Automatic Quantification of Cache Side-Channels

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

[1] IMDEA Software Institute, Spain
[2] Siemens AG, Germany
[3] TU Dortmund, Germany
{boris.koepf,laurent.mauborgne}@imdea.org
martin.ochoa@cs.tu-dortmund.de

Abstract. The latency gap between caches and main memory has been successfully exploited for recovering sensitive input to programs, such as cryptographic keys from implementation of AES and RSA. So far, there are no practical general-purpose countermeasures against this threat. In this paper we propose a novel method for automatically deriving upper bounds on the amount of information about the input that an adversary can extract from a program by observing the CPU's cache behavior. At the heart of our approach is a novel technique for efficient counting of concretizations of abstract cache states that enables us to connect state-of-the-art techniques for static cache analysis and quantitative information-flow. We implement our counting procedure on top of the AbsInt TimingExplorer, one of the most advanced engines for static cache analysis. We use our tool to perform a case study where we derive upper bounds on the cache leakage of a 128-bit AES executable on an ARM processor with a realistic cache configuration. We also analyze this implementation with a commonly suggested (but until now heuristic) countermeasure applied, obtaining a formal account of the corresponding increase in security.

1 Introduction

Many modern computer architectures use caches to bridge the latency gap between the CPU and main memory. Caches are small, fast memories that store the contents of previously accessed main memory locations; they can improve the overall performance because typical memory access patterns exhibit locality of reference. On today's architectures, an access to the main memory (i.e. a cache miss) may imply an overhead of hundreds of CPU cycles w.r.t. an access to the cache (cache hit).

While the use of caches is beneficial for performance reasons, it can have negative effects on security: An observer who can measure the time of memory lookups can see whether a lookup is a cache hit or miss, thereby learning partial information about the state of the cache. This partial information has been used for extracting cryptographic keys from implementations of AES [14,24,37], RSA [39], and DSA [8]. In particular AES is vulnerable to such cache attacks, because most high-speed software implementations make heavy use of look-up

tables. Cache attacks are the most effective known attacks against AES and allow to recover keys within minutes [24].

A number of countermeasures have been proposed against cache attacks. They can be roughly put in two classes: (1) Avoiding the use of caches for sensitive computations. This can be achieved, e.g. by using dedicated hardware implementations (For example, recent Intel processors offer support for AES [4]), or by side-stepping the use of caches in software implementations [27]. Both solutions obviously defeat cache attacks; however they are not universally applicable (i.e. to arbitrary programs), e.g. due to lack of available hardware support, or for reasons of performance. (2) Mitigation strategies for eliminating attack vectors and reducing leakage. Proposals include disabling high-resolution timers, hardening of schedulers [24], and preloading [14, 37] of tables. Such strategies are implemented, e.g. in the OpenSSL 1.0 [7] version of AES, however, their effectiveness is highly dependent on the operating system and the CPU. Without considering/modeling all implementation details, such mitigation strategies necessarily remain heuristic. In summary, there is no general-purpose countermeasure against cache attacks that is backed-up by mathematical proof.

In this paper, we propose a novel method for establishing formal security guarantees against cache-attacks that is applicable to arbitrary programs and a wide range of embedded platforms. The guarantees we obtain are upper bounds on the amount of information about the input that an adversary can extract by observing which memory locations are present in the CPU's cache after execution of the program; they are based on the actual program binary and a concrete processor model and can be derived entirely automatically. At the heart of our approach is a novel technique for effective counting of concretizations of abstract states that enables us to connect state-of-the-art techniques for static cache analysis and quantitative information-flow analysis.

Technically, we build on prior work on static cache analysis [23] that was primarily used for the estimation of worst-case execution time by abstract interpretation [20]. There, two abstract domains for cache-states are introduced; one of them captures a superset of the memory locations that *may* be in the cache, the other captures a subset of the memory locations that *must* be in the cache. For the purpose of this paper it suffices to know that both abstract analyses are sound, i.e. that each of them computes a superset of the set of reachable cache-states. We also leverage techniques from quantitative-information-flow analysis that enable establishing bounds for the amount of information that a program leaks about its input. One key observation is that (an upper bound on) the number of reachable states of a program corresponds to (an upper bound on) the number of leaked bits [31, 42]. Such upper bounds can be obtained by computing super-sets of the set of reachable states by abstract interpretation, and by determining their sizes [31].

We develop a novel technique for counting the number of cache states represented by the abstract states of the static cache analyses described above. In particular, we give formulas and algorithms for computing the respective sizes of the set of states represented by the may- and must-analysis, and for their intersection. We give a concise implementation of our counting procedures in Haskell [6] and we connect this counting engine to AbsInt's a^3 [1], the state-of-the-art tool for static cache analysis. a^3 efficiently analyzes binary code based

on accurate models of several modern embedded processors with a wide range of cache types (e.g. data caches, instruction caches, or mixed) and replacement strategies. Using this tool-chain, we perform an analysis of a binary implementation of 128-bit AES from the PolarSSL library [5], based on a 32-bit ARM processor with a 4-way set associative data cache with LRU replacement strategy. We analyze this implementation with and without the preloading countermeasure applied, with different cache sizes, and for two different adversary models, obtaining the following results.

Without preloading, the derived upper bounds for the leakage (about the payload and the key) in one execution exceed the size of the key and are hence too large for practical use. With preloading and a powerful adversary model, however, the derived bounds drop to values ranging from 55 to 1 bits, for cache sizes ranging from 16KB to 128KB. With a less powerful but realistic adversary model, the bounds drop even further to ranges from 6 to 0 bits, yielding strong security guarantees. This case study shows that the automated, formal security analysis of realistic cryptosystems and accurate real processor models is in fact feasible.

In summary, our contributions are threefold. Conceptually, we show how state-of-the-art tools for quantitative information-flow analysis and static cache analysis can be combined for quantifying cache side-channels. Technically, we develop and implement novel methods for counting abstract cache states. Practically, we perform a formal cache-analysis of a binary AES 128 implementation on a realistic processor model.

2 Preliminaries

In this section we revisit concepts from quantitative information-flow analysis. In particular, we introduce measures of confidentiality based on information theory in Section 2.1, and we present techniques for their approximation in Section 2.2.

2.1 Quantifying Information Leaks

A (deterministic) *channel* is a function $C\colon S \to O$ mapping a finite set of secrets S to a finite set of observations O. We characterize the security of a channel in terms of the difficulty of guessing the secret input from the observation. This difficulty can be captured using information-theoretic entropy, where different notions of entropy correspond to different kinds of guessing [17]. In this paper, we focus on min-entropy as a measure, because it is associated with strong security guarantees [42].

