and $t_{\perp}$ are the execution times (e.g. in clock cycles) of instructions that imply cache hits, cache misses, or no memory accesses at all. While the view of the time-based adversary as defined above is rather simplistic, e.g. disregarding effects of pipelining and out-of-order execution, notice that our semantics and our tool can be extended to cater for a more fine-grained, instruction- and context-dependent modeling of execution times. We denote the side channels corresponding to the four views by C^{acc}, C^{accd}, C^{tr}, and C^{time}, respectively. Figure 1 gives an overview.

3.5 Quantification of Side Channels

We characterize the security of a channel $C \colon I \to O$ as the difficulty of guessing the secret input from the channel output.

C^{acc}	Access-based adversary whose memory space is shared with the victim's.
C^{accd}	Access-based adversary whose memory space is disjoint from the victim's.
C^{tr}	Adversary who observes the trace of cache hits and misses.
C^{time}	Adversary who observes the overall execution time.

Figure 1: Channels corresponding to different adversary models.

Formally, we model the choice of a secret input by a random variable X with $ran(X) \subseteq I$ and the corresponding observation by a random variable $C(X)$ with $ran(C(X)) \subseteq O$. We model the attacker as another random variable $\hat{X}$. The goal of the attacker is to estimate the value of X, i.e. it is successful if $\hat{X} = X$. We make the assumption that the attacker does not have information about the value of X beyond what is contained in $C(X)$, which we formalize as the requirement that $X \to C(X) \to \hat{X}$ form a Markov chain. The following theorem expresses a security guarantee as an upper bound on the attacker's success probability in terms of the size of the range of C.

Theorem 1. *Let $X \to C(X) \to \hat{X}$ be a Markov chain. Then*

$$P(X = \hat{X}) \leq \max_{\sigma \in I} P(X = \sigma) \cdot |ran(C)|$$

For the interpretation of the statement observe that if the adversary has no information about the value of X (i.e., if $\hat{X}$ and X are statistically independent), its success probability is bounded by the probability of the most likely value of X, i.e. $P(X = \hat{X}) \leq \max_{\sigma \in I} P(X = \sigma)$, where equality can be achieved. Theorem 1 hence states that the size of the range of C is an upper bound on the factor by which this probability is increased when the attacker sees $C(X)$ and is, in that sense, an upper bound for the amount of information leaked by C. We will often give bounds on $|ran(C)|$ on a log-scale, in which case they represent upper bounds on the number of leaked *bits*. Notice that the guarantees of Theorem 1 fundamentally rely on assumptions about the initial distribution of X: if X is easy to guess to begin with, Theorem 1 does not imply meaningful security guarantees.

For more discussion on the interpretation of the security guarantees, see Section 7.4. For a formal connection to traditional (entropy-based) presentations of quantitative information-flow analysis [43] and a proof of Theorem 1, see the extended version [19].

3.6 Adversarially Chosen Cache States

We sometimes assume that initial states are pairs consisting of *high* and *low* components, i.e. $I = I_{hi} \times I_{lo}$, where only the high component is meant to be kept secret and the low component may be provided by the adversary, a common setting in information-flow analysis [42]. In this case, a program and a view define a *family* of channels $C_{\sigma_{lo}} : I_{hi} \to O$, one for each low component $\sigma_{lo} \in I_{lo}$.

A particularly interesting instance is the decomposition into secret memory $I_{hi} = \mathcal{M}$ and adversarially chosen cache $I_{lo} = \mathcal{C}$. While bounds for the corresponding channel can be derived by considering all possible initial cache states, corresponding analyses suffer from poor precision. The following lemma enables us to derive bounds for the general case, based on the empty cache state.

Lemma 1. *For all initial cache states $c \in \mathcal{C}$, adversaries $adv \in \{acc, accd, time, tr\}$, and LRU, FIFO, or PLRU replacement: If no block in c is accessed during program execution, then*

$$\left| ran(C^{adv}_{\emptyset}) \right| = \left| ran(C^{adv}_{c}) \right| \,, \qquad (1)$$

where $\emptyset$ is a shorthand for the empty cache state. For $adv \in \{acc, accd\}$ and LRU, $\left|ran(C^{adv}_{\emptyset})\right| \geq \left|ran(C^{adv}_{c})\right|$ holds without any constraints on the initial cache state c.

This lemma was proved in [34] for *acc*, *accd* and the LRU case with the initial cache state not containing any block of the victim. The proof is based on the fact that memory blocks in the cache do not affect the position of memory blocks that are accessed during computation whenever the two sets of memory blocks are disjoint, which allows us to construct a bijective function from $ran(C^{adv}_{\emptyset})$ to $ran(C^{adv}_{c})$. The argument immediately extends to FIFO, PLRU, and all *adv*. For LRU and access-based adversaries, the function remains surjective even without the disjointness requirement.

4 Automatic Quantification of Cache Side Channels

Theorem 1 enables the quantification of side channels by determining their range. As channels are defined in terms of views on computations, their range can be determined by computing *Col* and applying *view*. However, this entails computing a fixpoint of the *next* operator and is practically infeasible in most cases. Abstract interpretation [16] overcomes this fundamental problem by computing a fixpoint with respect to an efficiently computable over-approximation of *next*. This new fixpoint represents a superset of all computations, which is sufficient for deriving an upper bound on the range of the channel and thus on the leaked information.

In this section, we describe the interplay of the abstractions used for over-approximating *next* in CacheAudit (namely, those for memory, cache, and events), and we explain how the global soundness of CacheAudit can be established from local soundness conditions. This modularity is key for the future extension of CacheAudit using more advanced abstractions. Our results hold for all adversaries introduced in Section 3.4 and we omit the superscript *adv* from channels and views for readability.

4.1 Sound Abstraction of Leakage

We frame a static analysis by defining a set of abstract elements $Traces^\sharp$ together with an abstract transfer function $next^\sharp : Traces^\sharp \to Traces^\sharp$. Here, the elements $a \in Traces^\sharp$ represent subsets of *Traces*, which is formalized by a concretization function

$$\gamma : Traces^\sharp \to \mathcal{P}(Traces) \; .$$

The key requirements for $next^\sharp$ are (1) that it be efficiently computable, and (2) that it over-approximates the effect of *next* on sets of computations, which is formalized as the following local soundness condition:

$$\forall a \in Traces^\sharp : next\,(\gamma(a)) \subseteq \gamma(next^\sharp(a)) \; . \tag{2}$$

Intuitively, if we maintain a superset of the set of computations during each step of the transfer function as in (2), then this inclusion must also hold for the corresponding fixpoints. More formally, any post-fixpoint of $next^\sharp$ that is greater than an abstraction of the initial states I is a sound over-approximation of the collecting semantics. We use $Col^\sharp$ to denote any such post-fixpoint.

