A

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

Goran Doychev, IMDEA Software Institute
Boris Köpf, IMDEA Software Institute
Laurent Mauborgne, AbsInt GmbH
Jan Reineke, Saarland University

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 amount of information that is revealed.

In case studies we apply CacheAudit to binary executables of algorithms for sorting and encryption, including the AES implementation from the PolarSSL library, and the reference implementations of the finalists of the eSTREAM stream cipher competition. The results we obtain exhibit the influence of cache size, line size, associativity, replacement policy, and coding style on the security of the executables, and include the first formal proofs of security for implementations with countermeasures such as preloading and data-independent memory access patterns.

Categories and Subject Descriptors: D.2.0 [**Software Engineering**]: Protection Mechanisms; D.2.4 [**Software Engineering**]: Software/Program Verification; F.3.1 [**Logics and Meaning of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms: General Terms: Security, Verification

Additional Key Words and Phrases: Side channel attacks, Caches

1. INTRODUCTION

Side-channel attacks recover secret inputs to programs from non-functional characteristics of computations, such as time [Kocher 1996], power [Kocher et al. 1999], or memory consumption [Jana and Shmatikov 2012]. 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. This is demonstrated by three documented classes of side-channel attacks:

This article extends and generalizes the results presented in [Doychev et al. 2013]. In particular, it contains as novel material a unified description of cache update policies and their abstractions, algorithms for counting cache states, and an analysis of the finalists of the eSTREAM stream cipher competition. Furthermore the article is accompanied with a refactored and extended version of CacheAudit, which is available at http://software.imdea.org/cacheaudit.

DOI:http://dx.doi.org/10.1145/0000000.0000000

(1) In *time-based attacks* [Kocher 1996; Bernstein 2005] 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 [Aciiçmez et al. 2007].
(2) In *access-based attacks* [Percival 2005; Osvik et al. 2006; Gullasch et al. 2011] the adversary probes the victim's cache state by timing its own accesses to memory. Access-based attacks require the adversary and the victim to share the same hardware platform, which is common in the cloud and has already been exploited [Ristenpart et al. 2009; Zhang et al. 2012b].
(3) In *trace-based attacks* [Aciiçmez and Ç. K. Koç 2006] 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 to 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 [Gueron 2010], or require modifications to the platform [Wang and Lee 2007] 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 [Gullasch et al. 2011; Zhang et al. 2012a], the client application [Bernstein 2005; Osvik et al. 2006; Coppens et al. 2009], or both [Erlingsson and Abadi 2007; Kim et al. 2012]. Reasoning about these interactions can be tricky and error-prone because it relies on the specifics of the binary code and the microarchitecture.

Our approach. 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 lookup 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.

We demonstrate the scope of CacheAudit in case studies where we analyze the side-channel leakage of implementations of representative algorithms for symmetric encryption and sorting. We highlight the following results:

— For the PolarSSL implementation of AES, CacheAudit confirms that preloading of tables significantly improves the security of the executable: for most adversary models and replacement policies, 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 and PLRU caches, CacheAudit reports small, non-zero bounds. And indeed, with LRU and PLRU (in contrast to, e.g., FIFO), the *ordering* of blocks within a cache set reveals information about the victim's final memory accesses.
— An analysis of the software implementations of the four finalists of the eSTREAM competition [ECRYPT 2012] yields the following results: the stream ciphers without lookup tables (Rabbit and Salsa20) are secure against all kinds of cache attacks. In particular, CacheAudit can formally establish leakage bounds of zero, on the basis of the binary executable of the reference implementations, for all adversary models and replacement policies. For HC-128, which employs dynamically updated tables, CacheAudit can establish leakage bounds of zero for some adversary models, whenever the tables fit entirely into the cache. This is explained by the regularity

of the memory accesses of the dynamic updates, which ensure that the entire table is cached. Indeed, the leakage bounds we obtain strikingly resemble those obtained for AES with preloading. For Sosemanuk, the memory accesses do not exhibit such regularity and, indeed, CacheAudit consistently derives non-zero bounds.

Together, these results show how CacheAudit can help with extracting useful information about the security of the interactions of binary executables with the underlying cache architecture.

Technical contributions. On a technical level, our work builds on the fact that the amount of leaked information corresponds to the cardinality of the set of possible side-channel observations (that is, the size of the range of the side channel). This set can be over-approximated by abstract interpretation, which is a theory of sound approximation of program semantics [Cousot and Cousot 1977], and its cardinality can be determined by counting techniques [Köpf and Rybalchenko 2010].

To realize CacheAudit based on this insight, we propose three novel abstract domains (that is, 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. Moreover, we present counting algorithms that determine the cardinality of the set of observations represented by the abstract states of each of these domains. In particular:

(1) We propose an abstract domain that tracks information about the possible cache states, which are represented in terms of the memory blocks that may reside in the cache. We further propose an algorithm that counts the cache states that are represented by an abstract state. The domain and counting procedure are both parametric in the cache update policy, which is described by a permutation [Abel and Reineke 2013]. In contrast to existing abstract domains used in worst-case execution time analysis [Ferdinand et al. 1999; Grund 2012] and their counting procedures [Köpf et al. 2012], our novel domain provides increased precision and it enables the abstraction of a large class of update policies in a uniform and simple manner.
(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 a set of traces, and to determine its cardinality.
(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 these domains in an abstract interpretation framework that captures the relationship between microarchitectural state and program code. We use this framework to establish 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 rich set of adversary models. 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.

Current scope and future extensions of CacheAudit. The current version of CacheAudit offers support for data, instruction, and mixed caches with FIFO, LRU, and PLRU replacement policies, for programs using a limited subset of 32-bit x86 instruc-

tions and CPU flags. The current version does *not* offer support for multiple levels of caches, multiple CPU cores, speculative execution, out-of-order execution, virtual memory, or code using dynamic jump targets and flush instructions. As a consequence, CacheAudit does not make assertions about attacks such as [Yarom and Falkner 2014], which uses `clflush` to attack last-level caches in multicore CPUs. See Section 9 for a discussion of the implications of basing security analysis on such imperfect models and the challenges associated with extending CacheAudit accordingly.

We make the (OCaml) source code and documentation of CacheAudit available from the project page to facilitate future extension:

http://software.imdea.org/cacheaudit

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 in Section 8 and conclude in Section 10 after discussing challenges for future work in Section 9.