Formally, we model the choice of a secret input by a random variable X with $ran(X) = S$ and the corresponding observation by a random variable Y with $ran(Y) = O$. The dependency between X and Y is formalized as a conditional probability distribution $P_{Y|X}$ with $P_{Y|X}(o, s) = 1$ if $C(s) = o$, and 0 otherwise. We consider an adversary that wants to determine the value of X from the value of Y, where we assume that X is distributed according to P_X. The adversary's a priori uncertainty about X is given by the *min-entropy* [41]

$$H_\infty(X) = -\log_2 \max_s P_X(s)$$

of X, which captures the probability of correctly guessing the secret in one shot. The adversary's a posteriori uncertainty is given by the *conditional min-entropy*

$H_\infty(X|Y)$, which is defined by

$$H_\infty(X|Y) = -\log_2 \sum_o P_Y(o) \max_s P_{X|Y}(s,o)$$

and captures the probability of guessing the value of X in one shot when the value of Y is known.

The *(min-entropy) leakage* L of a channel with respect to the input distribution P_X is the reduction in uncertainty about X when Y is observed,

$$L = H_\infty(X) - H_\infty(X|Y) \ ,$$

and is the logarithm of the factor by which the probability of guessing the secret is reduced by the observation. Note that L is not a property of the channel alone as it also depends on P_X. We eliminate this dependency as follows.

Definition 1 (Maximal Leakage). *The* maximal leakage *ML of a channel C is the maximal reduction in uncertainty about X when Y is observed*

$$ML(C) = \max_{P_X}(H_\infty(X) - H_\infty(X|Y)) \ ,$$

where the maximum is taken over all possible input distributions.

For computing an upper bound for the maximal leakage of a deterministic channel, it suffices to compute the size of the range of C. While these bounds can be coarse in general, they are tight for uniformly distributed input.

Lemma 1.

$$ML(C) \leq \log_2 |C(S)| \ ,$$

where equality holds for uniformly distributed P_X.

Proof. The maximal leakage of a (probabilistic) channel specified by the distribution $P_{Y|X}$ can be computed by $ML(P_{Y|X}) = \log_2 \sum_o \max_s P_{Y|X}(o,s)$, where the maximum is assumed (e.g.) for uniformly distributed input [15,32]. For deterministic channels, the number of non-zero (hence 1) summands matches $|C(S)|$.

2.2 Static Analysis of Channels

In this paper we consider channels *of programs*, which are channels that are given by the semantics of (deterministic, terminating) programs. In this setting, the set of secrets is a part of the initial state of the program, and the set of observables is a part of the final state of the program. Due to Lemma 1, computing upper bounds on the maximal leakage of a program can be done by determining the set of final states of the program. Computing this set from the program code requires computation of a fixed-point and is not guaranteed to terminate for programs over unbounded state-spaces. Abstract interpretation [20] overcomes this fundamental problem by resorting to an approximation of the state-space and the transition relation. By choosing an adequate approximation one can enforce termination of the fixed-point computation after a finite number of steps. The soundness of the analysis follows from the soundness of the abstract domain, which is expressed in terms of a *concretization function* (denoted γ) relating

elements of the abstract domain to concrete properties of the program, ordered by implication.

For the purpose of this paper, we define soundness with respect to a channel, i.e., we will use a concretization function mapping to sets of observables (where implication corresponds to set inclusion).

Definition 2. *An abstract element $t^\sharp$ is* sound *for a concretization function γ with respect to a channel $C : S \to O$ if and only if $C(S) \subseteq \gamma(t^\sharp)$.*

The following theorem is an immediate consequence from Lemma 1; it states that a counting procedure for $\gamma(t^\sharp)$ can be used for deriving upper bounds on the amount of information leaked by C.

Theorem 1. *Let $t^\sharp$ be sound for γ with respect to C. Then*

$$ML(C) \leq \log_2 \left|\gamma(t^\sharp)\right| \ .$$

For a more detailed account of the connection between abstract interpretation and quantitative information-flow, see [31].

3 Cache Channels

In this section, we define channels corresponding to two adversary models that can only observe cache properties. We also revisit two abstract domains for reasoning about cache-states and show how they relate to those channels. We begin with a primer on caching.

3.1 Caches

Typical caches work as follows. The main memory is partitioned into *blocks* of size B that are referenced using locations *loc*. A cache consists of a number of *sets*, each containing a fixed number of *lines* that can each store one memory block. The size A of the cache sets is called the *associativity* of the cache. Each memory block can reside in exactly one cache set, which is determined by the block's location. We can formally define a single *cache set* as a mapping

$$t \colon \{1, \ldots, A\} \to loc \cup \{\bot\} \ ,$$

from line numbers to locations, where $\bot$ represents an empty line. The mapping t is injective, which captures that a memory block is stored in at most one line. A *cache* is a tuple of independent cache sets. For simplicity of presentation, we focus on single cache sets throughout the paper, except for the case study in Section 6.

What happens when a memory block is requested depends on the *replacement strategy*. Here we focus on the LRU (Least Recently Used) strategy, which is used e.g. in the Pentium I processor. With LRU, each cache set forms a queue. When a memory block is requested, it is appended to the head of the queue. If the block was already stored in the cache (cache hit), it is removed from its original position; if not (cache miss), it is fetched from main memory. Due to the queue structure of sets, memory blocks *age* when other blocks are looked up, i.e. they move towards the tail of the queue and (due to the fixed length of the queue) are eventually removed. For a formalization of the LRU set update function see [23]

or Appendix A. For a formalization of alternative update functions, such as FIFO (First In First Out) see [40]. Depending on the concrete processor model, data and instructions are processed using dedicated caches or a common one [23]. Unless mentioned otherwise (e.g. in the experiments on AES), our results hold for any cache analysis that is sound.

3.2 Two Adversary Models Observing the Cache

We consider a scenario where multiple processes share a common CPU. We assume that one of these processes is adversarial and tries to infer information about the computations of a victim process by inspecting the cache after termination. We distinguish between two adversaries Adv_{prec} and Adv_{prob}. Both adversaries can modify the initial state of the cache with memories in their virtual memory space, which we assume is not shared between processes, but they differ in their ability of observing the final cache state:

Adv_{prec} : This adversary can observe which memory blocks are contained in the cache at the end of the victim's computation.

Adv_{prob} : This adversary can observe which blocks of his virtual memory space are contained in the cache after the victim's computation.

Note that neither adversary can observe the actual data that is stored in each of the memory blocks that reside in the cache. The channel corresponding to the adversary Adv_{prec} simply maps the victim's input to the corresponding final cache state. The channel corresponding to Adv_{prob} can be seen as an abstraction of the channel corresponding to Adv_{prec}, as it can be described as the composition of the channel of Adv_{prec} with a function *blur* that maps all memory blocks not belonging to the adversary's virtual memory space to one undistinguishable element. Adv_{prob} corresponds to the adversaries encountered in synchronous "prime and probe" attacks [37], which observe the cache-state by performing accesses to different locations and use timing measurements to distinguish whether they are contained in the cache or not.