Theorem 2 (Local soundness implies global soundness, from [16])**.** *If* (2) *holds then*

$$Col \subseteq \gamma\left(Col^\sharp\right) .$$

The following theorem is an immediate consequence of Theorem 2 and the fact that $view\,(Col) = ran(C)$. It states that a sound abstract analysis can be used for deriving bounds on the size of the range of a channel.

Theorem 3 (Upper bounds on leakage)**.**

$$|ran(C)| \leq \left| view\left(\gamma\left(Col^\sharp\right)\right)\right| \; .$$

With the help of Theorem 1, these bounds immediately translate into security guarantees. The relationship of all steps leading to these guarantees is depicted in Figure 2.

4.2 Abstraction Using a Control Flow Graph

In order to come up with a tractable and modular analysis, we design independent abstractions for cache states, memory, and sequences of events.

- $\mathcal{M}^\sharp$ abstracts memory and $\gamma_{\mathcal{M}} : \mathcal{M}^\sharp \to \mathcal{P}(\mathcal{M})$ formalizes its meaning.
- $\mathcal{C}^\sharp$ abstracts cache configurations and $\gamma_{\mathcal{C}} : \mathcal{C}^\sharp \to \mathcal{P}(\mathcal{C})$ formalizes its meaning.
- $\mathcal{E}^\sharp$ abstracts sequences of events and $\gamma_{\mathcal{E}} : \mathcal{E}^\sharp \to \mathcal{P}(\mathcal{E}^*)$ formalizes its meaning.

But, since cache updates and events depend on memory state, independent analyses would be too imprecise. In order to maintain some of the relations, we link the three abstract domains for memory state, caches, and events through a finite set of labels L so that our abstract domain is

$$Traces^\sharp = L \to \mathcal{M}^\sharp \times \mathcal{C}^\sharp \times \mathcal{E}^\sharp \; ,$$

where we write $a^{\mathcal{M}}(l)$, $a^{\mathcal{C}}(l)$ and $a^{\mathcal{E}}(l)$ for the first, second, and third components of an abstract element $a(l)$.

Labels roughly correspond to nodes in a control flow graph in classical data-flow analyses. One could simply use program locations as labels. But in our setting, we use more general labels, allowing for a more fine-grained analysis in which we can distinguish values of flags or results of previous tests [36]. To capture that, we associate a meaning with each label via a function $\gamma_L : L \to \mathcal{P}(Traces)$. If the labels are program locations, then $\gamma_L(l)$ is the set of traces ending in a state in location l. The analogy with control flow graphs can be extended to edges of that graph: using the *next* operator, we define the successors and predecessors of a location l as: $succ(l) = \{k \mid next(\gamma_L(l)) \cap \gamma_L(k) \neq \emptyset\}$, and $pred(l) = \{k \mid next(\gamma_L(k)) \cap \gamma_L(l) \neq \emptyset\}$.

Then we can describe the meaning of an element $a \in Traces^\sharp$ with:

$$\begin{aligned}\gamma(a) = \{\sigma_0 e_0 \sigma_1 \ldots \sigma_n \in Traces \mid \forall i \leq n, \forall l \in L :\\ \sigma_0 e_0 \sigma_1 \ldots \sigma_i \in \gamma_L(l) \Rightarrow\\ \sigma_i^{\mathcal{M}} \in \gamma_{\mathcal{M}}(a^{\mathcal{M}}(l)) \wedge \sigma_i^{\mathcal{C}} \in \gamma_{\mathcal{C}}(a^{\mathcal{C}}(l))\\ \wedge e_0 \ldots e_{i-1} \in \gamma_{\mathcal{E}}(a^{\mathcal{E}}(l))\}\end{aligned} \tag{3}$$

That is, the meaning of an $a \in Traces^\sharp$ is the set of traces, such that for every prefix of a trace, if it "ends" at program location l, then the memory state, cache state, and the event sequence satisfy the respective abstract elements for that location.

The abstract transfer function $next^\sharp$ will be decomposed into:

$$next^\sharp(a) = \lambda l.\,(next_{\mathcal{M}^\sharp}(a,l), next_{\mathcal{C}^\sharp}(a,l), next_{\mathcal{E}^\sharp}(a,l)) \; , \tag{4}$$

$$\begin{tikzcd}
 & Col \arrow[r, phantom, "\overset{\text{Theorem 2}}{\subseteq}"] \arrow[d] & \gamma\left(Col^\sharp\right) \arrow[d] & Col^\sharp \arrow[l, "\text{Meaning}"'] \\
\text{Leakage} \arrow[r, phantom, "\overset{\text{Theorem 1}}{\leq}"] & |ran(C)| = |view\,(Col)| \arrow[r, phantom, "\overset{\text{Monotonicity}}{\leq}"] & \left|view\left(\gamma\left(Col^\sharp\right)\right)\right| &
\end{tikzcd}$$

Figure 2: Relationship of collecting semantics Col, abstract fixpoint $Col^\sharp$, side channels C, and leakage bounds.

where each next function over-approximates the corresponding concrete update function defined in the previous section. The effects used for defining the concrete updates are reflected as information flow between otherwise independent abstract domains, which is formalized as a *partial reduction* in the abstract interpretation literature [18].

4.3 Local Soundness

The products and powers of sound abstract domains with partial reductions are again sound abstract domains [17]. The soundness of $Traces^\sharp$ hence immediately follows from the local soundness of the memory, cache and event domains. Below we describe those soundness conditions for each domain.

The abstract $next^\sharp$ operation is implemented using local update functions for the memory, cache, and event components. For the memory domain we have, for each label $k \in L$ and each $l \in succ(k)$:

- an abstract memory update $upd_{\mathcal{M}^\sharp,(k,l)} : \mathcal{M}^\sharp \to \mathcal{M}^\sharp$, and
- an abstract memory effect $eff_{\mathcal{M}^\sharp,(k,l)} : \mathcal{M}^\sharp \to \mathcal{P}(\mathcal{E}_\mathcal{M})$.

For the cache domain, there is no need for separate functions for each pair (k,l), because the cache update only depends on the accessed block which is delivered by the abstract memory effect. Likewise, the update of the event domain only depends on the abstract cache effect. Thus, we further have:

- an abstract cache update $upd_{\mathcal{C}^\sharp} : \mathcal{C}^\sharp \times \mathcal{P}(\mathcal{E}_\mathcal{M}) \to \mathcal{C}^\sharp$,
- an abstract cache effect $eff_{\mathcal{C}^\sharp} : \mathcal{C}^\sharp \times \mathcal{P}(\mathcal{E}_\mathcal{M}) \to \mathcal{P}(\mathcal{E}_\mathcal{C})$, and
- an abstract event $upd_{\mathcal{E}^\sharp} : \mathcal{E}^\sharp \times \mathcal{P}(\mathcal{E}_\mathcal{C}) \to \mathcal{E}^\sharp$.