2. ILLUSTRATIVE EXAMPLE

In this section, we illustrate on a simple example program the kind of guarantees CacheAudit can derive. Namely, we consider the implementation of BubbleSort shown in Figure 1, 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  }
```

Fig. 1. An implementation of the BubbleSort algorithm

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 $\{2, \ldots, 64\}$.

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

We begin with a primer on caches, where we also define terminology. We then develop a program semantics that includes cache behavior, and we show how it can be used as a basis for quantifying the amount of information leaked by cache 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 lookup, each block can only be stored in a small number of cache lines. For this purpose, caches are partitioned into equally-sized *cache sets* $\mathcal{S}$. The size of a cache set is called the *associativity* k of the cache. There is a function $set : \mathcal{B} \to \mathcal{S}$ 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 [Grund 2012].

3.2. Programs and Computations

We introduce an abstract notion of programs and computations, which we then refine to capture cache behavior. Namely, 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 $next : Traces \to Traces$ 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 program state consists 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 $upd_{\mathcal{M}} \colon \mathcal{M} \to \mathcal{M}$ that is determined solely by the instruction set semantics. The memory update has effects on the cache that are described by a function $eff_{\mathcal{M}} \colon \mathcal{M} \to \mathcal{E}_{\mathcal{M}}$, which we call *memory effect*. The memory effect is an argument to the *cache update* function $upd_{\mathcal{C}} \colon \mathcal{C} \times \mathcal{E}_{\mathcal{M}} \to \mathcal{C}$. 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., $\mathcal{E}_{\mathcal{M}} = \mathcal{B} \cup \{\bot\}$, where $\bot$ denotes that no memory block is accessed.

We model the cache state as a function that assigns an *age* in $\{0, \dots, k-1, k\}$ to every memory block, where the age determines the order in which blocks are evicted. Here, we require that no two blocks that reside in the same cache set have the same age, and we represent blocks that are *not* cached using age k. Formally:

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

Note that $\mathcal{C}$ includes states that cannot occur under some replacement policies: For example, under LRU and FIFO, a block of age $a \in \{1, \dots, k-1\}$ is always preceded by a block of age $a - 1$.

The cache update works as follows. Upon a cache miss, a block is loaded from main memory into a cache set, where it gets assigned age 0. The ages of the other memory blocks in this cache set are incremented by one. In particular, this means that a block of age $k-1$ is evicted from the cache. Upon a cache hit to a block of age $a \in \{0, \dots, k-1\}$, the ages of the blocks in the same set are updated by applying a permutation $\Pi_a \colon A \to A$, which is determined by the replacement policy. We first give a formalization of the cache update in which the replacement policy is kept parametric, before we define

concrete permutations describing LRU, FIFO, and PLRU replacement in Section 3.4:

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

Each cache update results in a cache hit or a cache miss, which we formally capture in terms of a function *cache effect* $\mathit{eff}_{\mathcal{C}} \colon \mathcal{C} \times \mathcal{B} \to \mathcal{E}$:

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

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

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

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

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

3.4. Replacement Policies defined by Permutations

Upon a cache hit, the different replacement policies update the ages of blocks within a cache set according to different permutations. In the following, we define these permutations for the FIFO, LRU, and PLRU replacement policies.

The FIFO replacement policy does not change the ages of the blocks upon cache hits. Its is thus readily modeled as the identity permutation.

$$\Pi_a^{FIFO}(a') = a'$$

The LRU replacement policy sets the age of an accessed block to 0 upon a cache hit, making sure that always the least-recently used blocks get evicted. Formally, we cast this behavior as

$$\Pi_a^{LRU}(a') = \begin{cases} 0 & : a' = a \\ a'+1 & : a' < a \\ a' & : a' > a \end{cases}$$

The operation of the PLRU replacement policy, which is a cost-efficient approximation to LRU, requires a more detailed explanation. For an associativity which is a power of two (the case considered in this paper), PLRU represents each cache set as a full binary tree storing the blocks at its leaves, and each non-leaf stores a bit which represents an arrow pointing to one of the children. Upon a cache miss, the block to be evicted is determined by following the arrows starting from the root. Upon any cache access (regardless whether it is a hit or a miss), the arrows on the way to the accessed block are flipped. Figure 2 shows an example of two consecutive cache hits in a 4-way cache. This construction ensures that upon consecutive cache misses, all cached blocks will be evicted in an order depending on the current settings of the arrows, which allows casting the effect of cache hits as a permutation of the ages. We formally define

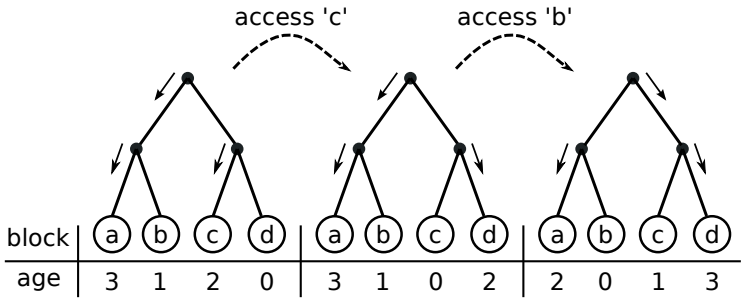


Fig. 2. An example of two consecutive cache hits with PLRU.

this PLRU permutation policy Π^{PLRU} as

$$\Pi_a^{PLRU}(a') = \begin{cases} 0 & : a' = a \\ a' & : a \text{ even} \wedge a' \text{ odd} \\ a'+1 & : a \text{ odd} \wedge a' \text{ even} \\ 2 \cdot \Pi_{\lfloor a/2 \rfloor}^{PLRU}(\lfloor a'/2 \rfloor) & : \text{ otherwise} \end{cases}$$

The intuition behind this formalization is presented in the following. The case distinction in the definition of Π^{PLRU} stems from the observation that in the PLRU binary tree, all blocks stored in the subtree to which the arrow at the root points (the *odd subtree*) have an odd age, and the remaining blocks (in the *even subtree*) have an even age. In the second and the third case in the definition of Π^{PLRU}, we update the age of blocks which are in a different subtree than the accessed block. If the accessed block is from the even subtree (the second case), then the arrow at the root is not flipped, and all the blocks in the odd subtree retain their ages; if the accessed block is from the odd subtree (the third case), the arrow at the root is flipped, which increases the age of all blocks in the even subtree by one. For the blocks in the same subtree as the accessed blocks (the fourth case), the relative order of the ages of those blocks is the same as the order of ages of those blocks if only the subtree is considered as a (twice-smaller) cache set; the new ages are twice the ages in the smaller cache set as only every second evicted block in the actual cache is going to be from this subtree.

3.5. Side Channels

We now define side channels corresponding to access-based, trace-based, and timing-based side-channel adversaries. For the access-based adversaries, we restrict the presentation to *synchronous* adversary models, i.e. those that can control and observe the cache state before and after, but not during, the execution of the victim program. A description of CacheAudit's support for concurrent, *asynchronous* access-based adversaries as in [Gullasch et al. 2011] can be found in [Barthe et al. 2014b].

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.

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 which memory blocks are contained in the cache upon termination of the victim. In practice, this is achieved by probing the cache, which changes the cache state and hence leads to information loss; the assumption that the adversary can determine the cache state is a safe over-approximation of a real adversary. An adversary that does *not* share the memory space also sees the cache state, but cannot distinguish between the different blocks the victim has loaded in each cache set. 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 whether each instruction results in a hit, miss, or does not access memory. The view of a *time-based* adversary is defined by

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

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, thanks to its modular design. We denote the side channels corresponding to the four views by C^{acc}, C^{accd}, C^{tr}, and C^{time}, respectively.

3.6. 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.

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 $C(X)$ (or just C) with $ran(C) \subseteq O$. Here $ran(\cdot)$ denotes the range of the respective random variable. We model the adversary as another random variable $\hat{X}$. The goal of the adversary is to estimate the value of X, i.e. it is successful if $\hat{X} = X$.

Consider first the special case where the adversary does not have access to the side-channel information, but knows the distribution of X. A straightforward upper bound for the probability of correctly guessing the value of X in one shot is given by the probability of the most likely value, where equality can be achieved:

$$P(\hat{X} = X) \leq \max_{\sigma \in I} P(X = \sigma) \; . \tag{1}$$