Considering that our adversary models allow some choice of the initial state, they formally define families of channels that are indexed by the adversarially chosen part of the initial cache. To give an upper bound on the leakage of all channels in those families we would need relational information, which is not supported by the existing cache analysis tools. One possible solution is to consider an abstract initial state approximating all possible adversary choices, which leads to imprecision in the analysis. In the particular case of a LRU replacement strategy, we can use the following property:

Proposition 1. *For caches with LRU strategy, the leakage to Adv_{prec} (Adv_{prob}) w.r.t. any initial cache state containing only memory locations from the adversary's memory space corresponds to the leakage to Adv_{prec} (Adv_{prob}) w.r.t. an empty initial cache state.*

This result follows from the following observation: for each initial cache state containing locations disjoint from the victim's memory space, the first i lines of the final cache state will contain the locations accessed by the victim, and the remaining lines will contain the first $A - i$ locations of the initial state shifted to the right, where i depends on that particular run of the victim. That is, modulo

the adversarial locations, the number of possible final cache states corresponding to an empty initial state matches the number of final cache states corresponding to an initial state that does not contain locations from the victim's memory space. The assertion then follows immediately from Theorem 1. Proposition 1 will be useful in our case study, since the analysis we use provides a more accurate final state when run with an initial empty cache.

3.3 Abstract Domains for Cache Analysis

Ferdinand et al. [23] propose abstract interpretation techniques for cache analysis and prove their soundness with respect to reachability of cache states, which corresponds to soundness w.r.t the channel of Adv_{prec} according to Definition 2. In particular, they present two abstract domains for cache-states: The first domain corresponds to a *may*-analysis and represents the set of memory locations that possibly reside in the cache. The second domain corresponds to a *must*-analysis and represents the set of memory locations that are definitely in the cache. In both cases, an abstract cache set is represented as a function

$$t^\sharp \colon \{1, \ldots, A\} \to 2^{loc}$$

mapping set positions to sets of memory locations, where $t^\sharp(i) \cap t^\sharp(j) = \emptyset$ whenever $i \neq j$. In the following we will use ${t_1}^\sharp$ and ${t_2}^\sharp$ for abstract sets corresponding to the *may* and *must* analysis respectively. For the *may* analysis, the concretization function $\gamma^\cup$ is defined by

$$\gamma^\cup({t_1}^\sharp) = \{t \mid \forall j \in \{1, \ldots, A\} \colon t(j) = \bot \ \vee \ \exists i \leq j : t(j) \in {t_1}^\sharp(i)\} \ .$$

This definition implies that each location that appears in the concrete state appears also in the abstract state, and the position in the abstract state is a lower bound for the position in the concrete. For the *must* analysis, the concretization function $\gamma^\cap$ is defined by

$$\gamma^\cap({t_2}^\sharp) = \{t \mid \forall i \in \{1, \ldots, A\} \colon \forall a \in {t_2}^\sharp(i) \colon \exists \ j \leq i : t(j) = a\} \ .$$

This definition implies that each location that appears in the abstract state is required to appear in the concrete, and its position in the abstract is an upper bound for its position in the concrete.

Example 1. Consider the following program running on a 4-way fully associative (i.e. only one set) data cache where $\ldots x \ldots$ stands for an instruction that references location x, and let e, a, b are pairwise distinct locations.

$$\mathsf{if} \ldots e \ldots \mathsf{then} \ldots a \ldots \mathsf{else} \ldots b \ldots$$

With an empty initial abstract cache before execution, the abstract may- and must-analyses return

$${t_1}^\sharp = [\{a, b\}, \{e\}, \{\}, \{\}] \quad \text{and} \quad {t_2}^\sharp = [\{\}, \{e\}, \{\}, \{\}]$$

as final states, respectively. The following caches states are contained in their respective concretizations:

$$[a, \bot, \bot, \bot], [a, b, e, \bot], [\bot, e, a, b] \in \gamma^\cup({t_1}^\sharp)$$
$$[a, e, \bot, \bot], [e, \bot, \bot, \bot], [\bot, e, a, b] \in \gamma^\cap({t_2}^\sharp)$$

Notice that both concretizations include the two possible states $[a, e, \bot, \bot]$ and $[b, e, \bot, \bot]$ (which is due to the soundness of the analyses) but also impossible states (which is due to the imprecision of the analysis). In particular, states in which empty cache lines are followed by non-empty cache lines are artifacts of the abstraction (i.e. they cannot occur according to the concrete cache semantics from [23], as we prove in the Appendix). More precisely, we have

$$\forall i, j \in \{1, \ldots, A\} \colon t(i) = \bot \wedge j > i \implies t(j) = \bot \ . \tag{1}$$

It is hence sufficient to consider only the concrete states that also satisfy (1), which enables us to derive tighter bounds in Section 4. For simplicity of notation we will implicitly assume that (1) is part of the definition of $\gamma^{\cup}$ and $\gamma^{\cap}$.

To obtain the channel corresponding to the adversary model Adv_{prob}, we just need to apply *blur* to the concretization of the must and may cache analysis, which is equivalent to first applying *blur* to the sets appearing in the abstract elements and then concretizing.

4 Counting Cache States

We have introduced channels corresponding to two adversaries, together with sound abstract interpretations. The final step needed for obtaining an automatic quantitative information-flow analysis from Theorem 1 are algorithms for counting the concretizations of the abstract cache states presented in Section 3.3, which we present next. As before, we restrict our presentation to single cache sets. Counting concretizations of caches with multiple sets can be done by taking the product of the number of concretizations of each set.

4.1 Concrete states respecting *may*

We begin by deriving a formula for counting the concretizations of an abstract may-state ${t_1}^\sharp$. To this end, let $n_i = \left|{t_1}^\sharp(i)\right|$, $n_i^* = \sum_{j=1}^{i} n_j$, for all $i \in \{1, \ldots, A\}$ and $n^* = n_A^*$. The definition of $\gamma^{\cup}({t_1}^\sharp)$ informally states that when reading the content of $t^\sharp$ and $t \in \gamma^{\cup}({t_1}^\sharp)$ from head to tail in lockstep, each non-empty line in t has appeared in the same or a previous line of ${t_1}^\sharp$. That is, for filling line k of t there are n_k^* possibilities, of which $k-1$ are already used for filling lines $1, \ldots, k-1$. The number of concrete states with a fixed number i of non-empty lines is hence given by

$$\prod_{k=1}^{i} (n_k^* - (k-1)) \tag{2}$$

As the definition of $\gamma^{\cup}$ does not put a lower bound on the number i of nonempty lines, we need to consider all $i \in \{1, \ldots, A\}$. We obtain the following explicit formula for the number of concretizations of ${t_1}^\sharp$.