With these functions, we can approximate the effect of *next* on each label l, using the abstract values associated with the labels that can lead to l, $pred(l)$. For the example of the cache domain, this yields

$$next_{\mathcal{C}^\sharp}(a,l) = \bigsqcup_{k \in pred(l)}^{\mathcal{C}^\sharp} upd_{\mathcal{C}^\sharp}\left(a^\mathcal{C}(k), eff_{\mathcal{M}^\sharp,(k,l)}(a^\mathcal{M}(k))\right),$$

where $\bigsqcup^{\mathcal{C}^\sharp}$ refers to the join function and can be thought of as set union. That is, $next_{\mathcal{C}^\sharp}(a,l)$ collects all cache states that can reach l within one transition when updated with an over-approximation of the corresponding memory blocks. See the full version [19] for a description of the corresponding update functions for memory and effects.

Now from Equations 2, 3, and 4, we can derive conditions for each domain that are sufficient to guarantee local soundness for the whole analysis:

Definition 1 (Local soundness of abstract domains)**.** *The abstract domains are locally sound if the abstract joins are over-approximations of unions, and if for any function* $f^\sharp \in \{upd_{\mathcal{M}^\sharp,(k,l)}, eff_{\mathcal{M}^\sharp,(k,l)}, upd_{\mathcal{C}^\sharp}, eff_{\mathcal{C}^\sharp}, upd_{\mathcal{E}^\sharp}\}$ *approximating concrete function* $f \in \{upd_\mathcal{M}, eff_\mathcal{M}, upd_\mathcal{C}, eff_\mathcal{C}, next\}$ *and corresponding meaning function* γ_f*, we have for any abstract value x:*

$$\gamma_f\left(f^\sharp(x)\right) \supseteq f\left(\gamma_f(x)\right).$$

For example, for the cache abstract domain, we have the following local soundness conditions:

$$\begin{aligned}
&\forall c^\sharp \in \mathcal{C}^\sharp, M \in \mathcal{P}(\mathcal{E}_\mathcal{M}) : \\
&\qquad \gamma_\mathcal{C}(upd_{\mathcal{C}^\sharp}(c^\sharp, M)) \supseteq upd_\mathcal{C}(\gamma_\mathcal{C}(c^\sharp), M), \\
&\qquad\qquad eff_{\mathcal{C}^\sharp}(c^\sharp, M) \supseteq eff_\mathcal{C}(\gamma_\mathcal{C}(c^\sharp), M),
\end{aligned}$$

$$\forall \mathcal{G}^\sharp \subseteq \mathcal{C}^\sharp : \ \gamma_\mathcal{C}\left(\bigsqcup^{\mathcal{C}^\sharp} \mathcal{G}^\sharp\right) \supseteq \bigcup_{G^\sharp \in \mathcal{G}^\sharp} \gamma_\mathcal{C}\left(G^\sharp\right).$$

Lemma 2 (Local Soundness Conditions)**.** *If local soundness holds on the abstract memory, cache, and events domains, then the corresponding* $next^\sharp$ *function satisfies local soundness.*

Due to the above lemma, abstract domains for the memory, cache, and events can be separately developed and proven correct. We exploit this fact in this paper, and we plan to develop further abstractions in the future, targeting different classes of adversaries or improving precision.

4.4 Soundness of Delivered Bounds

We implemented the framework described above in a tool named CacheAudit. Thanks to the previous results, CacheAudit provides the following guarantees.

Theorem 4. *The bounds derived by CacheAudit soundly over-approximate* $\left|ran(C^{adv})\right|$, *for* $adv \in \{acc, accd, tr, time\}$, *and hence correspond to upper bounds on the maximal amount of leaked information.*

The statement is an immediate consequence of combining Lemma 2 with Theorems 2 and 3, under the assumption that all involved abstract domains satisfy local soundness conditions, and that the corresponding counting procedures are correct. We formally prove the validity of these assumptions only for our novel relational and trace domains (see Section 6). For the other domains, corresponding proofs are either standard (e.g. the value domain) or out of scope of this submission.

5 Tool Design and Implementation

In this section we describe the architecture and implementation of CacheAudit.

We take advantage of the compositionality of the framework described in Section 4 and use a generic iterator module to compute fixpoints, where we rely on independent modules for the abstract domains that correspond to the components of the $next^\sharp$ operation. Figure 3 depicts the overall architecture of CacheAudit, with the individual modules described below.

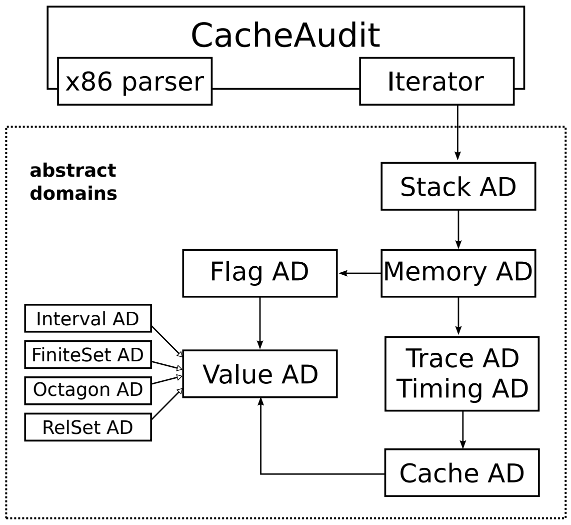

Figure 3: The architecture of CacheAudit. The solid boxes represent modules. Black-headed arrows mean that the module at the head is an argument of the module at the tail. White-headed arrows represent is-a relationships.

5.1 Control Flow Reconstruction

The first stage of the analysis is similar to a compiler front end. The main challenge is that we directly analyze x86 executables with no explicit control flow graph, which we need for guiding the fixpoint computation.

For the parsing phase, we rely on Chlipala's parser for x86 executables [13], which we extend to a set of instructions that is sufficient for our case studies (but not yet complete). For the control-flow reconstruction, we consider only programs without dynamically computed jump and call targets, which is why it suffices to identify the basic blocks and link them according to the corresponding branching conditions and (static) branch targets. We plan to integrate more sophisticated techniques for control-flow reconstruction [30] in the future.

5.2 Iterator

The iterator module is responsible for the computation of the $next^\sharp$ operator and of the approximation of its fixpoint using adequate iteration strategies [17]. Our analysis uses an *iterative* strategy, i.e., it stabilizes components of the abstract control flow graph according to a weak topological ordering, which we compute using Bourdoncle's algorithm [12].