Consider now the case where the adversary can observe C, and where moreover this is the only information he has about X. We formalize this as the requirement that $X \to C \to \hat{X}$ form a Markov chain, which means that X and $\hat{X}$ do not share information beyond what is contained in C or, more technically, is equivalent to requiring that X and $\hat{X}$ are statistically independent when conditioned on C. The following theorem expresses a security guarantee as an upper bound on the adversary's success probability in terms of the size of the range of C.

THEOREM 3.1. *Let $X \to C \to \hat{X}$ be a Markov chain. Then*

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

PROOF.

$$\begin{aligned}
P(X=\hat{X}) &= \sum_{\sigma,o} P(X=\hat{X}=\sigma \mid C=o)P(C=o) \\
&\overset{(I)}{=} \sum_{o} P(C=o) \sum_{\sigma} P(X=\sigma|C=o)P(\hat{X}=\sigma|C=o) \\
&\leq \sum_{o} P(C=o) \max_{\sigma} P(X=\sigma|C=o) \qquad (2) \\
&\overset{(II)}{=} \sum_{o} \max_{\sigma} P(X=\sigma)P(C=o|X=\sigma) \\
&\overset{(III)}{\leq} \max_{\sigma} P(X=\sigma) \sum_{o} \max_{\sigma} P(C=o|X=\sigma) \qquad (3) \\
&\leq \max_{\sigma} P(X=\sigma)\, |ran(C)|
\end{aligned}$$

where (I) is due to the conditional independence of X and $\hat{X}$ (i.e. the Markov property). Equality (II) follows directly from Bayes' theorem. Inequality (III) is an equality in the case of uniformly distributed X, and the final step follows from the fact that each of the summands is less than or equal to 1. □

A comparison of Equation (1) and Theorem 3.1 shows that the size of the range of C is an upper bound on the factor by which the probability of correct guessing increases when the adversary sees the output of the side-channel $C(X)$ 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 3.1 fundamentally rely on assumptions about the initial distribution of X: if X is easy to guess to begin with, Theorem 3.1 does not imply meaningful security guarantees.

For a formal connection to traditional (entropy-based) presentations of quantitative information-flow analysis, observe that the negative logarithm of (1) is the min-entropy $H(X)$ of X. Likewise the negative logarithm of (2) is the conditional min-entropy $H(X|C)$ of X given C (see [Dodis et al. 2008; Smith 2009] for definitions), i.e., (2) corresponds to $2^{-H(X|C)}$. The logarithm of the factor by which the terms in (1) and Theorem 3.1 differ is a well-known upper bound for $H(X) - H(X|C)$, that is, for the reduction in uncertainty about X when one learns the output of the channel C e.g. [Braun et al. 2009; Köpf and Smith 2010].

3.7. Adversarially Chosen Input

In Section 3.6 we have assumed that the entire initial state is secret. Now we consider the case that initial states are pairs consisting of *high* components that are meant to be kept secret and *low* components that may be provided by the adversary, i.e., $I = I_{hi} \times I_{lo}$. For example, for a decryption algorithm, the high component of the initial state is the key and the low component is the cache state and the ciphertext.

With low inputs, a program and a view define a *family* of channels $C_{\sigma_{lo}} \colon I_{hi} \to O$, one for each low component $\sigma_{lo} \in I_{lo}$. In this case we strive for an upper bound on $|ran(C_{\sigma_{lo}})|$, for *all* $\sigma_{lo} \in I_{lo}$. Such a bound enables us to use Theorem 3.1 to bound the probability of correctly guessing the high component σ_{hi} of the initial state, regardless of the specific choice of σ_{lo}. Note, however, that in multiple program executions with a fixed high input σ_{hi} and different low inputs, information about σ_{hi} may aggregate. A safe upper bound for the range of the corresponding channel is obtained by taking the product of the ranges of the individual channels or, equivalently, by adding the bounds on the number of leaked bits.

We conclude this section by considering the special case in which only the cache state is adversarially chosen, i.e., we consider $I_{lo} = \mathcal{C}$ and $I_{hi} = \mathcal{M}$. We show that the size of the range of the channel corresponding to *one* specific initial cache state can be used as a bound for the size of the range of channels for a larger class of initial cache states. Based on this insight we only have to apply our analysis to the case of that one specific state to obtain sound results for *all* cases of that class. This is particularly useful as our analysis is more precise for known than for unknown initial cache states.

LEMMA 3.2.

(*1*) *For adversaries* $adv \in \{acc, accd\}$*, all permutation-based replacement policies, and all* $c_1, c_2 \in \mathcal{C}$ *such that* c_1 *does not contain empty cache lines: If no program execution accesses blocks in* c_1 *or* c_2*, then* $\left|ran(C_{c_1}^{adv})\right| \geq \left|ran(C_{c_2}^{adv})\right|$.

(*2*) *For adversaries* $adv \in \{time, tr\}$*, all permutation-based replacement policies, and all* $c \in \mathcal{C}$*: If no program execution accesses blocks in* c*, then* $\left|ran(C_{\emptyset}^{adv})\right| = \left|ran(C_{c}^{adv})\right|$.

(*3*) *For adversaries* $adv \in \{acc, accd\}$*, the LRU replacement policy, and all* $c \in \mathcal{C}$*:* $\left|ran(C_{\emptyset}^{adv})\right| \geq \left|ran(C_{c}^{adv})\right|$.

Here, $\emptyset$ *is a shorthand for the empty cache state.*

PROOF. For the proof of (1) and (2) we rely on two properties of permutation-based policies. First, newly inserted blocks have age 0. Second, upon a cache hit, the permutation that is applied to the ages of memory blocks is determined by the age of the requested block. As we assume that the program does not touch any blocks from c_1 or c_2, the ages of the blocks that are loaded during each execution—as well as cache effects—are entirely determined by the sequence of memory accesses of the program. In particular, $ran(C_{c_1}^{adv}) = ran(C_{c_2}^{adv})$ for $adv \in \{time, tr\}$, where c_1 may be empty. For $adv \in \{acc, accd\}$, we define a function that maps a cache in $ran(C_{c_1}^{adv})$ to a cache in $ran(C_{c_2}^{adv})$ by replacing each block b that is also contained in c_1 (i.e., $b = c_1(i)$, for some $i < k$), with the block b' of the same age in c_2 (i.e., $b' = c_2(i)$). This function is well-defined because all lines of c_1 are filled with distinct blocks. The mapping is always surjective, and it is also injective if c_2 does also not contain empty lines. The non-injective case corresponds to the fact that an access-based adversary cannot distinguish between the empty lines in c_2.

For (3) we define a mapping $f_c \colon ran(C_{\emptyset}^{adv}) \to ran(C_{c}^{adv})$ and show that it is surjective. For simplicity, we consider only one cache set, which we view as a sequence of blocks that are indexed by their ages. Then $f_c(d)$ is obtained by appending to the end of d the subsequence of blocks in c that do not appear in d. For showing surjectivity of f_c, pick a state d' in $ran(C_{c}^{adv})$ and consider any sequence of memory accesses that leads to d' from c. When applied to the empty cache state, that sequence leads to a cache state d such that $f_c(d) = d'$. Note that f_c is well-defined because, for LRU, the ordering of the blocks in c that do not appear in d does not depend on the sequence of memory accesses that leads to d (which is, e.g., not true for PLRU). Also note that f_c is in general not injective: A program that either accesses block b or no block at all will produce two possible final cache states when run on an empty initial cache, but only one possible final cache state when run on an initial cache that contains only block b. □

4. AUTOMATIC QUANTIFICATION OF CACHE SIDE CHANNELS

Theorem 3.1 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 interpreta-

tion [Cousot and Cousot 1977] 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 3.5 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 $\mathit{Traces}^\sharp$ together with an abstract transfer function $next^\sharp : \mathit{Traces}^\sharp \to \mathit{Traces}^\sharp$. Here, the elements $a \in \mathit{Traces}^\sharp$ represent subsets of Traces, which is formalized by a concretization function

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

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

$$\forall a \in \mathit{Traces}^\sharp : next\,(\gamma(a)) \subseteq \gamma(next^\sharp(a)) \ . \tag{4}$$

Intuitively, if we maintain a superset of the set of computations during each step of the transfer function as in (4), 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, which is a central result from [Cousot and Cousot 1977]. We use $Col^\sharp$ to denote any such post-fixpoint.

THEOREM 4.1 (LOCAL SOUNDNESS IMPLIES GLOBAL SOUNDNESS). *If local soundness holds, formalized by Equation* (4)*, then*

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

The following theorem is an immediate consequence of Theorem 4.1 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 4.2 (UPPER BOUNDS ON LEAKAGE).

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

With the help of Theorem 3.1, these bounds immediately translate into security guarantees. The relationship of all steps leading to these guarantees is depicted in Figure 3.

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.

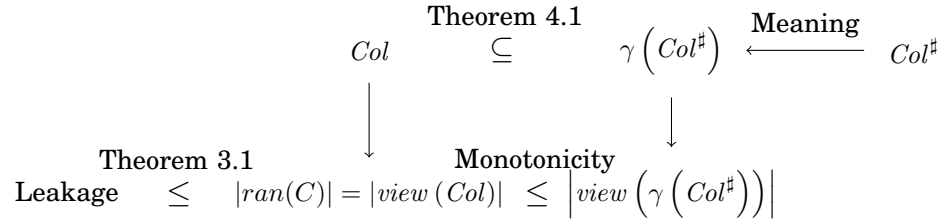


Fig. 3. Relationship of collecting semantics Col, abstract fixpoint $Col^\sharp$, side channels C, and leakage bounds.

However, 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 [Mauborgne and Rival 2005]. 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, respectively, of a location l as follows:

$$succ(l) = \{k \mid next(\gamma_L(l)) \cap \gamma_L(k) \neq \emptyset\}$$
$$pred(l) = \{k \mid next(\gamma_L(k)) \cap \gamma_L(l) \neq \emptyset\}$$

With this we can describe the meaning of an abstract element $a \in Traces^\sharp$ by:

$$\gamma(a) = \{\sigma_0 e_0 \sigma_1 \ldots \sigma_n \in Traces \mid \forall i \leq n, \forall l \in L : \quad \sigma_0 e_0 \sigma_1 \ldots \sigma_i \in \gamma_L(l) \Rightarrow$$
$$\sigma_i^{\mathcal{M}} \in \gamma_{\mathcal{M}}(a^{\mathcal{M}}(l)) \wedge \sigma_i^{\mathcal{C}} \in \gamma_{\mathcal{C}}(a^{\mathcal{C}}(l)) \wedge e_0 \ldots e_{i-1} \in \gamma_{\mathcal{E}}(a^{\mathcal{E}}(l))\} \quad (5)$$

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:

$$next^\sharp(a) = \lambda l.\,(next_{\mathcal{M}^\sharp}(a,l), next_{\mathcal{C}^\sharp}(a,l), next_{\mathcal{E}^\sharp}(a,l)) \ , \quad (6)$$

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 [Cousot et al. 2012].

4.3. Local Soundness

The products and powers of sound abstract domains with partial reductions are again sound abstract domains [Cousot and Cousot 1979]. 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 $eff_{\mathcal{C}^\sharp} : \mathcal{C}^\sharp \times \mathcal{P}(\mathcal{E}_\mathcal{M}) \to \mathcal{P}(\mathcal{E})$, and
— an abstract event $upd_{\mathcal{E}^\sharp} : \mathcal{E}^\sharp \times \mathcal{P}(\mathcal{E}) \to \mathcal{E}^\sharp$.

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.

Now from Equations 4, 5, and 6, we can derive conditions for each domain that are sufficient to guarantee local soundness for the whole analysis:

Definition 4.3 (*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:

$$\begin{aligned} \forall c^\sharp \in \mathcal{C}^\sharp, M \in \mathcal{P}(\mathcal{E}_\mathcal{M}):\ \gamma_\mathcal{C}(upd_{\mathcal{C}^\sharp}(c^\sharp, M)) &\supseteq upd_\mathcal{C}(\gamma_\mathcal{C}(c^\sharp), M), \\ eff_{\mathcal{C}^\sharp}(c^\sharp, M) &\supseteq eff_\mathcal{C}(\gamma_\mathcal{C}(c^\sharp), M), \\ \forall \mathcal{G}^\sharp \subseteq \mathcal{C}^\sharp:\ \gamma_\mathcal{C}\left(\bigsqcup^{\mathcal{C}^\sharp} \mathcal{G}^\sharp\right) &\supseteq \bigcup_{G^\sharp \in \mathcal{G}^\sharp} \gamma_\mathcal{C}\left(G^\sharp\right). \end{aligned}$$

LEMMA 4.4 (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

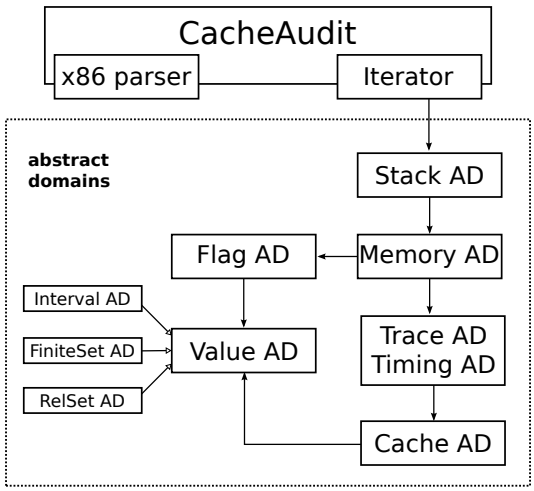


Fig. 4. The architecture of CacheAudit. The solid boxes represent modules. Black-headed arrows mean that the module at the head is an argument of the module at the tail. White-headed arrows represent is-a relationships.

to develop further abstractions in the future, targeting different classes of adversaries, more hardware features, 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 4.5. *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 4.4 with Theorems 4.1 and 4.2, 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 4 depicts the overall architecture of CacheAudit, with the individual modules described below.

The current version of CacheAudit allows analysing a first level cache that is parametric in the cache size, the line size, the associativity, and the replacement policy. We currently support the permutation-based policies LRU, FIFO, and PLRU. We implement a *write-through* cache with *no write-allocate*, i.e., cache writes are directly written to main memory, and when a write-miss occurs, no data is loaded to cache.

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 [Chlipala 2006], which we extend to a set of instructions that is sufficient for our case studies (but not 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 [Kinder et al. 2009] 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 [Cousot and Cousot 1979]. Our analysis uses the *iterative strategy* [Bourdoncle 1993], i.e., it stabilizes components of the abstract control flow graph according to a weak topological ordering, which we compute using Bourdoncle's algorithm.

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. 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 unfoldings we want to perform. Most of our examples (e.g. cryptographic algorithms) require only a small, constant number of loop iterations, and 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 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 domains are used by the cache abstract domain to represent ages of blocks in the cache (in that case, the variables are the ages of blocks), 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 which is described in [Feld 2013].

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 works in conjunction with a value abstract domain to store the state of registers and memory other than flags. It 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, Trace, and Timing Abstract Domains. The cache abstract domain 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 domain tracks which addresses are accessed during computation and returns information about the presence or absence of cache hits and misses to the trace domain. The timings are then obtained as an abstraction from the traces. We describe the details of cache, trace, and timing domains in Section 6 below.

6. ABSTRACT DOMAINS FOR CACHE ADVERSARIES

6.1. Domains for cache states

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. 1999] maintains an upper and a lower bound on the age of every memory block. This abstraction was developed with the 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. To derive tighter leakage bounds, we improve previous work along two dimensions:

(1) Instead of *intervals* of ages, we maintain *sets* of ages of memory blocks.
(2) Upon each cache update, we apply a reduction that eliminates impossible combinations of the ages of the blocks within each cache set.

In addition to increasing precision, these improvements enable us to analyze caches with LRU, FIFO and PLRU replacement in a simple and uniform manner. Interval-based analysis of FIFO and PLRU has been shown to be rather imprecise in the context of worst-case execution time analysis [Heckmann et al. 2003].

Representation and Concretization. In our domain, an abstract cache state $c^\sharp : \mathcal{B} \to \mathcal{P}(A)$ maintains a set of possible ages for each memory block. We update the ages of the blocks belonging to different cache sets independently; in particular, the concretization of an abstract cache state is the Cartesian product of the concretizations of the individual sets.[1] We present a formalization of the case in which the cache has only one set

[1]The conference version of this paper [Doychev et al. 2013] also contains a domain that tracks relational information between the ages of blocks that are cached in different sets. While this relational domain can

that is initially empty, and we informally discuss the extension to multiple cache sets that do not contain empty lines (as required for Lemma 3.2(1)).

At its core, the concretization of a cache set corresponds to the Cartesian product of the ages of the blocks it contains. However, we filter out states that are unreachable in real caches, namely (a) those in which two distinct blocks have the same age, unless that age is k; (b) those with invalid age combinations. For example, for LRU and FIFO replacement, a block of age $a \in \{1, \dots, k-1\}$ is necessarily preceded by a block of age $a-1$, i.e. cache sets never contain "holes". For PLRU replacement, such holes are possible at certain positions, but not at others.

For a given replacement policy, we represent the set of valid age combinations as a subset $V \subseteq \mathcal{P}(A)$. For example, in a 4-way LRU cache, $\{0,1,2,4\} \in V_{\text{LRU}}$ but $\{0,1,3,4\} \notin V_{\text{LRU}}$ because the missing age 2 constitutes an impossible hole. We use the following algorithm to compute the set V of valid age combinations for a replacement policy defined by permutations Π_i:

(1) $V \coloneqq \{\emptyset\}$; $R \coloneqq \emptyset$;
(2) Choose $S \in V \setminus R$; $R \coloneqq R \cup \{S\}$
    (a) Simulate a cache miss by incrementing ages in S and adding 0:
        $S_M \coloneqq \{\min(a+1, k) \mid a \in S\} \cup \{0\}$
    (b) Simulate cache hits to blocks of ages $i \in S$:
        $S_i \coloneqq \{\Pi_i(a) \mid a \in S\}$
(3) $V \coloneqq V \cup \{S_M\} \cup \bigcup_{i \in S} \{S_i\}$
(4) If $V \setminus R \neq \emptyset$ then go to step (2);
(5) Return V;

Technically, the algorithm computes the fixpoint that is reached when updating the empty state with arbitrary sequences of hits and misses. It follows by construction that the final set V contains all possible age combinations for the replacement policy represented by the permutations Π.

With this, we define the concretization $\gamma_{\mathcal{C}}(c^\sharp)$ of an abstract cache (with one cache set) as the Cartesian product of the sets of ages of memory blocks, from which we remove states that map different blocks to the same age and states whose age combinations are not represented in V:

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

So far we have assumed that the cache is initially empty. For *non-empty* initial caches, the holes in valid age combinations will contain blocks from the initial state, which can be distinguished. To account for this, we augment V to represent the identities of those blocks and extend the fixpoint computation accordingly. As there are only k possible blocks in the initial state, the fixpoint can still be effectively computed. The concretization $\gamma_{\mathcal{C}}(c^\sharp)$ then ensures that every concrete cache is an extension of a cache in that fixpoint.

Abstract Cache Update. We implement two algorithms for abstract cache updates, where each offers a different trade-off between precision and performance. First, we implement an abstract transformer that updates the possible ages of each memory block considering only the possible ages of the accessed block, but without considering ages other blocks in the same cache set. This corresponds to a *direct product* [Cousot and Cousot 1979]. For LRU and FIFO we gain precision by additionally performing a