Proposition 2 (Counting May).

$$\left|\gamma^{\cup}({t_1}^\sharp)\right| = \sum_{i=0}^{A} \prod_{k=1}^{i} (n_k^* - (k-1))$$

Example 2. When applied to the abstract may-state ${t_1}^\sharp = [\{a, b\}, \{e\}, \{\}, \{\}]$ obtained from the analysis of the program in Example 1 we obtain $\left|\gamma^{\cup}({t_1}^\sharp)\right| = 11$, which illustrates that the bounds obtained by Proposition 2 can be coarse.

4.2 Concrete states respecting *must*

For counting the concretizations of an abstract must-state ${t_2}^\sharp$, let $m_i = \left|{t_2}^\sharp(i)\right|$, $m_i^* = \sum_{j=1}^{i} m_j$, for all $i \in \{1, \ldots, A\}$ and $m^* = m_A^*$. The definition of $\gamma^\cap$ informally states that when reading the lines of an abstract state ${t_2}^\sharp$ and a concrete state $t \in \gamma^\cap({t_2}^\sharp)$ from head to tail in lockstep, each element of ${t_2}^\sharp$ has already appeared in the same or a previous line of t. More precisely, the m_j elements contained in line j of ${t_2}^\sharp$ appear in lines $1, \ldots, j$ of t, of which m_{j-1}^* are already occupied by the must-constraints of lines $1, \ldots, j-1$. This leaves $\binom{j-m_{j-1}^*}{m_j}\, m_j!$ possibilities for placing the elements of ${t_2}^\sharp(j)$, which amounts to a total of

$$\prod_{j=1}^{A} \binom{j - m_{j-1}^*}{m_j}\, m_j! \tag{3}$$

possibilities for placing all elements in ${t_2}^\sharp$. However, notice that $m^* \leq A$ is possible, i.e. must-constraints can leave cache lines unspecified. The number of possibilities for filling those unspecified lines is

$$\prod_{k=m^*+1}^{A} (\ell - (k-1)) \;, \tag{4}$$

where $\ell = |loc|$ is the number of possible memory locations.

Finally, observe that (3) and (4) count concrete states in which each line is filled. However, the definition $\gamma^\cap$ only mandates that at least m^* lines of each concrete state be filled. We account for this by introducing a variable i that ranges from m^* to A. We modify (3) by choosing from $\min(i, j)$ instead of j positions[4] and we modify (4) by replacing the upper bound by i. This yields the following for explicit formula for the number of concretizations of ${t_2}^\sharp$.

Proposition 3 (Counting Must).

$$\left|\gamma^\cap({t_2}^\sharp)\right| = \sum_{i=m^*}^{A} \left(\prod_{j=1}^{A} \binom{\min(i,j) - m_{j-1}^*}{m_j}\, m_j! \prod_{k=m^*+1}^{i} (\ell - (k-1)) \right)$$

Example 3. When applied to the must-state ${t_2}^\sharp = [\{\}, \{e\}, \{\}, \{\}]$ and a set of locations $loc = \{a, b, c, d, e\}$, Proposition 3 yields a number of 81 concretizations of ${t_2}^\sharp$. This over-approximation stems from the fact that the abstract state requires only the containment of e and that the rest of the lines can be chosen from loc.

We next tackle this imprecision by considering the intersection of may and must.

4.3 Concrete states respecting *must* and *may*

For computing the number of concrete states respecting both ${t_2}^\sharp$ and ${t_1}^\sharp$ we reuse the notation introduced in Sections 4.1 and 4.2. As in Section 4.2 we use (3) for counting the cache lines constrained by the must-information. However, instead

[4] The index j still needs to go up to A in order to collect all constraints

of filling the unconstrained lines with *all* possible memory locations, we now choose only from the lines specified by the may-information. The counting is similar to equation (2), the difference being that, as in (4), the product starts with $k = m^* + 1$ because the content of m^* lines is already fixed by the must-constraints. The key difference to (4) is that now we pick only from at most n_k^* lines instead of ℓ lines. We obtain the following proposition.

Proposition 4 (Counting May and Must).

$$\left|\gamma^{\cup}(t_1{}^{\sharp}) \cap \gamma^{\cap}(t_2{}^{\sharp})\right| \leq \sum_{i=m^*}^{A} \left(\prod_{j=1}^{A} \binom{\min(i,j) - m_{j-1}^*}{m_j} m_j! \prod_{k=m^*+1}^{i} (n_k^* - (k-1)) \right)$$

Two comments are in order. First, notice that the inequality Proposition 4 stems from the fact that the lines unconstrained by the must-information may be located at positions $j < k$. Using the constraint n_j^* instead of n_k^* would lead to tighter bounds, however, an explicit formula for this case remains elusive. Second, observe that the rightmost product is always non-negative. For this it is sufficient to prove that the first factor $n_{m^*+1}^* - m^*$ is non-negative, because the value of subsequent factors decreases by at most 1. Assume that $n_{m^*+1}^* - m^* < 0$ (and hence $n_{m^*}^* < m^*$). By (1), $n_j^* < j$ implies that line j is empty for all concrete states, which for $j = m^*$ contradicts the requirement that all states contain at least m^* lines.

Example 4. When applied to the abstract cache states $t_1{}^{\sharp} = [\{a,b\},\{e\},\{\},\{\}]$ and $t_2{}^{\sharp} = [\{\},\{e\},\{\},\{\}]$ from Example 1, Proposition 4 delivers a total of 9 concrete states.

It is easy to see that the expression in Proposition 4 can be evaluated in time $O(A^3)$ because both the factorial and and n_i^* can be computed in linear time and they are nested in two loops of length at most A. Although efficient, an approximation using Proposition 4 can be coarse: In Example 4 we computed a bound of 9 states, although (as is easily verified manually) there are only 4 concrete states respecting the constraints of both abstract states. We have developed more accurate (but more complex) variants of Proposition 4 that yield the exact bounds for this example, however, they are also not tight in general.

In the absence of a closed expression for the exact number of concrete states, one can proceed by enumerating the set of all concrete states respecting may, and filtering out those not respecting must. We present an implementation of the exact counting by enumeration in Section 5.2 The price to pay for this brute-force approach is a worst-case time complexity of $O(A!)$, e.g. if there are no must-constraints and the first location of the abstract may-state contains A or more locations. This is not a limitation for the small associativities often encountered in practice ($A = 2$ or $A = 4$), however, for fully associative caches in which A equals the total number of lines of the cache, the approximation given by Proposition 4 is the more adequate tool.

4.4 Counting for Probing Adversaries

For counting the possible observations of Adv_{prob} for arbitrary replacement strategies, we can apply the techniques presented above to previously blurred

abstract states. For the case of a LRU strategy, we obtain the following better bounds.