The iterator also implements parts of the reduced cardinal power, based on the labels computed according to the control-flow graph: Each label is associated with an initial abstract state. The analysis computes the effect of the commands executed from that label to its successors on the initial abstract state, and propagates the resulting final states using the abstract domains described below. In order to increase precision, we expand locations using loop unfolding, so that we have a number of different initial and final abstract states for each label inside loops, depending on a parameter describing the number of loop unfoldings we want to perform. Most of our examples (such as the cryptographic algorithms) require only a small, constant number of loop iterations, so that we can choose unfolding parameters that avoid joining states stemming from different iterations.

5.3 Abstract Domains

As described in Section 4, we decompose the abstract domain used by the iterator into mostly independent abstract domains describing different aspects of the concrete semantics.

Value Abstract Domains A value abstract domain represents sets of mappings from variables to (integer) values. Value abstract domains are used by the cache abstract domain to represent ages of blocks in the cache, and by the flag abstract domain to represent values stored at the addresses used in the program. We have implemented different value abstract domains, such as the interval domain, an exact finite sets domain (where the sets become intervals when they are growing too large) and a relational set domain (as described in Section 6.1).

Flag Abstract Domain In x86 binaries, there are no high level guards: instead, most operations modify flags which are then queried in conditional branches. In order to deal precisely with such branches, we need to record relational information between the values of variables and the values of these flags. To that end, for each operation that modifies the flags, we compute an over-approximation of the values of the arguments that may lead to a particular flag combination. The flag abstract domain represents an abstract state as a mapping from values of flags to elements of the value abstract domain. When the analysis reaches a conditional branch, it can identify which combination of flag values corresponds to the branch and propagate the appropriate abstract values.

Memory Abstract Domain The memory abstract domain associates memory addresses and registers with variables and translates machine instructions into the corresponding operations on those variables, which are represented using flag abstract domains as described above. One important aspect for efficiency is that variables corresponding to addresses are created dynamically during the analysis whenever they are needed. The memory abstract domain further records all accesses to main memory using a cache abstract domain, as described below.

Stack Abstract Domain Operations on the stack are handled by a dedicated stack abstract domain. In this way the memory abstract domain does not have to deal with stack operations such as procedure calls, for which special techniques can be implemented to achieve precise interprocedural analysis.

Cache Abstract Domain The cache abstract domain only tracks information about the cache state. We represent this state by sets of mappings from blocks to ages in the cache, which we implement using an instance of value abstract domains. Effects from the memory domain are passed to the cache domain through the trace domain. The cache abstract domain tracks which addresses are touched during computation and returns information about the presence or absence of cache hits and misses to the trace abstract domain, which we present in Section 6.2. The timings are then obtained as an abstraction from the traces.

6 Abstract Domains for Cache Adversaries

6.1 Cache State Domains

Abstractions of cache states are at the heart of analyses for all three cache adversaries considered in this paper. Thus, precise abstraction of cache states is crucial to determine tight leakage bounds.

The current state-of-the-art abstraction for LRU replacement by Ferdinand et al. [21] maintains an upper and a lower bound on the age of every memory block. This abstraction was developed with the sole goal of classifying memory accesses as cache hits or cache misses. In contrast, our goal is to develop abstractions that yield tight bounds on the maximal leakage of a channel. For access-based adversaries the leakage is bounded by the size of the concretization of an abstract cache state, i.e. the size of the set of concrete cache states represented by the abstract state.

Intuition behind Relational Sets To derive tighter leakage bounds, we propose a new domain called *relational sets* that improves previous work along two dimensions:

1. Instead of *intervals* of ages of memory blocks, we maintain *sets* of ages of memory blocks.
2. Instead of maintaining *independent* information about the age of each memory blocks, we record the *relation* between ages of different memory blocks.

In addition to increasing precision, moving from intervals to sets allows us to analyze caches with FIFO and PLRU replacement. Interval-based analysis of FIFO and PLRU has been shown to be rather imprecise in the context of worst-case execution time analysis [24].

Motivating Example Consider the following method, which performs a table lookup based on a secret input, as it may occur in e.g. an AES implementation:

```
unsigned int A[size];

int getElement(int secret) {
    if (secret < size)
        return A[secret];
}
```

Assume we want to determine the possible cache states after one invocation of `getElement`. As the value of `secret` is unknown to the analysis, every memory location of the array might be accessed.

`size`	8	16	32	64	128	256
LRU/*IV*	1	2	4	8	16	32
LRU/*Set*	1	2	4	8	16	32
LRU/*Rel*	1	1.58	2.32	3.17	4.01	5.04

Figure 4: Bounds on the number of leaked bits about the parameter *secret* for varying array sizes. The cache parameters are fixed, with a block size of 32 bytes, associativity 4 and cache size 4 KB.

Assuming the array was not cached before the invocation of `getElement`, the interval-based domain by Ferdinand et al. [21] determines a lower bound of 0 and an upper bound of k on the age of each array element.

By tracking sets instead of intervals of ages for each memory block, we would get 0 and k as possible ages of each array element.

Both non-relational domains, however, are not powerful enough to infer or even express the fact, that *only one* of the array's memory blocks has been accessed, and can thus be cached. Therefore, the number of possible cache states represented by non-relational abstractions grows exponentially in the size of the array, while the actual number of possible cache states only grows linearly.

A relational domain, tracking the possible ages of, e.g., pairs of memory blocks, would indeed yield a linear growth in the number of possible cache states. For each pair of array elements, it would be able to infer that only one of the two blocks may be cached. From this, it follows that only one of all of the array elements may be cached.

Figure 4 shows experimental results for the example program with three domains: the interval domain (*IV*), and two instances of the *relational sets* domain, tracking sets of ages of individual blocks (*Set*) and sets of ages of pairs of blocks (*Rel*), respectively.

We do not see an improvement of *sets* over *intervals* in this particular example, as the information that a block has either age 0 or age k can be inferred from the intervals in the counting procedure. This is because the considered arrays are small and thus no two array elements map to the same cache set. We have, however, observed in case studies that *sets* alone often improve over *intervals*.

A detailed formalization of relational sets and their implementation, including efficient counting, is provided in the extended version of this paper [19]. There, we also show that the domain is locally sound according to Definition 1:

Lemma 3. *The relational sets domain is locally sound.*

6.2 A Trace Domain

We devise an abstract domain for keeping track of the sets of event traces that may occur during the execution of a program. Following the way events are computed in the concrete, namely as a function from cache states and memory effects (see Section 3.3), the abstract cache domain provides abstract cache effects.