offer a benefit in some cases (see [Feld 2013]), it is rather complex and does not provide significantly better bounds on the examples studied in this paper, which is why we chose not to present it here.

reduction after each cache update, which makes sure that no impossible states with holes are represented. The reduction works by restricting the maximal age of blocks in a cache set to the number of currently cached blocks. The following example shows imprecisions we avoid when using the reduction, and points out additional imprecisions which motivate the use of the second algorithm we use for abstract update. It is based on actual allocations we encountered when analyzing AES (see Section 7).

Example 6.1. Let a, b, c, d be blocks which fall into the same set of a 4-way LRU cache, and e be a block which falls into a different cache set. We write $x \in \{n_1, n_2, n_3\}$ if block x has possible ages n_1, n_2, n_3. At a point of the program execution, the analysis has reached one of the following states:

(i) $a \in \{0, 1\}$; $b \in \{0, 4\}$; $c, d \in \{4\}$
(ii) $a \in \{0, 1\}$; $b \in \{0, 4\}$; $c \in \{1, 2\}$; $d \in \{4\}$
(iii) $a \in \{0, 1, 2\}$; $b \in \{0, 4\}$; $c \in \{2, 3\}$; $d \in \{0, 1, 2\}$

In state (i), if ages are updated without a reduction, then an access to b will result in $a \in \{1, 2\}$. This would mean that a possible allocation is $(b = 0, a = 2)$, however it cannot occur with LRU because it has a hole, as there is no element with age 1. If we perform the reduction described above, as only two elements may be cached, we will only allow ages 0 or 1, which solves the problem. This reduction, however, cannot solve the following problems. Consider a program reaching the state (ii), where we know that either b or e is accessed, after performing reduction, the updated states will be $a \in \{0, 1, 2\}$; $b \in \{0, 4\}$; $c \in \{1, 2\}$. Disallowing c to grow to 3 eliminated a possible hole, however here we obtain a possible allocation $(b = 0, c = 1, a = 2)$, which cannot be reached from state (ii), because it would be only reachable from the impossible allocation $(c = 0, a = 1)$. In state (iii), if we access b or e as in state (ii), the resulting reduced state will be $a \in \{0, 1, 2, 3\}$; $b \in \{0, 4\}$; $c \in \{2, 3, 4\}$; $d \in \{0, 1, 2, 3\}$. Now an additional problem can be observed: the allocation $c = 4$ is possible, i.e., c can be outside of the cache; however when starting at state (iii), we know that c must remain in the cache. This imprecision can then propagate, because if c is accessed at a later point, instead of a sure hit, we will record a hit or a miss.

To avoid the above-mentioned problems, we additionally implement an abstract transformer that concretizes the abstract cache set, updates each concrete state, and then abstracts again. This corresponds to the explicit computation of the *best abstract transformer* [Cousot and Cousot 1979]. We use it in cases where its computation is feasible, which was the case with most experiments from our case study. The soundness of the best abstract transformer follows by construction. The direct product is sound because (a) the transformer assumes no knowledge on other blocks' ages and thus excludes fewer states than the best abstract transformer, and (b) the reduction removes only impossible configurations which are also removed when concretizing.

The cache join and abstract cache effect are implemented in a straightforward fashion. Two cache states are joined by a set union of the possible ages of all blocks. The abstract cache effect is a union of the effects of all possibly accessed blocks, for all possible ages. The soundness of these operations follows directly.

LEMMA 6.2. *The cache domains are locally sound.*

Counting Cache States. We describe the counting of observations for abstract cache states with one cache set; for cache states with more than one cache set, we compute the product of the number of concretizations of the individual sets.

For counting the observations a shared-memory access-based adversary C^{acc} can make, we simply enumerate the concretizations $\gamma_{\mathcal{C}}(c^\sharp)$ and count their number. For a disjoint-memory access-based adversary C^{accd}, we also enumerate concretizations, but we take into account that a disjoint-memory adversary cannot distinguish between

different blocks that have been loaded during execution. That is, we only need to track the number of elements of the fixpoint V that we observe during enumeration of $\gamma_{\mathcal{C}}(c^\sharp)$.

While each counting procedure takes exponential time in the associativity of the cache, this is not a bottleneck in practice, where associativities 2,4,8 are common.

6.2. A Domain for Traces

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

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

LEMMA 6.3. *The trace domain is locally sound.*

Efficient Implementation of the Event Trace Domain. We represent sets of finite event traces corresponding to a 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, $\gamma(r) = \{\epsilon\}$
(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 $u_1, \ldots, u_n$,

$$\gamma(v) = \left\{ t.u \mid u \in L(v) \wedge t \in \bigcup_{i=1}^{n} \gamma(u_i) \right\}$$

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 (a) the join $\sqcup^{\mathcal{E}^\sharp}$ of two sets of traces and the (b) addition $upd_{\mathcal{E}^\sharp}(\mathcal{S}, E)$ of an 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 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 this counting procedure is precise if the labels represent singleton events (because then every trace is uniquely represented in the graph), but that its precision decreases dramatically with larger sets of labels. In our case, labels contain at most three events and the counting is sufficiently precise.

6.3. A Domain for Time

We currently model execution time as a simple abstraction of traces, see Section 3. In particular, timing is computed from a trace over $\mathcal{E} = \{hit, miss, \bot\}$ by multiplying the number of occurrences of each event by the time they consume: t_{hit}, t_{miss}, and $t_\bot$, 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 set of possible timing behaviors, follows by construction.

7. CASE STUDIES

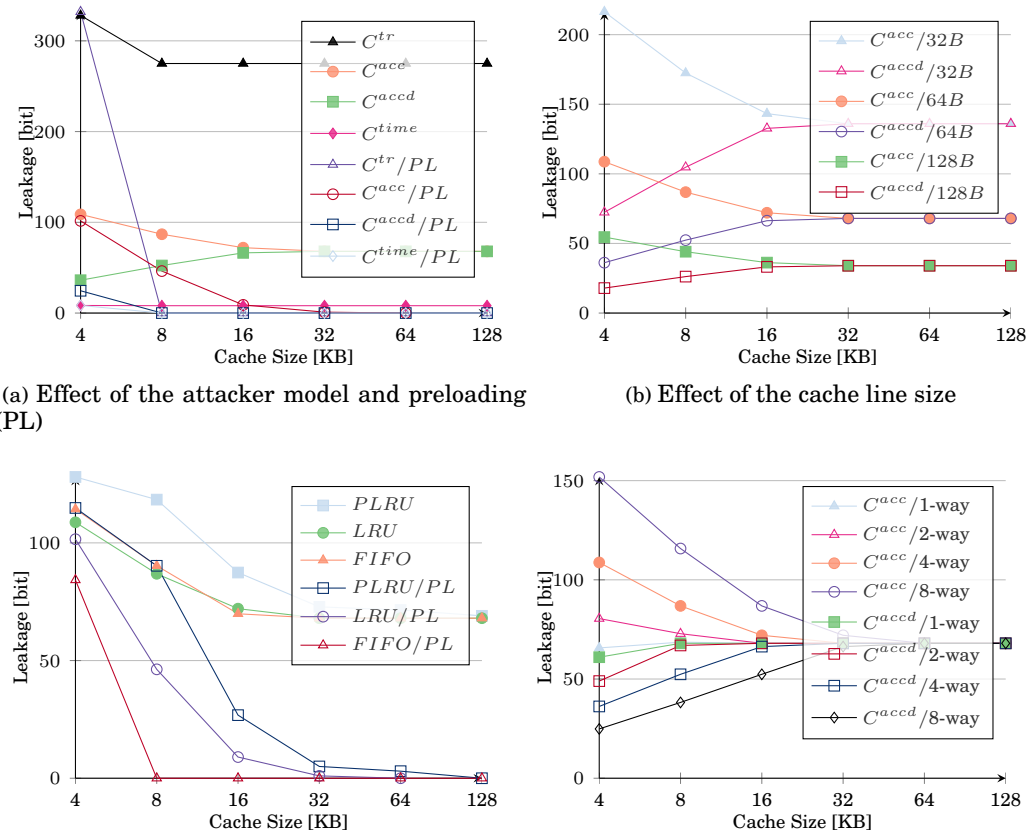
In this section we demonstrate the capabilities of CacheAudit in case studies where we use it to analyze the cache side channels of implementations of algorithms for symmetric encryption and sorting. All results are based on the automatic analysis of corresponding 32-bit x86 Linux executables that we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

7.1. AES

We analyze the AES implementation from the PolarSSL 1.3.7 library with keys of $n \in \{128, 192, 256\}$ bits, where we consider the implementation with and without preloading of lookup tables. We analyze with respect to the $C^{acc}, C^{accd}, C^{tr}, C^{time}$ side models, with the LRU, FIFO, and PLRU replacement policies, for a set of cache sizes, associativities, and line sizes. All results are presented as upper bounds of the leakage in bits; for their interpretation see Theorem 3.1. In some cases, CacheAudit reports upper bounds that exceed the key size n, which corresponds to an imprecision of the static analysis. We opted against truncating to n bits to illustrate the degree of imprecision. We highlight some of our findings.

In the following, we first present results for 256-bit keys before we discuss the effect of varying the key size. The base case we consider is an LRU cache with associativity 4 and line size of 64B. We also explore the effect of varying each of these parameters.

Preloading. Preloading the lookup tables almost consistently leads to better security guarantees in all scenarios (see e.g. Figure 5a). 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 LRU and PLRU, this result does *not* hold because the adversary can obtain information about the order of memory blocks in the cache.



(a) Effect of the attacker model and preloading (PL)

(b) Effect of the cache line size

(c) Effect of the replacement policy

(d) Effect of the associativity

Fig. 5. Security guarantees for PolarSSL's AES implementation with 256-bit keys. The base case considers a 4-way set associative cache with a line size of 64B, and the LRU replacement policy, and varying cache sizes.

Line size. A larger line size consistently leads to better security guarantees for access-based adversaries (see e.g. Figure 5b). This follows because more array indices map to a line which decreases the resolution of the attacker's observations.

Replacement policy. In terms of replacement policies, for C^{accd}, C^{tr}, and C^{time} we consistently derive the lowest bounds for LRU. For C^{acc} and preloading, FIFO exhibits the lowest leakage, with significantly lower bounds than the other policies, as shown in Figure 5c. The reason for this is that with LRU and PLRU (but not with FIFO), consecutive cache hits can lead to reordering of the cached elements, and thus the access-based adversary can obtain information about the ordering of memory blocks in the cache.

Associativity. When increasing associativity, we observe opposing effects on the leakage of C^{acc} and C^{accd} (see Figure 5d). 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.