Proposition 5. *The number of observations Adv_{prob} can make is bounded by*

$$min(n^*, A) - m^* + 1$$

The assertion follows from the fact that, after the computation, each cache set will first contain the victim's locations (which Adv_{prob} cannot distinguish), and then a fixed sequence of locations from the adversary's virtual memory whose length only depends on the number of the victim's blocks. I.e., when starting from an empty cache set, the adversary can only observe the length of the final cache set. This size is at least m^* (because at least that number of lines must be filled), and at most $\min(n^*, A)$. The additional 1 accounts for the empty state.

5 Implementation

In this section we report on the implementation of a tool for quantifying cache leaks. Its building blocks are the AbsInt a^3 tool for static cache analysis, and a novel counting engine for cache-states based on the results presented in Section 4.

5.1 Abstract Interpreter

The AbsInt a^3 [1] is a suite of industral-strength tools for the static analysis of embedded systems. In particular, a^3 comprises tools (called aiT and TimingExplorer) for the estimation of worst-case execution times based on the static cache analysis by Ferdinand et al. [23]. The tools cover a wide range of CPUs, such as ERC32, TriCore, M68020, LEON3 and several PowerPC models (aiT), as well as CPU models with freely configurable LRU cache (TimingExplorer). We base our implementation on the TimingExplorer for reasons of flexibility.

The TimingExplorer receives as input a program binary and a cache configuration and delivers as output a control flow graph in which each (assembly-level) instruction is annotated by the corresponding abstract *may* and *must* information, where memory locations are represented by strings, abstract cache lines are lists of memory locations, abstract sets are lists of abstract lines, and abstract caches are lists of abstract sets. We extract the annotations of the final state of the program, and provide them as input to the counting engine.

5.2 Counting Engine

We implemented an engine for counting the concretizations of abstract cache states according to the development in Section 4. Our language of choice is Haskell [6], because it allows for a concise representation of sums, products, and enumerations using list comprehensions. For brevity of presentation, we give only the procedures for exact counting sketched in Section 4.3.

We use the following data types for representing abstract cache sets, which matches the output of the TimingExplorer described above.

```
type Loc = String                type ConcreteSet = [Loc]
type AbstractLine = [Loc]        type AbstractSet = [AbstractLine]
```

The function `allStates` is the core of the exact counting of concrete cache states in the intersection defined by may and must.

```
allStates :: AbstractSet -> AbstractSet -> [ConcreteSet]
allStates may must = filter (checkMust must) (genAllMay may)
```

As described in Section 4.3, this is achieved by enumerating all concrete states that satisfy a given set of may-constraints (done by `genAllMay`), and keeping only those that also satisfy the must-constraints (done by filtering with `checkMust`). At the core of the function `genAllMay` is the following function `genMay` that returns all concretizations of the same length as the given abstract set,

```
genMay:: AbstractSet -> [ConcreteSet]
genMay (a:as) = [c:cs| c<-a, cs<-genMay (carry (delete c a) as)]
genMay [] = [[]]
```

where it relies on a function `carry` that carries unused may-constraints to the next line of the abstract state.

Finally, the function `checkMust` tests whether a concrete set satisfies the must-constraints, by checking whether all elements in line number `n` (denoted by `as!!(n-1)`) of the abstract state also appear in the prefix of length `n` of the concrete state.

```
checkMust :: AbstractSet -> ConcreteSet -> Bool
checkMust as cs = and [elem a (take n cs)| n<-[1..length as],
                                           a<-as!!(n-1)]]
```

6 Case Study

In this section we report on a case-study where we use the methods developed in this paper for analyzing the cache side-channel of a widely used AES implementation on a realistic processor model with different cache configurations.

6.1 Target Implementations

Code. We analyze the implementation of 128 bit AES encryption from the PolarSSL library [5], a lightweight crypto suite for embedded platforms. As is standard for software implementations of AES, the code consists of single loop (corresponding to the rounds of AES) in which heavy table lookups are performed to indices computed using bit-shifting and masking, see Appendix C for details. We also analyze a modified version of this implementation, where we add a loop that loads the entire lookup table into the cache before encryption. This preloading has been suggested as countermeasure against cache attacks because, intuitively, all lookups during encryption will hit the cache.

Platform. We compile the AES C source code into a binary for the ARM7TDMI [2] CPU using the GNU ARM GCC compiler [3]. Although the original ARM7TDMI does *not* have any caches, the AbsInt TimingExplorer supports this CPU with the possibility of specifying arbitrary configurations of data/instruction/mixed caches with LRU strategy. For our experiments we use data caches with sizes of 16-128 KB, associativity of 4 ways, and a line size of 32 Bytes, which are common configurations in practice.

6.2 Improving Precision by Partitioning

The TimingExplorer can be very precise for simple expressions, but loses precision when analyzing array lookups to non-constant indexes. This source of imprecision is well-known in static analysis, and abstract interpretation offers techniques to regain precision, such as abstract domains specialized for arrays [21], or automatic refinement of transfer functions. For our analysis, we use results on *trace partitioning* [33], which consists in performing the analysis on a partition of all possible runs of a program, each partition yielding more precise results.

We have implemented a simple trace partitioning strategy using program transformations that do not modify the data cache (which is crucial for the soundness of our approach). For each access to the look-up table, we introduce conditionals on the index, where each branch corresponds to one memory block, and we perform the table access in all branches. As the conditionals cover all possible index values for the table access, we add one memory access to the index before the actual table look-up, which does not change the cache state for an LRU cache strategy, since the indices have to be fetched before accessing the table anyway. An example of the AES code with trace partitioning can be found in Appendix C.

Note that the same increase in precision could be achieved without program transformation if the trace partitioning were implemented at the level of the abstract interpreter, which would also allow us to consider instruction caches and cache strategies beyond LRU. Given that the TimingExplorer is closed-source, we opted for partitioning by code transformation.

6.3 Results and Security Interpretation

The results of our analysis with respect to the adversary Adv_{prec} are depicted in Figure 1. For AES without preloading of tables, the bounds we obtained exceed 160 bits for all cache sizes. For secret keys of only 128 bits, they are not precise enough for implying meaningful security guarantees. With preloading, however, those bounds drop down to 55 bits for caches sizes of 16KB and to only 1 bit for sizes of 128KB, showing that only a small (in the 128KB case) fraction of the key bits can leak in one execution.

The results of our analysis with respect to the (less powerful, but more realistic) adversary Adv_{prob} are depicted in Figure 2. As for Adv_{prec}, the bounds obtained without preloading exceed the size of the secret key. With preloading, however, they remain below 6 bits and even drop to 0 bits for caches of 128KB, giving a formal proof of noninterference for this implementation and platform.

To formally argue tightness of the non-zero bounds, we would need to show that this information can be effectively recovered (i.e. devise an attack), which is out of the scope of this paper. Manual inspection of the final cache states shows that the non-zero bounds stem from AES tables sharing the same set with other memory locations used by the AES code, which may indeed be exploitable.