In our current implementation of CacheAudit, we use an exact representation for sets of event traces: we can represent any finite set of event traces, and assuming an incoming set of traces $\mathcal{S}$ and a set of cache effects E, we compute the resulting event set precisely as follows:

$$upd_{\mathcal{E}^\sharp}(\mathcal{S},E) = \{\sigma.e \mid \sigma \in \mathcal{S} \wedge e \in E\}$$

Then soundness is obvious, since the abstract operation is the same as its concrete counterpart. Due to loop unfolding, we do not require widenings, even though the domain contains infinite ascending chains (see Section 5.2).

Lemma 4. *The trace domain is locally sound.*

Representation for Sets of Event Traces We represent sets of finite event traces corresponding to a particular program location by a directed acyclic graph (DAG) with vertices V, a dedicated root $r \in V$, and a node labeling $\ell \colon V \to \mathcal{P}(\mathcal{E}) \cup \{\sqcup\}$. In this graph, every node $v \in V$ represents a set of traces $\gamma(v) \in \mathcal{P}(\mathcal{E}^*)$ in the following way:

1. For the root r, $\gamma(r) = \{\varepsilon\}$
2. For v with $L(v) = \sqcup$ and predecessors $u_1,\ldots,u_n$, $\gamma(v) = \bigcup_{i=1}^{n} \gamma(u_i)$.
3. For v with $L(v) \neq \sqcup$ and predecessors $u_1,\ldots,u_n$, $\gamma(v) = \{t.u \mid u \in L(v) \wedge t \in \bigcup_{i=1}^{n} \gamma(u_i)\}$

Intuitively, every $v \in V$ represents a set of event traces, namely the sequences of labels of paths from r to v.

In the context of CacheAudit, we need to implement two operations on this data structure, namely (1) the join $\sqcup^{\mathcal{E}^\sharp}$ of two sets of traces and the (2) addition $upd_{\mathcal{E}^\sharp}(\mathcal{S},E)$ of a cache event to a particular set of traces.

For the join of two sets of traces represented by v and w, we add a new vertex u with label $\sqcup$ and add edges from v and w to u.

For the extension of a set of traces represented by a vertex v by a set of cache events E, we first check whether v already has a child w labeled with E. If so, we use w as a representation of the extended set of traces. If not, we add a new vertex u with label E and add an edge (u,v). In this way we make maximal use of sharing and obtain a prefix DAG. The correctness of the representation follows by construction. In CacheAudit, we use hash consing for efficiently building the prefix DAG.

Counting Sets of Traces The following algorithm $count_{tr}$ overapproximates the number of traces that are represented by a given graph.

1. For the root r, $count_{tr}(r) = 1$
2. For v with $L(v) = \sqcup$ and predecessors $u_1, \ldots, u_n$, $count_{tr}(v) = \sum_{i=1}^{n} count_{\tau}(u_i)$
3. For v with $L(v) \neq \sqcup$ and predecessors $u_1, \ldots, u_n$, $count_{tr}(v) = |L(v)| \cdot \sum_{i=1}^{n} count_{tr}(u_i)$

The soundness of this counting, i.e. the fact that $|\gamma(v)| \leq count_{tr}(v)$, follows by construction. Notice that the precision dramatically decreases with larger sets of labels. In our case, labels contain at most three events and the counting is sufficiently precise.

Counting Timing Variations We currently model execution time as a simple abstraction of traces, see Section 3. In particular, timing is computed from a trace over $\mathcal{E} = \{hit, miss, \perp\}$ by multiplying the number of occurrences of each event by the time they consume: t_{hit}, t_{miss}, and $t_{\perp}$, respectively. The following algorithm $count_{time}$ over-approximates the set of timing behaviors that are represented by a given graph.

1. For the root r, $count_{time}(r) = \{0\}$
2. For v with $L(v) = \sqcup$ and predecessors $u_1, \ldots, u_n$, $count_{time}(v) = \bigcup_{i=1}^{n} count_{time}(u_i)$
3. For v with $L(v) \neq \sqcup$ and predecessors $u_1 \ldots, u_n$,

$$count_{time}(v) = \left\{ t_x + t \;\middle|\; x \in L(v) \wedge t \in \bigcup_{i=1}^{n} count_{time}(u_i) \right\}$$

The soundness of $count_{time}$, i.e. the fact that it delivers a superset of the number of possible timing behaviors, follows by construction.

7 Case Studies

In this section we demonstrate the capabilities of CacheAudit in case studies where we use it to analyze the cache side channels of algorithms for sorting and symmetric encryption. All results are based on the automatic analysis of corresponding 32-bit x86 Linux executables that we compiled using *gcc*.

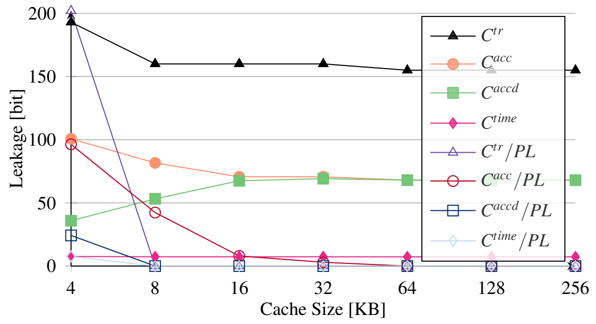

Figure 5: Effect of the attacker model and preloading (PL) on the security guarantee, for varying cache sizes. The results are given for a 4-way set associative cache with a line size of 64B and the LRU replacement strategy.

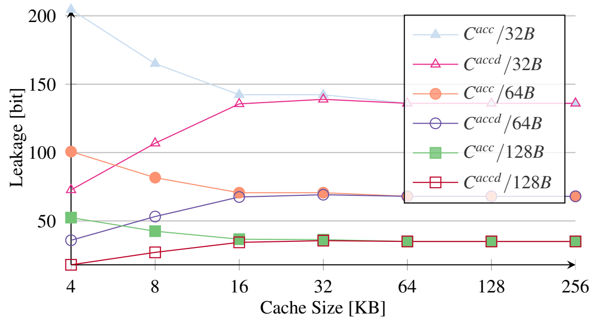

Figure 6: Effect of the cache line size on the security guarantee, for C^{acc} and C^{accd}, for varying cache sizes. The results are given for a 4-way set associative cache with the LRU replacement strategy.

7.1 AES 128

We analyze the AES implementation from the PolarSSL library [3] with keys of 128 bits, where we consider the implementation with and without preloading of tables, for all attacker models, different replacement strategies, associativities, and line sizes. All results are presented as upper bounds of the leakage in *bits*; for their interpretation see Theorem 1. In some cases, CacheAudit reports upper bounds that exceed the key size (128 bits), which corresponds to an imprecision of the static analysis. We opted against truncating to 128 bits to illustrate the degree of imprecision. The full data of our analysis are given in the extended version of this paper [19]. Here, we highlight some of our findings.