Cache size. In terms of *cache size*, we consistently derive lower bounds for larger caches, with the exception of C^{accd}. For C^{accd}, 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 5b). 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.

Key size. The choice of key size increases the leakage significantly for C^{tr} and C^{time}, and leads to small variations for C^{acc}, as exemplified in Figure 6. The increase in leakage for C^{tr} and C^{time} can be explained by the longer computations for bigger keys: A key size of 128, 192, or 256 bits results in 10, 12, or 14 rounds of transforming the input, respectively. For bigger keys, as more rounds are performed, there are more accesses to the lookup tables, and each access that cannot be statically predicted as a cache hit or miss doubles the number of possible traces.

	8KB cache				16KB cache			
AES key size	C^{tr}	C^{time}	C^{acc}	C^{accd}	C^{tr}	C^{time}	C^{acc}	C^{accd}
128 bit	199	7.64	89.09	52.79	199	7.64	72.85	66.93
192 bit	223	7.81	87.79	52.79	223	7.81	72.85	66.93
256 bit	279	8.13	90.07	52.79	279	8.13	73.44	66.93

Fig. 6. Leakage in bits with AES when varying the key size, for a configuration with an 8KB and 16KB 4-way cache, with line size of 64B, with the LRU replacement policy.

The variations for C^{acc} are more subtle and have to do with two contradictory effects. The first effect results from the key expansion process, during which the key is expanded to round keys of 128 bits per round. For an n-bit key, the round keys are computed in a loop which generates n bits per iteration. Thus, for smaller keys this loop requires *more* iterations: 10 for 128-bit keys, 8 for 192-bit keys, and 7 for 256-bit keys. The leakage differs because within the i-th iteration, an integer round-constant `rcon[i]` is read from an array in memory, from which more values are read if the key is small, thus more blocks may compete with the lookup tables for the same cache sets. This explains why in Figure 6, for the 8KB cache, there is less C^{acc}-leakage with 192-bit keys than with 128-bit keys.

The second effect results from the size of the expanded key: when a smaller encryption key is being used, there are *less* round keys, and less blocks corresponding to the round keys compete with the lookup tables for their position in the cache sets. Thus, the end-state can contain less possible ages for those blocks, corresponding to less leakage. This explains the increased C^{acc}-leakage for 256-bit keys in Figure 6.

For C^{accd}, no difference is observed between key sizes for the analyzed cache configurations, as the above-described effects affect the ordering of blocks in cache, but not whether those blocks are in cache or not.

7.2. The eSTREAM Portfolio

The goal of the eSTREAM project[ECRYPT 2012] was to foster the creation of novel practical stream ciphers. The project concluded in 2008 with the announcement of the final eSTREAM portfolio, which consists of four ciphers that are particularly suited for implementation in software (Profile 1) and of three ciphers that are suited for implementation in hardware (Profile 2).

We applied CacheAudit to analyse the reference implementations of the ciphers from the eSTREAM Profile 1 portfolio, namely HC-128, Rabbit, Salsa20, and Sosemanuk. We tested the versions of the algorithms with 128-bit keys for the encryption of a 512-byte message, for a 4-way cache with a line size of 64B. The results for LRU are summarized in Figure 7a for C^{acc}, Figure 7b for C^{accd}, Figure 7c for C^{tr}, Figure 7d for C^{time}, and are briefly discussed in the following paragraphs. The effects on which we elaborate are observable with all replacement policies, and we restrict the presentation to LRU for brevity.

7.2.1. HC-128. HC-128 is a stream cipher by [Wu 2004] that relies on a 128-bit key, a 128-bit initialization vector, and an internal state of 4KB, which is stored in two S-boxes of 512 entries of 32-bit values each. During keystream generation, new S-box values are generated every 512 steps.

Cache attacks against the HC series of ciphers have been demonstrated by [Zenner 2009] and [Paul and Raizada 2012], for a different (but similar) adversary model. For small caches, CacheAudit confirms this, and for all considered adversary models we obtain a non-zero leakage. When increasing the size of the cache, CacheAudit shows that the leakage disappears; varying the cache size was not considered by the mentioned attacks. The effect on HC-128 leakage when varying the cache size is similar to AES leakage with preloading (see Section 7.1 and compare with Figure 5a and Figure 5c). The reason for this is that (a) AES and HC-128 rely on lookup tables of similar sizes; (b) dynamically generating HC-128 S-boxes makes sure that they are freshly loaded in cache as a whole, similarly to preloading AES lookup tables. Non-zero leakage is observed when the cache is small, as some of the memory blocks containing S-box values are evicted from cache or other memory competes with them for the same cache sets. For tested configurations with a bigger cache, we obtain zero-leakage.

7.2.2. Rabbit. Rabbit is a stream cipher by [Boesgaard et al. 2005] that relies on a 128-bit key and a 64-bit initialization vector. A set of eight 32-bit state registers and eight 32-bit counters is used to perform encryption based on basic arithmetic and bit-operations. The lack of key-dependent memory lookups intends to avoid any leakage to the cache. This is reflected by the results we obtained with CacheAudit: for all adversary models and all tested cache configurations, we obtain zero-leakage.

It should be noted that [Bernstein 2015a] observes the possibility of a timing leak due to operand-dependent timing of integer multiplication on platforms such as the Motorola PowerPC G4e 7450. The current version of CacheAudit does not support operand-dependent timing and hence does not detect this kind of leak.

7.2.3. Salsa20. Salsa20 is a stream cipher by [Bernstein 2015b]. 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 any leakage to the cache. With CacheAudit we could formally confirm this intuition, and we consistently obtain upper bounds of 0 for the leakage.

7.2.4. Sosemanuk. Sosemanuk is a stream cipher by [Berbain et al. 2005] that relies on keys of length ranging from 128 to 256 bits and an initialization vector of 128 bits. Sosemanuk uses a 10-word linear feedback shift register, a finite-state machine, and an output function for combining both of their outputs into the keystream. The reference implementation we analyze relies on static tables of 4 KB for fast implementation of the feedback register, which have been shown to be susceptible to cache attacks [Leander et al. 2009]. The analysis using CacheAudit confirms this weakness. In particular, we obtain non-zero leakage for all tested configurations and adversary models, with higher leakage observed for smaller cache. In this respect, the results resemble those for AES without preloading (see Section 7.1).

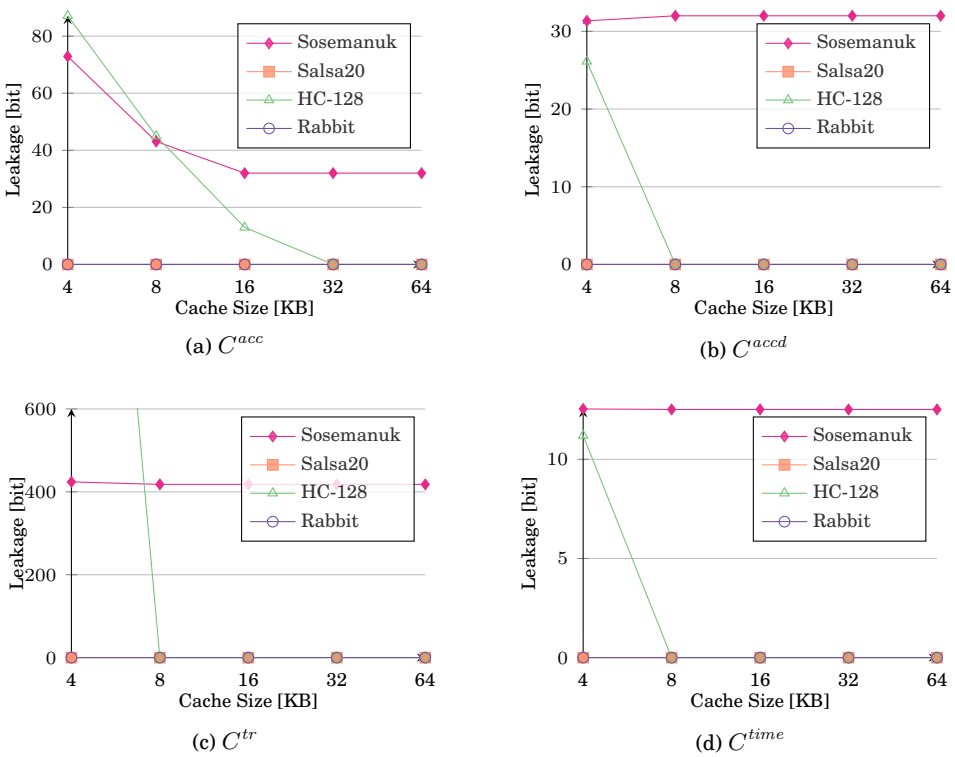


Fig. 7. Security guarantees of eSTREAM finalists for $C^{acc}, C^{accd}, C^{tr}, C^{time}$, for varying cache sizes. The results are given for a 4-way set associative cache with a line size of 64B and the LRU replacement policy.

Summary. With CacheAudit, we can prove zero-leakage for Rabbit and Salsa20 for all tested configurations. For HC-128, we prove zero-leakage only for bigger cache sizes. For Sosemanuk we obtain non-zero leakage bounds for all tested configurations; for small caches, Sosemanuk's leakage bounds are lower than the bounds for HC-128.

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 [Agat and Sands 2001]. While the authors of [Agat and Sands 2001] 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 implementations from [Code Beach 2008] of the sorting algorithms BubbleSort (see Figure 1), InsertionSort, and SelectionSort. We use integer arrays of lengths between 8 and 64.

The results of our analysis are summarized in Figure 8. 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

len	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}
BS	28	4.9	0	120	6.9	0	496	9	0	2016	11	0
IS	15.3	6.9	0	44.3	10.1	0	118	12.5	0	296	14.6	0
SS	28	4.9	0	120	6.9	0	496	9	0	2016	11	0

Fig. 8. The table illustrates the security guarantees derived by CacheAudit for the implementations of BubbleSort (BS), InsertionSort (IS), and SelectionSort (SS), for trace-based, timing-based, and access-based adversaries, for LRU caches of 4KB and line sizes of 32B, for array length (len) between 8 and 64.

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 8, 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

A number of comments are in order when interpreting the bounds delivered by CacheAudit.

Meaning of Bounds. The quantities computed by CacheAudit are upper bounds on the leaked information that are not necessarily tight, that is, they may be pessimistic. There are two reasons why the bounds may be pessimistic: First, CacheAudit may overestimate 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.

The fact that CacheAudit delivers upper bounds has two consequences. First, the results can only be used for certifying that a system is secure; they cannot be used for proving that it is *not*. Second, the natural ordering on bounds cannot be directly used for comparing the real-world security of systems. For example, “at most two bits leak” is a correct (but pessimistic) bound for a system that does not leak any information, and “at most one bit leaks” is a correct (and tight) bound for a system that leaks one bit. The first bound is lower than the second, even though the first system is more secure than the second.