7 Prior Art

Timing attacks against cryptosystems date back to [28]. They can be divided into those exploiting timing variations due to control-flow [16,28] and those exploiting timing variations of the execution platform, e.g. due to caches [9,11,14,37,38,39], or branch prediction units [10]. In this paper we focus solely on caching.

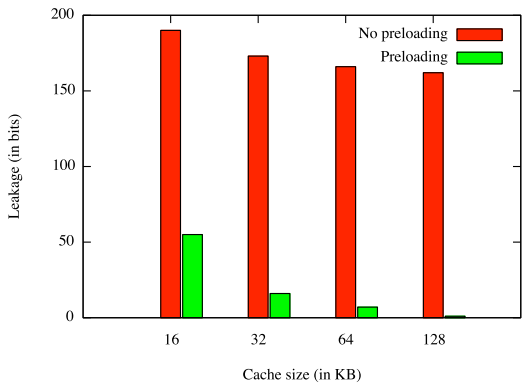

Fig. 1. Upper bounds for the maximal leakage w.r.t. the adversary Adv_{prec} and a 4-way set associative cache with 32B lines of sizes 16KB-128KB

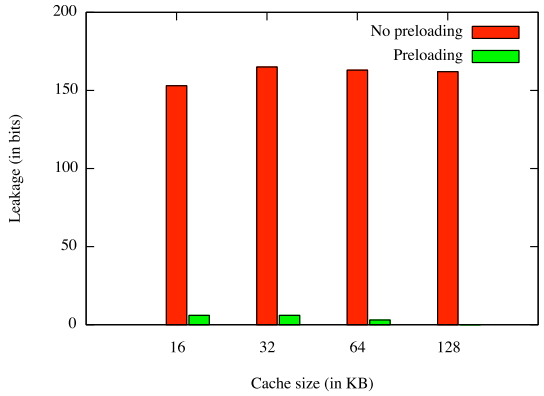

Fig. 2. Upper bounds for the maximal leakage w.r.t. the adversary Adv_{prob} and a 4-way set associative cache with 32B lines of sizes 16KB-128KB

The literature on cache attacks is stratified according to a variety of different adversary models: In *time-driven attacks* [11, 14] the adversary can observe the overall execution time of the victim process and estimate the overall number of cache hits and misses. In *trace-driven attacks* [9] the adversary can observe whether a cache hit or miss occurs, for every single memory access of the victim process. In *access-driven attacks* [37,39] the adversary can probe the cache either during computation (*asynchronous* attacks) or after completion (*synchronous* attacks) of the victim's computation, giving him partial information about the memory locations accessed by the victim. Finally, some attacks assume that the adversary can choose the cache state before execution of the victim process [37], whereas others only require that the cache does not contain the locations that are looked-up by the victim during execution [11]. The information-theoretic bounds we derive hold for single executions of synchronous access-driven adversaries, where we consider initial states that do not contain the victim's data. The derivation of bounds for alternative adversary models is left future work.

A number of mitigation techniques have been proposed to counter cache attacks. Examples include coding guidelines [19] for thwarting cache attacks on x86

CPUs, or novel cache-architectures that are more resistant to cache attacks [44]. One commonly proposed mitigation technique is preloading of tables [14, 37]. However, as first observed by [14], it is a non-trivial issue to establish the efficacy of this countermeasure. As [37] comments:

> [...], it should be ensured that the table elements are not evicted by the encryption itself, by accesses to the stack, inputs or outputs. Ensuring this is a delicate architecture-dependent affair [...]."

The methods developed in this paper enable us to automatically and formally deal with these delicate affairs based on an accurate model of the CPU.

For the case of AES, there are efficient software implementations that avoid the use of data caches by bit-slicing, and achieve competitive performance by relying on SIMD (Single Instruction, Multiple Data) support [27]. Furthermore, a model for statistical estimation of the effectiveness of AES cache attacks based on sizes of cache lines and lookup tables has been presented in [43]. For programs beyond AES that are not efficiently implementable using bit-slicing, our analysis technique allows to derive formal assertions about their leakage, based on the actual program semantics and accurate models of the CPU.

Technically, our work builds on methods from quantitative information-flow analysis (QIF) [18], where the automation by reduction to counting problems appears in [13, 26, 34, 36], and the connection to abstract interpretation in [31]. Prior applications of QIF to side-channels in cryptosystems [29,30,32] are limited to stateless systems. For the analysis of caches, we rely on the abstract domains from [23] and their implementation in the AbsInt TimingExplorer [1]. Finally, our work goes beyond language-based approaches that consider caching [12, 25] in that we rely on more realistic models of caches and aim for more permissive, quantitative guarantees.

Finally, leakage-resilient cryptography (see e.g. [22, 35]) aims at providing cryptographic primitives that remain secure even if their implementation leaks information. As future work, we plan to investigate whether the bounds derived by our technique can be used for justifying the leakage models underlying those approaches.

8 Conclusions and future work

We have shown that cache side-channels can be automatically quantified. For this, we have leveraged powerful tools for static cache analysis and quantitative information-flow analysis, which we connect using novel techniques for counting the concretizations of abstract cache states. We have demonstrated the practicality of our approach by deriving information-theoretic security guarantees for an off-the-shelf implementation of 128-bit AES (with and without a commonly suggested countermeasure) on a realistic model of an embedded CPU.

Our prime targets for future work are to develop abstract cache domains that enable the derivation of bounds that hold for an arbitrary number of executions of the victim process, and to extend our quantification to cater for alternative adversary models, such as asynchronous, trace-based, and timing-based. Progress along these lines will enable the automatic derivation of formal, quantitative security guarantees for a larger class of relevant attack scenarios.

Acknowledgments The authors would like to thank Reinhard Wilhelm for pointing us to his group's work on static cache analysis, Martin Kaiser and the AbsInt team for their friendly support, Stephan Max for his help in exploring the AbsInt TimingExplorer, and Daniel Bernstein, Pierre Ganty, and Andrew Myers for helpful feedback and suggestions.