- *Preloading* almost consistently leads to better security guarantees in all scenarios (see e.g. Figure 5). However, the effect becomes clearly more apparent for cache sizes beyond 8KB, which is explained by the PolarSSL AES tables exceeding the size of the 4KB cache by 256B. For cache sizes that are larger than the preloaded tables, we can prove noninterference for C^{acc} and FIFO, C^{accd} and LRU, and for C^{tr} and C^{time} on LRU, FIFO, and PLRU. For C^{acc} with shared memory spaces and LRU,

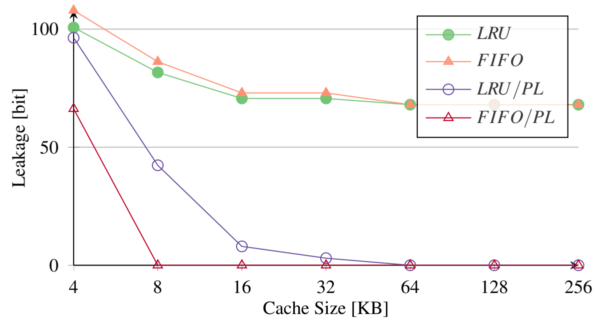

Figure 7: Effect of the replacement strategy on the security guarantee for C^{acc}, with and without preloading (PL), for varying cache sizes. The results are given for a 4-way set associative cache with a line size of 64B.

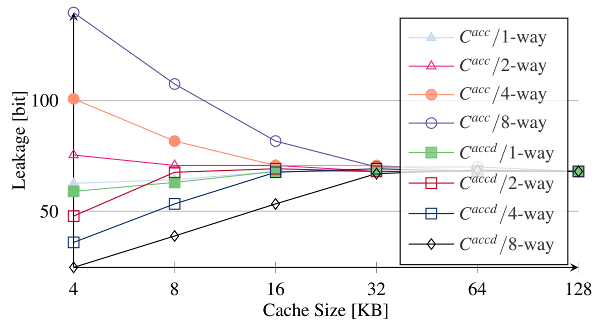

Figure 8: Effect of the associativity on the security guarantee, for C^{acc} and C^{accd}, without preloading, for varying cache sizes. The results are given for a cache with a line size of 64B and the LRU replacement strategy.

this result does *not* hold because the adversary can obtain information about the order of memory blocks in the cache.

• A larger *line size* consistently leads to better security guarantees for access-based adversaries (see e.g. Figure 6). This follows because more array indices map to a line which decreases the resolution of the attacker's observations.

• In terms of *replacement strategies*, we consistently derive the lowest bounds for LRU, followed by PLRU and FIFO (see the extended version [19]), where the only exception is the case of C^{acc} and preloading (see Figure 7). In this case FIFO is more secure because with LRU the adversary can obtain information about the ordering of memory blocks in the cache.

• In terms of *cache size*, we consistently derive better bounds for larger caches, with the exception of C^{accd}. For this adversary model the bounds increase because larger caches correspond to distributing the table to more sets, which increases its possibilities to observe variations. The guarantees we obtain for C^{accd} and C^{acc} converge for caches of 4 ways and sizes beyond 16KB (see e.g. Figure 6). This is due to the fact that each cache set can contain at most one unique block of the 4KB table. In that way, the ability to observe ordering of blocks within a set does not give C^{acc} any advantage.

• When increasing *associativity*, we observe opposing effects on the leakage of C^{acc} and C^{accd} (see Figure 8). This is explained by the fact that, for a fixed cache size, increasing associativity means decreasing the number of sets. For C^{accd} which can only observe the number of blocks that have been loaded into each set, this corresponds to a decrease in observational capability; for C^{acc} which can observe the ordering of blocks, this corresponds to an increase. This difference vanishes for larger cache sizes because then each set contains at most one unique block of the AES tables.

Comparison to [34]: In a recent study [34] we analyzed the PolarSSL AES implementation with respect to access-based adversaries and LRU replacement, using the cache component of a closed-source tool for worst-case execution time analysis [1]. The results we obtain using CacheAudit go beyond that analysis in that we derive bounds w.r.t. access-based, trace-based, and time-based adversaries, for LRU, FIFO, and PLRU strategies. For access-based adversaries and LRU, the bounds we derive are lower than those in [34]; in particular, for C^{accd} we derive bounds of zero for implementations with preloading for *all* caches sizes that are larger than the AES tables—which is obtained in [34] only for caches of 128KB. While these results are obtained for different platforms (x86 vs. ARM) and are hence not directly comparable, they do suggest a significant increase in precision. In contrast to [34], this is achieved without any code instrumentation.

7.2 Salsa20

Salsa20 is a stream cipher by Bernstein [11]. Internally, the cipher uses XOR, addition mod 2^{32}, and constant-distance rotation operations on an internal state of 16 32-bit words. The lack of key-dependent memory lookups intends to avoid cache side channels in software implementations of the cipher. With CacheAudit we could formally confirm this intuition by automated analysis of the reference implementation of Salsa20 encryption, which includes a function call to a hash function. Specifically, we analyze the leakage of the encryption operation on an arbitrary 512-byte message for C^{acc}, C^{tr}, and C^{time}, FIFO and LRU strategies, on 4KB caches with line size of 32B, where we consistently obtain upper bounds of 0 for the leakage. The time required for analyzing each of the cases was below 11s.

7.3 Sorting Algorithms

In this section we use CacheAudit to establish bounds on the cache side channels of different sorting algorithms. This case study is inspired by an early investigation of secure sorting algorithms [8]. While the authors of [8] consider only time-based adversaries and noninterference as a security property, CacheAudit allows us to give quantitative answers for a comprehensive set of side-channel adversaries, based on the binary executables and concrete cache models.

As examples, we use the implementations of BubbleSort, InsertionSort, and SelectionSort from [4], which are given in Section 2 and Appendix A, respectively, where we use integer arrays of lengths from 8 to 64.

The results of our analysis are summarized in Figure 9. In the following we highlight some of our findings.

- We obtain the same bounds for BubbleSort and SelectionSort, which is explained by the similar structure of their control flow. A detailed explanation of those bounds is given in Section 2. InsertionSort has a different control flow structure, which is reflected by our data. In particular InsertionSort has only $n!$ possible execution traces due to the possibility of leaving the inner loop, which leads to better bounds w.r.t. trace-based adversaries. However, InsertionSort leaks more information to timing-based adversaries, because the number of iterations in the inner loop varies and thus fewer executions have the same timing.
- For access-based adversaries we obtain zero bounds for all algorithms. For trace-based adversaries, the derived bounds do not imply meaningful security guarantees: the bounds reported for InsertionSort are in the order of $\log_2(n!)$, which corresponds to the maximum information contained in the ordering of the elements; the bounds reported for the other sorting algorithms exceed this maximum, which is caused by the imprecision of the static analysis.
- We performed an analysis of the sorting algorithms for smaller (256B) and larger (64KB) cache sizes and obtained the exact same bounds as in Figure 9, with the exception of the case of arrays of 64 entries and 256B caches: there the leakage increases because the arrays do not fit entirely into the cache due to their misalignment with the memory blocks.