Instead, lower bounds represent a better state of affairs in systematic reasoning about the security of a system, which is a desirable goal for implementors of (cryptographic) algorithms and side-channel countermeasures.

Use of Imperfect Models. The guarantees delivered by CacheAudit are only valid to the extent to which the models used accurately capture the relevant aspects of the execution platform. A recent empirical study of OS-level side channels on different platforms [Cock et al. 2014] shows that advanced microarchitectural features may interfere with the cache, which can render countermeasures ineffective — and formal

guarantees invalid. See Section 9 for a discussion of the challenges associated with extending CacheAudit with such advanced features.

Multiple Executions. For the case of the cryptosystems we analyzed, the 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 [Dziembowski and Pietrzak 2008; Yu et al. 2010], where our work can be used to bound the range of the leakage functions, as demonstrated in [Barthe et al. 2014b].

Initial Cache State. We obtained all bounds in our experiments for initial states that do not contain blocks that are accessed by the program. As described in Section 3.7, they immediately extend to bounds for initial cache states containing empty lines. 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, as justified by Lemma 3.2(3).

8. RELATED WORK

Existing work on mitigation techniques for cache side channels can be classified as hardware-based, OS-based, code-based, or mixed:

— Hardware-based techniques include [Tiwari et al. 2011], who present a novel microarchitecture that facilitates information-flow tracking by design, where they use noninterference as a baseline confidentiality property. [Domnitser et al. 2012] propose non-monopolizable caches, which is a hardware defense against access-based attacks that puts a bound on the number of lines in each cache set that can be used by a process. Depending on the degree of "non-monopolization", an adversary cannot evict any or only some of the victim's data from the cache, which eliminates or at least weakens access-based attacks. [Wang and Lee 2008] propose novel cache architectures that achieve attractive trade-offs between security and performance. In particular, they rely on randomized cache replacement policies that are designed to achieve security.

— OS-based techniques include StealthMem [Kim et al. 2012], 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. A formalization of StealthMem is provided in [Barthe et al. 2014a]. [Baig et al. 2014] propose CloudFlow, which is a cloud-wide information-control layer based on OpenStack, which relies on a novel, fast VM introspection mechanism. [Raj et al. 2009] propose system-level defenses for isolating machines in software-as-a-service environments, such as cache-aware CPU core assignment and cache-aware memory-management. [Ford 2012] proposes information-flow control based on explicit timing labels, together with operating system support for its enforcement.

— Code-based techniques include the program counter security model [Molnar et al. 2006], which is to assume that an adversary can observe the value of the program counter at every step. The authors also propose a program transformation that achieves security in this model. Security implies resistance against control-flow based timing attacks, but does not account for leaks through secret-dependent mem-

ory lookups. [Agat 2000] propose a code transformation for Java Bytecode to eliminate control-flow based attacks in Java Bytecode, together with proofs of soundness. [Hedin and Sands 2005] extend this timing model with execution histories, offering a hook for reasoning about cache state. [Käsper and Schwabe 2009] propose bitslicing to avoid the use of data caches and show that this leads to efficient software implementations of AES. Finally, [Coppens et al. 2009] discuss practical coding techniques for mitigating cache attacks on x86 CPUs.

— Mixed techniques include [Zhang et al. 2012a], who 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 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 that reduce the number of possible timing observations.

The goal of our work is orthogonal to those approaches in that we focus entirely 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.

Technically, our work builds on methods from quantitative information-flow analysis (QIF) [Clark et al. 2007], where the automation by reduction to counting problems appears in [Backes et al. 2009; Newsome et al. 2009; Heusser and Malacaria 2010; Meng and Smith 2011], the connection to abstract interpretation in [Köpf and Rybalchenko 2010], and the application to side channel analysis in [Köpf and Basin 2007].

The development of CacheAudit is inspired by a feasibility study [Köpf et al. 2012], where we quantify cache side channels by connecting a commercial, closed-source tool for the static analysis of worst-case execution times [AbsInt Angewandte Informatik GmbH 2015] 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 policies, and is modular and open for further extensions. Furthermore, the bounds delivered by CacheAudit are significantly tighter than those reported in [Köpf et al. 2012]: For access-based adversaries and LRU, the bounds we derive are lower than those in [Köpf et al. 2012]; 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 [Köpf et al. 2012] 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 [Köpf et al. 2012], this is achieved without any code instrumentation.

9. CHALLENGES FOR FUTURE WORK

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 microarchitectures that it does not yet capture (and make assertions about), including *second and third level caches*; *shared caches* in multi cores; *DRAM*, commonly used as main memory, which—just like caches—exhibits varying access latencies depending on the history of memory accesses; *speculation*, which may introduce memory accesses that are not part of the "logical" execution of the program; *out-of-order execution*, which may reorder memory accesses; *translation lookaside buffers (TLBs)* and other mechanisms related to the implementation of *virtual memory*.

There are two immediate challenges regarding the features mentioned above:

(1) To obtain detailed models that faithfully capture the behavior of these features in modern microarchitectures.
(2) To devise abstractions of these models that enable precise, yet efficient analysis.

Challenge (1) is daunting, as modern microarchitectures are extremely complex and at the same time poorly documented, at least when it comes to documentation that is publicly available. One promising approach to deal with this challenge is to apply techniques from machine learning to reverse engineer microarchitectural models, as recently demonstrated in [Abel and Reineke 2013]. Such approaches will, however, never be able to provide absolute certainty about the correctness of the models.

Challenge (2) is equally daunting. The worst-case execution time (WCET) community has gathered experience in the design of analyses for some of the features mentioned above, and it has been observed that speculation and out-of-order execution dramatically increase analysis complexity. Current consumer microarchitectures are at least an order of magnitude more complex than the most advanced microarchitectures used in safety-critical systems for which WCET analyses have been devised. Thus, breakthroughs in analysis technology will be required to solve Challenge (2).

Maybe a more viable approach than to attack Challenges (1) and (2) by devising ever more complex models and analyses, is to actually make *stronger abstractions* that are based upon *fewer* assumptions about the microarchitecture. This would have two beneficial effects:

— An increase in confidence in the analysis results, as they would be based on fewer assumptions that may or may not hold in reality.
— The fewer assumptions are made, the more microarchitectures satisfy the assumptions. Security statements could possibly be made for large classes of architectures.

As an example, in the context of cache side channels one could base the side-channel analysis on a lower bound on the cache capacity and a lower bound on the number of cache sets, rather than the cache's exact geometry. The challenge is to reduce assumptions without sacrificing precision.

In a similar spirit, side-channel analysis may be performed at a higher level of abstraction. Proposals such as StealthMem [Kim et al. 2012] introduce a security layer that enables the construction of secure programs without having to know about microarchitectural details. Can we characterize the guarantees that such *security APIs* provide and analyze applications based on these guarantees? Also, can we analyze the implementations of these *security APIs* to prove that they deliver the promised guarantees?

10. 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 formal security guarantees for binary executables of sorting algorithms and state-of-the-art cryptosystems.

Acknowledgments. We thank Adam Chlipala and the anonymous reviewers of ACM TISSEC and the USENIX Security Symposium for their constructive feedback, and Ignacio Echeverría, Dominik Feld, and Guillermo Guridi for helping with the implementation.

This work was partially funded by Spanish Project TIN2012-39391-C04-01 StrongSoft, Madrid Regional Project S2013/ICE-2731 N-GREENS, and by the German Research Council (DFG) as part of the Transregional Collaborative Research Center AVACS.

REFERENCES