References

1. AbsInt aiT Worst-Case Execution Time Analyzers. `http://www.absint.com/a3/`.
2. Arm7tdmi datasheet. `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0234b/DDI0234.pdf`.
3. GNU ARM. `http://www.gnuarm.com/`.
4. Intel Advanced Encryption Standard (AES) Instructions Set. `http://software.intel.com/file/24917`.
5. PolarSSL. `http://polarssl.org/`.
6. The Haskell Programming Language. `http://www.haskell.org/`.
7. The Open Source toolkit for SSL/TSL. `http://www.openssl.org/`.
8. O. Aciiçmez, B. B. Brumley, and P. Grabher. New results on instruction cache attacks. In *Proc. 12th International Workshop of Cryptographic Hardware and Embedded Systems (CHES 2010)*, volume 6225 of *Lecture Notes in Computer Science*, pages 110–124. Springer, 2010.
9. O. Aciiçmez and Çetin Kaya Koç. Trace-Driven Cache Attacks on AES (Short Paper). In *8th International Conference on Information and Communications Security (ICICS 2006)*, volume 4307 of *Lecture Notes in Computer Science*, pages 112–121. Springer, 2006.
10. O. Aciiçmez, Çetin Kaya Koç, and J.-P. Seifert. Predicting secret keys via branch prediction. In *The Cryptographers' Track at the RSA Conference (CT-RSA '07)*, volume 4377 of *Lecture Notes in Computer Science*, pages 225–242. Springer, 2007.
11. O. Aciiçmez, W. Schindler, and Çetin Kaya Koç. Cache based remote timing attack on the aes. In *The Cryptographers' Track at the RSA Conference 2007 (CT-RSA 2007)*, volume 4377 of *Lecture Notes in Computer Science*, pages 271–286. Springer, 2007.
12. J. Agat. Transforming out Timing Leaks. In *Proc. 27th ACM Symposium on Principles of Programming Languages (POPL 2000)*, pages 40–53. ACM, 2000.
13. M. Backes, B. Köpf, and A. Rybalchenko. Automatic Discovery and Quantification of Information Leaks. In *Proc. 30th IEEE Symposium on Security and Privacy (S&P '09)*, pages 141–153. IEEE, 2009.
14. D. J. Bernstein. Cache-timing attacks on aes. Technical report, 2005.
15. C. Braun, K. Chatzikokolakis, and C. Palamidessi. Quantitative notions of leakage for one-try attacks. In *Proc. of the 25th Conference on Mathematical Foundations of Programming Semantics (MFPS 2009)*, volume 249 of *ENTCS*, pages 75–91. Elsevier, 2009.
16. D. Brumley and D. Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.
17. C. Cachin. Entropy Measures and Unconditional Security in Cryptography. Dissertation No. 12187. ETH Zürich, 1997.
18. D. Clark, S. Hunt, and P. Malacaria. A static analysis for quantifying information flow in a simple imperative language. *Journal of Computer Security*, 15(3):321–371, 2007.
19. B. Coppens, I. Verbauwhede, K. D. Bosschere, and B. D. Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *Proc. 2009 IEEE Symposium on Security and Privacy (Oakland 2009)*, pages 45–60. IEEE Computer Society, 2009.

20. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *4th ACM Symposium on Pinciples of Programming Languages (POPL '77)*, pages 238–252.
21. P. Cousot, R. Cousot, and L. Mauborgne. A scalable segmented decision tree abstract domain. In Z. Manna and D. Peled, editors, *Pnueli Festschrift*, volume 6200 of *Lecture Notes in Computer Science*, pages 72–95, Heidelberg, 2010. Springer-Verlag.
22. S. Dziembowski and K. Pietrzak. Leakage-Resilient Cryptography. In *Proc. 49th IEEE Symposium Symposium on Foundations of Computer Science (FOCS 2008)*, pages 293–302. IEEE Computer Society, 2008.
23. C. Ferdinand, F. Martin, R. Wilhelm, and M. Alt. Cache behavior prediction by abstract interpretation. *Science of Computer Programming*, 35(2):163 – 189, 1999.
24. D. Gullasch, E. Bangerter, and S. Krenn. Cache games - bringing access-based cache attacks on aes to practice. In *Proc. 2011 IEEE Symposium on Security and Privacy (Oakland 2011)*, pages 490–505. IEEE Computer Society, 2011.
25. D. Hedin and D. Sands. Timing Aware Information Flow Security for a JavaCard-like Bytecode. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 141(1):163–182, 2005.
26. J. Heusser and P. Malacaria. Quantifying information leaks in software. In *26th Annual Computer Security Applications Conference, (ACSAC '10)*, pages 261–269. ACM, 2010.
27. E. Käsper and P. Schwabe. Faster and timing-attack resistant aes-gcm. In *Proc. 11th International Workshop on Cryptographic Hardware and Embedded Systems (CHES '11)*, volume 5747 of *LNCS*, pages 1–17, 2009.
28. P. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Proc. Advances in Cryptology (CRYPTO 1996)*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996.
29. B. Köpf and D. Basin. An Information-Theoretic Model for Adaptive Side-Channel Attacks. In *Proc. 14th ACM Conference on Computer and Communication Security (CCS 2007)*, pages 286–296. ACM, 2007.
30. B. Köpf and M. Dürmuth. A Provably Secure And Efficient Countermeasure Against Timing Attacks. In *Proc. 22nd IEEE Computer Security Foundations Symposium (CSF 2009)*, pages 324–335. IEEE Computer Society, 2009.
31. B. Köpf and A. Rybalchenko. Approximation and Randomization for Quantitative Information-Flow Analysis. In *Proc. 23rd IEEE Computer Security Foundations Symposium (CSF 2010)*, pages 3–14. IEEE Computer Society, 2010.
32. B. Köpf and G. Smith. Vulnerability Bounds and Leakage Resilience of Blinded Cryptography under Timing Attacks. In *Proc. 23rd IEEE Computer Security Foundations Symposium (CSF 2010)*, pages 44–56. IEEE Computer Society, 2010.
33. L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzers. In M. Sagiv, editor, *European Symposium on Programming (ESOP'05)*, volume 3444 of *Lecture Notes in Computer Science*, pages 5–20. Springer-Verlag, 2005.
34. Z. Meng and G. Smith. Calculating bounds on information leakage using two-bit patterns. In *6th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS '11)*. ACM, 2011.
35. S. Micali and L. Reyzin. Physically Observable Cryptography (Extended Abstract). In *Proc. First Theory of Cryptography Conference (TCC 2004)*, volume 2951 of *Lecture Notes in Computer Science*, pages 278–296. Springer, 2004.
36. J. Newsome, S. McCamant, and D. Song. Measuring channel capacity to distinguish undue influence. In *Proc. Workshop on Programming Languages and Analysis for Security (PLAS '09)*, pages 73–85. ACM, 2009.

37. D. A. Osvik, A. Shamir, and E. Tromer. Cache Attacks and Countermeasures: the Case of AES. In *Proc. RSA Conference 2006, Cryptographers' Track*, volume 3860 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2006.
38. D. Page. Theoretical use of cache memory as a cryptanalytic side-channel, 2002.
39. C. Percival. Cache missing for fun and profit. In *Proc. of BSDCan 2005*, 2005.
40. J. Reineke. Caches in WCET Analysis. PhD thesis, Saarland University, 2008.
41. A. Rényi. On measures of entropy and information. In *Proceedings of the 4th Berkeley Symposium on Mathematics, Statistics and Probability 1960*, pages 547–561, 1961.
42. G. Smith. On the Foundations of Quantitative Information Flow. In *Proc. 13th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS 2009)*, volume 5504 of *Lecture Notes in Computer Science*, pages 288–302. Springer, 2009.
43. K. Tiri, O. Aciiçmez, M. Neve, and F. Andersen. An Analytical Model for Time-Driven Cache Attacks. In *14th International Workshop on Fast Software Encryption (FSE '07)*, volume 4593 of *Lecture Notes in Computer Science*, pages 399–413. Springer, 2007.
44. Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *34th International Symposium on Computer Architecture (ISCA 2007)*, pages 494–505. ACM, 2007.