7.4 Discussion and Outlook

A number of comments are in order when interpreting the bounds delivered by CacheAudit. First, we obtained all of the bounds for an *empty* initial cache. As described in Section 3.6, they immediately extend to bounds for *arbitrary* initial cache states, as long as the victim does not access any block that is contained in it. This is relevant, e.g. for an adversary who can fill the initial cache state only with lines from its own disjoint memory space. For LRU and access-based adversaries, our bounds extend to arbitrary initial cache states without further restriction.

Second, while CacheAudit relies on more accurate models of cache and timing than any information-flow analysis we are aware of, there are several timing-relevant features of hardware it does not capture (and make assertions about) yet, including out-of-order execution, which may reorder memory accesses, TLBs, and multiple levels of caches.

Third, for the case of AES and Salsa20, the derived bounds hold for the leakage about the key in *one* execution, with respect to any payload. For the case of zero leakage (i.e., noninterference), the bounds trivially extend to bounds for multiple executions and imply strong security guarantees. For the case of non-zero leakage, the bounds can add up when repeatedly running the victim process with a fixed key and varying payload, leading to a decrease in security guarantees. One of our prime targets for future work is to derive security guarantees that hold for multiple executions of the victim process. One possibility to achieve this is to employ leakage-resilient cryptosystems [20, 47], where our work can be used to bound the range of the leakage functions.

Finally, note that the bounds delivered by CacheAudit can only be used for certifying that a system is secure; they cannot be used for proving that it is *not*. There are two reasons why the bounds may be overly pessimistic: First, CacheAudit may over-estimate the amount of leaked information due to imprecision of the static analysis. Second, the secret input may not be effectively recoverable from the leaked information by an adversary that is computationally bounded.

8 Related Work

The work most closely related to ours is [34]. There, the authors quantify cache side channels by connecting a commercial, closed-source tool for the static analysis of worst-case execution times [1] to an algorithm for counting concretizations of abstract cache states. The application of the tool to side-channel analysis is limited to access-based adversaries and requires heavy code instrumentation. In contrast, CacheAudit provides tailored abstract domains for all kinds of cache side-channel adversaries, different replacement strategies, and is modular and open for further extensions. Furthermore, the bounds delivered by CacheAudit are significantly tighter than those reported in [34]; see Section 7.

Zhang et al. [48] propose an approach for mitigating timing side channels that is based on contracts betweens software and hardware. The contract is enforced on the software side using a type system, and on the hardware

array length	8			16			32			64		
	C^{tr}	C^{time}	C^{acc}	C^{tr}	C^{time}	C^{acc}	C^{tr}	C^{time}	C^{acc}	C^{tr}	C^{time}	C^{acc}
BubbleSort	28	4.86	0	120	6.92	0	496	8.96	0	2016	11	0
InsertionSort	15.23	6.91	0	44.3	10.15	0	117.7	13.3	0	296	15.8	0
SelectionSort	28	4.86	0	120	6.92	0	496	8.96	0	2016	11	0

Figure 9: The table illustrates the security guarantees derived by CacheAudit for the implementations of BubbleSort, SelectionSort, and InsertionSort, for trace-based, timing-based, and access-based adversaries, for LRU caches of 4KB and line sizes of 32B.

side, e.g., by using dedicated hardware such as partitioned caches. The analysis ensures that an adversary cannot obtain any information by observing public parts of the memory; any confidential information the adversary obtains must be via timing, which is controlled using dedicated mitigate commands. Tiwari et al. [45] sketch a novel microarchitecture that faciliates information-flow tracking by design, where they use noninterference as a baseline confidentiality property. Other mitigation techniques include coding guidelines [15] for thwarting cache attacks on x86 CPUs, or novel cache architectures that are resistant to cache side-channel attacks [46]. The goal of our approach is orthogonal to those approaches in that we focus on the *analysis* of microarchitectural side channels rather than on their mitigation. Our approach does not rely on a specific platform; rather it can be applied to any language and hardware architecture, for which abstractions are in place.

Kim et al. put forward StealthMem [29], a system-level defense against cache-timing attacks in virtualized environments. The core of StealthMem is a software-based mechanism that locks pages of a virtual machine into the cache and prevents their eviction by other VMs. StealthMem can be seen as a lightweight variant of flushing/preloading countermeasures. As future work, we plan to use our tool to derive formal, quantitative guarantees for programs using StealthMem.

For the case of AES, there are efficient software implementations that avoid the use of data caches by bit-slicing [28]. Furthermore, a model for statistical estimation of the effectiveness of AES cache attacks based on sizes of cache lines and lookup tables has been presented in [44]. In contrast, our analysis technique applies to arbitrary programs.

Technically, our work builds on methods from quantitative information-flow analysis (QIF) [14], where the automation by reduction to counting problems appears in [9, 38, 26, 37], the connection to abstract interpretation in [35], and the application to side channel analysis in [33]. Finally, our work goes beyond language-based approaches that consider caching [7, 25] in that we rely on more realistic models of caches and aim for more permissive, quantitative guarantees.

9 Conclusions

We presented CacheAudit, the first automatic tool for the static derivation of formal, quantitative security guarantees against cache side-channel attacks. We demonstrate the usefulness of CacheAudit by establishing the first formal proofs of security of software-based countermeasures for a comprehensive set of adversaries and based on executable code.

The open architecture of CacheAudit makes it an ideal platform for future research on microarchitectural side channels. In particular, we are currently investigating the derivation of security guarantees for concurrent adversaries. Progress along those lines will provide a handle for extending our security guarantees to the operating system level. We will further investigate abstractions for hardware features such as pipelines, out-of-order execution, and leakage-resilient cache designs, with the goal of providing broad tool support for reasoning about side-channels arising at the hardware/software interface.

Acknowledgments We thank Adam Chlipala and the anonymous reviewers for the constructive feedback, and Ignacio Echeverría and Guillermo Guridi for helping with the implementation.

This work was partially funded by European Projects FP7-256980 NESSoS and FP7-229599 AMAROUT, by the Spanish Project TIN2012-39391-C04-01 Strong-Soft, by the Madrid Regional Project S2009TIC-1465 PROMETIDOS, and by the German Research Council (DFG) as part of the Transregional Collaborative Research Center AVACS.

References

[1] AbsInt aiT Worst-Case Execution Time Analyzers. `http://www.absint.com/a3/`.