Andreas Abel and Jan Reineke. 2013. Measurement-based modeling of the cache replacement policy. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*. IEEE, 65–74.

AbsInt Angewandte Informatik GmbH. 2015. AbsInt aiT Worst-Case Execution Time Analyzers. http://www.absint.com/ait/. (Accessed: 7 January 2015).

Onur Aciiçmez and Ç. K. Koç. 2006. Trace-Driven Cache Attacks on AES. In *ICICS*. Springer, 112–121.

Onur Aciiçmez, Werner Schindler, and Ç. K. Koç. 2007. Cache Based Remote Timing Attack on the AES. In *CT-RSA*. Springer, 271–286.

Johan Agat. 2000. Transforming out Timing Leaks. In *POPL 2000*. ACM, 40–53.

Johan Agat and David Sands. 2001. On Confidentiality and Algorithms. In *SSP*. IEEE, 64–77.

Michael Backes, Boris Köpf, and Andrey Rybalchenko. 2009. Automatic Discovery and Quantification of Information Leaks. In *SSP*. IEEE, 141–153.

Mirza Basim Baig, Connor Fitzsimons, Suryanarayanan Balasubramanian, Radu Sion, and Donald E Porter. 2014. CloudFlow: Cloud-wide policy enforcement using fast VM introspection. In *IC2E*. IEEE.

Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Carlos Luna, and David Pichardie. 2014a. System-level non-interference for constant-time cryptography. Cryptology ePrint Archive, Report 2014/422. (2014).

Gilles Barthe, Boris Köpf, Laurent Mauborgne, and Martín Ochoa. 2014b. Leakage Resilience against Concurrent Cache Attacks. In *Proc. 3rd Conference on Principles of Security and Trust (POST'14)*. Springer.

Come Berbain, Olivier Billet, Anne Canteaut, Nicolas Courtois, Henri Gilbert, Louis Goubin, Aline Gouget, Louis Granboulan, Cdric Lauradoux, Marine Minier, Thomas Pornin, and Herv Sibert. 2005. Sosemanuk, a fast software-oriented stream cipher. http://www.ecrypt.eu.org/stream/p3ciphers/sosemanuk/sosemanuk_p3.pdf. (2005).

Daniel Bernstein. 2005. Cache-timing attacks on AES. http://cr.yp.to/antiforgery/cachetiming-20050414.pdf. (2005).

Daniel Bernstein. 2015a. Leaks. http://cr.yp.to/streamciphers/leaks.html. (Accessed: 7 January 2015).

Daniel Bernstein. 2015b. Snuffle 2005: the Salsa20 encryption function. http://cr.yp.to/snuffle.html. (Accessed: 7 January 2015).

Martin Boesgaard, Mette Vesterager, Thomas Christensen, and Erik Zenner. 2005. The Stream Cipher Rabbit. http://www.ecrypt.eu.org/stream/p3ciphers/rabbit/rabbit_p3.pdf. (2005).

François Bourdoncle. 1993. Efficient Chaotic Iteration Strategies With Widenings. In *FMPA*. Springer.

Christelle Braun, Konstantinos Chatzikokolakis, and Catuscia Palamidessi. 2009. Quantitative notions of leakage for one-try attacks. *Electronic Notes in Theoretical Computer Science* 249 (2009), 75–91.

Adam Chlipala. 2006. Modular development of certified program verifiers with a proof assistant. In *ICFP*. ACM, 160–171.

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

David Cock, Qian Ge, Toby Murray, and Gernot Heiser. 2014. The Last Mile: An Empirical Study of Timing Channels on seL4. In *CCS*. ACM.

Code Beach. 2008. Sorting Algorithms. http://www.codebeach.com/2008/09/sorting-algorithms-in-c.html. (2008). Accessed: 7 January 2015.

Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. 2009. Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors. In *SSP*. IEEE, 45–60.

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: a unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *POPL*. 238–252.

Patrick Cousot and Radhia Cousot. 1979. Systematic Design of Program Analysis Frameworks. In *POPL*.

Patrick Cousot, Radhia Cousot, and Laurent Mauborgne. 2012. Theories, Solvers and Static Analysis by Abstract Interpretation. *J. ACM* 59, 6 (2012), 31.

Yevgeniy Dodis, Rafail Ostrovsky, Leonid Reyzin, and Adam Smith. 2008. Fuzzy Extractors: How to Generate Strong Keys from Biometrics and Other Noisy Data. *SIAM J. Comput.* 38, 1 (2008), 97–139.

Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael B. Abu-Ghazaleh, and Dmitry Ponomarev. 2012. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *TACO* 8, 4 (2012), 35.

Goran Doychev, Dominik Feld, Boris Köpf, Laurent Mauborgne, and Jan Reineke. 2013. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. In *22nd USENIX Security Symposium*. USENIX.

Stefan Dziembowski and Krzysztof Pietrzak. 2008. Leakage-Resilient Cryptography. In *FOCS*. IEEE.

ECRYPT. 2012. The eSTREAM Portfolio in 2012. http://www.ecrypt.eu.org/documents/D.SYM.10-v1.pdf. (2012).

Úlfar Erlingsson and Martín Abadi. 2007. *Operating system protection against side-channel attacks that exploit memory latency*. Technical Report.

Dominik Feld. 2013. *Relational Domains for the Quantification of Cache Side Channels*. Master's thesis. Saarland University.

Christian Ferdinand, Florian Martin, Reinhard Wilhelm, and Martin Alt. 1999. Cache behavior prediction by abstract interpretation. *Science of Computer Programming* 35, 2 (1999), 163 – 189.

Bryan Ford. 2012. Plugging Side-Channel Leaks with Timing Information Flow Control. In *HotCloud*. USENIX.

Daniel Grund. 2012. *Static Cache Analysis for Real-Time Systems – LRU, FIFO, PLRU*. Ph.D. Dissertation. Saarland University.

Shay Gueron. 2010. Intel Advanced Encryption Standard (AES) Instructions Set. http://software.intel.com/file/24917. (2010).

David Gullasch, Endre Bangerter, and Stephan Krenn. 2011. Cache Games - Bringing Access-Based Cache Attacks on AES to Practice. In *SSP*. IEEE, 490–505.

Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. 2003. The Influence of Processor Architecture on the Design and the Results of WCET Tools. *IEEE Proceedings on Real-Time Systems* 91, 7 (2003), 1038–1054.

Daniel Hedin and David Sands. 2005. Timing Aware Information Flow Security for a JavaCard-like Bytecode. *ENTCS* 141, 1 (2005), 163–182.

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

Suman Jana and Vitaly Shmatikov. 2012. Memento: Learning Secrets from Process Footprints. In *SSP*. IEEE, 143–157.

Emilia Käsper and Peter Schwabe. 2009. Faster and Timing-Attack Resistant AES-GCM. In *CHES*. 1–17.

Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. 2012. StealthMem: System-level Protection Against Cache-Based Side Channel Attacks in the Cloud. In *19th USENIX Security Symposium*. USENIX.

Johannes Kinder, Florian Zuleger, and Helmut Veith. 2009. An Abstract Interpretation-Based Framework for Control Flow Reconstruction from Binaries. In *VMCAI*. Springer, 214–228.

Paul Kocher. 1996. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *CRYPTO*. Springer, 104–113.

Paul Kocher, Joshua Jaffe, and Benjamin Jun. 1999. Differential Power Analysis. In *CRYPTO*. Springer.

Boris Köpf and David Basin. 2007. An Information-Theoretic Model for Adaptive Side-Channel Attacks. In *CCS*. ACM, 286–296.

Boris Köpf, Laurent Mauborgne, and Martín Ochoa. 2012. Automatic Quantification of Cache Side-Channels. In *CAV*. Springer, 564–580.

Boris Köpf and Andrey Rybalchenko. 2010. Approximation and Randomization for Quantitative Information-Flow Analysis. In *CSF*. IEEE, 3–14.

Boris Köpf and Geoffrey Smith. 2010. Vulnerability bounds and leakage resilience of blinded cryptography under timing attacks. In *CSF*. IEEE, 44–56.

Gregor Leander, Erik Zenner, and Philip Hawkes. 2009. Cache timing analysis of LFSR-based stream ciphers. In *Cryptography and Coding*. Springer, 433–445.

Laurent Mauborgne and Xavier Rival. 2005. Trace Partitioning in Abstract Interpretation Based Static Analyzers. In *ESOP (LNCS)*, Vol. 3444. Springer, 5–20.

Ziyuan Meng and Geoffrey Smith. 2011. Calculating Bounds on Information Leakage Using Two-Bit Patterns. In *PLAS*. ACM.

David Molnar, Matt Piotrowski, David Schultz, and David Wagner. 2006. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *Information Security and Cryptology-ICISC 2005*. Springer, 156–168.

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

Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: the Case of AES. In *CT-RSA (LNCS)*, Vol. 3860. Springer, 1–20.

Goutam Paul and Shashwat Raizada. 2012. Impact of extending side channel attack on cipher variants: a case study with the HC series of stream ciphers. In *Security, Privacy, and Applied Cryptography Engineering*. Springer, 32–44.

Colin Percival. 2005. Cache missing for fun and profit. In *BSDCan*.

Himanshu Raj, Ripal Nathuji, Abhishek Singh, and Paul England. 2009. Resource management for isolation enhanced cloud services. In *Proc. ACM Cloud Computing Security Workshop, (CCSW)*. 77–84.

Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 2009. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *CCS*. ACM, 199–212.

Geoffrey Smith. 2009. On the Foundations of Quantitative Information Flow. In *FoSSaCS*. Springer.

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

Zhenghong Wang and Ruby B. Lee. 2007. New cache designs for thwarting software cache-based side channel attacks. In *ISCA*. ACM, 494–505.

Zhenghong Wang and Ruby B. Lee. 2008. A novel cache architecture with enhanced performance and security. In *41st IEEE/ACM Intl. Symposium on Microarchitecture (MICRO)*. 83–93.

Hongjun Wu. 2004. The Stream Cipher HC-128. http://www.ecrypt.eu.org/stream/p3ciphers/hc/hc128_p3.pdf. (2004).

Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *Proc. 23rd USENIX Security Symposium*. 719–732.

Yu Yu, François-Xavier Standaert, Olivier Pereira, and Moti Yung. 2010. Practical leakage-resilient pseudorandom generators. In *CCS*. ACM, 141–151.

Erik Zenner. 2009. A cache timing analysis of HC-256. In *Selected Areas in Cryptography*. Springer.

Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. 2012a. Language-based control and mitigation of timing channels. In *PLDI*. ACM, 99–110.

Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2012b. Cross-VM Side Channels and Their Use to Extract Private Keys. In *CCS*. ACM.