A Omitted Proofs

The concrete cache updating function for a set t for access of a location m as defined in [23] defines a new set t' as follows

$$\mathcal{U}(t,m) = \begin{cases} t'(1) = m, \\ t'(i) = t(i-1) \mid i = 2 \ldots h, \\ t'(i) = t(i) \mid i = h+1 \ldots A; & \text{if } \exists h : t(h) = m \\ \\ t'(1) = m, \\ t'(i) = t(i-1) \text{ for } i = 2 \ldots A; & \text{otherwise} \end{cases}$$

This can be generalized for a list $\langle m_1, \ldots, m_k \rangle$ of locations:

$$\mathcal{U}(t, \langle m_1, \ldots, m_k \rangle) = \mathcal{U}(\mathcal{U}(\ldots \mathcal{U}(t, m_1) \ldots), m_k)$$

Lemma 2. *Let Φ be the following property over cache sets t:*

$$\Phi(t) : t(k) = \emptyset \Rightarrow \forall\ k' > k\ t(k') = \emptyset.$$

Then if an initial concrete state t satisfies Φ, then the final concrete state after the update of a sequence of memory references by means of the cache update function $\mathcal{U}$ also respects Φ.

Proof. We consider the case where the cache consists of only one cache set (the generalization is similar). We apply induction on the list of memory references for:

$$t' = \mathcal{U}(t, \langle m_1, \ldots, m_k \rangle)$$

If the list is empty, the claim follows trivially. Otherwise, there are two cases:

$\exists h : t(h) = m$: As m is in the cache, it is shifted to the first position and all other locations are placed afterwards. Therefore Φ holds on the resulting cache.

otherwise: As m is not in the cache, it is inserted in the first positions and the existing locations are placed afterwards. Therefore Φ holds on the resulting cache.

B Excerpt final cache state of AES with preloading (16KB, 4-Way, LRU, 32B line size)

```
must = [
[["0x1f000"],[],[],[]],  [["0x1f020"],[],[],[]],
[["0x1f040"],[],[],[]],  [["0x1f060"],[],[],[]],
[["0x1f080"],[],[],[]],  [["0x1f0a0"],[],[],[]],
[[],["0x1f0c0","0x280c0"],[],[]],  [[],["0x1f0e0","0x280e0"],[],[]],
[[],["0x1f100","0x28100"],[],[]],  [[],["0x1f120","0x28120"],[],[]],
[[],["0x1f140","0x28140"],[],[]],  [["0x28160"],["0x1f160"],[],[]],
[["0x1f180"],[],[],[]],  [["0x1f1a0"],[],[],[]],
[["0x1b1c0"],[],["0x191c0"],[]],  [[],["0x151e0","0x1f1e0"],[],[]],
[[],[],[],["0x13200"]],  [],
[[],["0x9240","0x1f240"],[],[]],   [[],["0x1e260","0x1f260"],[],[]],
...]


may = [
[["0x1f000"],[],[],[]],  [["0x1f020"],[],[],[]],
[["0x1f040"],[],[],[]],  [["0x1f060"],[],[],[]],
[["0x1f080"],[],[],[]],  ["0x1f0a0"],[],[],[]],
["0x1f0c0","0x280c0"],[],[],[]],   ["0x1f0e0","0x280e0"],[],[],[]],
[["0x1f100","0x28100"],[],[],[]],  [["0x1f120","0x28120"],[],[],[]],
[["0x1f140","0x28140"],[],[],[]],  [["0x28160"],["0x1f160"],[],[]],
[["0x1f180"],[],[],[]],  [["0x1f1a0"],[],[],[]],
[["0x1b1c0"],["0x151c0","0x171c0","0x191c0","0x1f1c0"],[],[]],   [["0x151e0","0x1f1e0"],[],[],[]],
[["0x13200","0x1f200"],["0x11200"],[],[]],  [["0x11220","0x1f220"],["0xf220"],["0x9220","0xb220","0xd220"],[]],
[["0x9240","0x1f240"],[],[],[]],  [["0x1e260","0x1f260"],[],[],[]],
...]
```

C AES Transformation with Partitioning (Snippet)

In the encrypting process of the AES code as implemented in [5], access to the look-up tables are performed for example in the various AES forward rounds. For encryption for example, the forward rounds are initially contained in a loop,

```
for(j = (nr >> 1) - 1; j > 0; j--) {
        AES_FROUND(Y0, Y1, Y2, Y3, X0, X1, X2, X3);
        AES_FROUND(X0, X1, X2, X3, Y0, Y1, Y2, Y3);
}
```

where for a 128 bit key `nr=10`. The AES forward round is defined as:

```
#define AES_FROUND(X0,X1,X2,X3,Y0,Y1,Y2,Y3)     \
{                                               \
    X0 = *RK++ ^ FT0[ ( Y0       ) & 0xFF ] ^   \
                 FT1[ ( Y1 >>  8 ) & 0xFF ] ^   \
                 FT2[ ( Y2 >> 16 ) & 0xFF ] ^   \
                 FT3[ ( Y3 >> 24 ) & 0xFF ];

                 ...
}
```

For partitioning (see 6.2) we have defined a function `TEST_INDEX` as follows:

```
#define TEST_INDEX(i,a,v) { \
                if(i<4) {  \
                  v = a[i]; \
                } \
                else{ \
                      if(i<12){ \
                                  v = a[i]; \
                       } \
...
                              if(i<252){ \
                                        v = a[i];        \
                              }\
                              else {\
                                      v = a[i];\
                                      }
...
}
```

This code checks the range of the index i within a partition of the possible index range of table a. In this case i ranges from 0 to 255, partitioned in 33 sub-ranges of length at most 8 corresponding to the sets associated to the table blocks within a given sub-range. Then we have changed the original look-up by the following semantically equivalent code.

```
#define AES_FROUND(X0,X1,X2,X3,Y0,Y1,Y2,Y3) \
{ \
/* temporary variable */ \
unsigned long t; \
\
TEST_INDEX(Y0,FT0,t); \
X0 = *RK++ ^ t; \
TEST_INDEX(Y1 >> 8,FT1,t); \
X0 ^= t; \
TEST_INDEX(Y2 >> 16,FT2,t); \
X0 ^= t; \
TEST_INDEX(Y3 >> 24,FT3,t); \
X0 ^= t;
...}
```