[2] Intel Advanced Encryption Standard (AES) Instructions Set. `http://software.intel.com/file/24917`.

[3] PolarSSL. `http://polarssl.org/`.

[4] Sorting algorithms. `http://www.codebeach.com/2008/09/sorting-algorithms-in-c.html`.

[5] O. Aciiçmez and Ç. K. Koç. Trace-driven cache attacks on AES. In *ICICS*, pages 112–121. Springer, 2006.

[6] O. Aciiçmez, W. Schindler, and Ç. K. Koç. Cache based remote timing attack on the AES. In *CT-RSA*, pages 271–286. Springer, 2007.

[7] J. Agat. Transforming out timing leaks. In *POPL 2000*, pages 40–53. ACM, 2000.

[8] J. Agat and D. Sands. On confidentiality and algorithms. In *SSP*, pages 64–77. IEEE, 2001.

[9] M. Backes, B. Köpf, and A. Rybalchenko. Automatic discovery and quantification of information leaks. In *SSP*, pages 141–153. IEEE, 2009.

[10] D. Bernstein. Cache-timing attacks on AES. `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`.

[11] D. Bernstein. Salsa20. `http://cr.yp.to/snuffle.html`.

[12] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *FMPA*, pages 128–141. Springer, 1993.

[13] A. Chlipala. Modular development of certified program verifiers with a proof assistant. In *ICFP*, pages 160–171. ACM, 2006.

[14] D. Clark, S. Hunt, and P. Malacaria. A static analysis for quantifying information flow in a simple imperative language. *JCS*, 15(3):321–371, 2007.

[15] B. Coppens, I. Verbauwhede, K. D. Bosschere, and B. D. Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *SSP*, pages 45–60. IEEE, 2009.

[16] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[17] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282, 1979.

[18] P. Cousot, R. Cousot, and L. Mauborgne. Theories, solvers and static analysis by abstract interpretation. *Journal of the ACM*, 59(6):31, 2012.

[19] G. Doychev, D. Feld, B. Köpf, L. Mauborgne, and J. Reineke. CacheAudit: A tool for the static analysis of cache side channels. `http://eprint.iacr.org/2013/253`.

[20] S. Dziembowski and K. Pietrzak. Leakage-resilient cryptography. In *FOCS*, pages 293–302. IEEE, 2008.

[21] C. Ferdinand, F. Martin, R. Wilhelm, and M. Alt. Cache behavior prediction by abstract interpretation. *Science of Computer Programming*, 35(2):163 – 189, 1999.

[22] D. Grund. *Static Cache Analysis for Real-Time Systems – LRU, FIFO, PLRU*. PhD thesis, Saarland University, 2012.

[23] D. Gullasch, E. Bangerter, and S. Krenn. Cache games - bringing access-based cache attacks on AES to practice. In *SSP*, pages 490–505. IEEE, 2011.

[24] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *IEEE Proceedings on Real-Time Systems*, 91(7):1038–1054, 2003.

[25] D. Hedin and D. Sands. Timing aware information flow security for a JavaCard-like bytecode. *ENTCS*, 141(1):163–182, 2005.

[26] J. Heusser and P. Malacaria. Quantifying information leaks in software. In *ACSAC*, pages 261–269. ACM, 2010.

[27] S. Jana and V. Shmatikov. Memento: Learning secrets from process footprints. In *SSP*, pages 143–157. IEEE, 2012.

[28] E. Käsper and P. Schwabe. Faster and timing-attack resistant AES-GCM. In *CHES*, pages 1–17, 2009.

[29] 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.

[30] J. Kinder, F. Zuleger, and H. Veith. An abstract interpretation-based framework for control flow reconstruction from binaries. In *VMCAI*, pages 214–228. Springer, 2009.

[31] P. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *CRYPTO*, pages 104–113. Springer, 1996.

[32] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *CRYPTO*, pages 388–397. Springer, 1999.

[33] B. Köpf and D. Basin. An Information-Theoretic Model for Adaptive Side-Channel Attacks. In *CCS*, pages 286–296. ACM, 2007.

[34] B. Köpf, L. Mauborgne, and M. Ochoa. Automatic quantification of cache side-channels. In *CAV*, pages 564–580. Springer, 2012.

[35] B. Köpf and A. Rybalchenko. Approximation and randomization for quantitative information-flow analysis. In *CSF*, pages 3–14. IEEE, 2010.

[36] L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyz-

ers. In *ESOP*, volume 3444 of *LNCS*, pages 5–20. Springer, 2005.

[37] Z. Meng and G. Smith. Calculating bounds on information leakage using two-bit patterns. In *PLAS*. ACM, 2011.

[38] J. Newsome, S. McCamant, and D. Song. Measuring channel capacity to distinguish undue influence. In *PLAS*, pages 73–85. ACM, 2009.

[39] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of AES. In *CT-RSA*, volume 3860 of *LNCS*, pages 1–20. Springer, 2006.

[40] C. Percival. Cache missing for fun and profit. In *BSDCan*, 2005.

[41] 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*, pages 199–212. ACM, 2009.

[42] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

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

[44] K. Tiri, O. Aciiçmez, M. Neve, and F. Andersen. An analytical model for time-driven cache attacks. In *FSE*, volume 4593 of *LNCS*, pages 399–413. Springer, 2007.

[45] M. Tiwari, J. Oberg, X. Li, J. Valamehr, T. E. Levin, B. Hardekopf, R. Kastner, F. T. Chong, and T. Sherwood. Crafting a usable microkernel, processor, and I/O system with strict and provable information flow security. In *ISCA*, pages 189–200. ACM, 2011.

[46] Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *ISCA*, pages 494–505. ACM, 2007.

[47] Y. Yu, F.-X. Standaert, O. Pereira, and M. Yung. Practical leakage-resilient pseudorandom generators. In *CCS*, pages 141–151. ACM, 2010.

[48] D. Zhang, A. Askarov, and A. C. Myers. Language-based control and mitigation of timing channels. In *PLDI*, pages 99–110. ACM, 2012.

[49] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-VM side channels and their use to extract private keys. In *CCS*. ACM, 2012.

A Example Code

Selection Sort

```
void SelectionSort(int a[], int array_size){
   int i;
   for (i = 0; i < array_size - 1; ++i){
      int j, min, temp;
      min = i;
      for (j = i+1; j < array_size; ++j){
          if (a[j] < a[min])
             min = j;
      }
      temp = a[i];
      a[i] = a[min];
      a[min] = temp;
   }
}
```

Insertion Sort

```
void InsertionSort(int a[], int array_size){
  int i, j, index;
  for (i = 1; i < array_size; ++i){
     index = a[i];
     for (j = i; j > 0 && a[j-1] > index; j--)
          a[j] = a[j-1];
     a[j] = index;
  }
}
```