Quantifying the Information Leak in Cache Attacks through Symbolic Execution

Sudipta Chattopadhyay
Saarland University
Moritz Beck
Saarland University
Ahmed Rezine
Linköping University
Andreas Zeller
Saarland University

Abstract—Cache timing attacks allow attackers to infer the properties of a secret execution by observing cache hits and misses. But how much information can actually leak through such attacks? For a given program, a cache model, and an input, our CHALICE framework leverages symbolic execution to compute the amount of information that can possibly leak through cache attacks. At the core of CHALICE is a novel approach to quantify information leak that can highlight critical cache side-channel leaks on arbitrary binary code. In our evaluation on real-world programs from OpenSSL and Linux GDK libraries, CHALICE effectively quantifies information leaks: For an AES-128 implementation on Linux, for instance, CHALICE finds that a cache attack can leak as much as 127 out of 128 bits of the encryption key.

1. Introduction

Cache timing attacks [1] are among the best known side channel attacks to determine secret features of a program execution without knowing its input or output. The general idea of a timing attack is to observe, for a known program, a timing of cache hits and misses, and then to use this timing to determine or constrain features of the program execution, including secret data that is being processed.

The precise nature of the information that can leak through such attacks depends on the cache and its features, as well as the program and its features. Consequently, given a model of the cache and a program run, it is possible to analyze which and how much information would leak through a cache attack. This is what we do in this paper. Given a program execution and a cache model, our CHALICE approach automatically determines which bits of the input would actually leak through a potential cache attack.

As an example, consider an implementation of the popular AES encryption algorithm. Given an input and an encryption key (say, 128 bits for AES-128), CHALICE can determine which and how many of the bits of the key would leak if the execution were subject to a cache attack. To this end, CHALICE uses a novel symbolic execution over the given concrete input. During symbolic execution, CHALICE derives symbolic timings of cache hits and misses; these then again reveal under which circumstances individual bits of encryption key may leak.

The reason why CHALICE works is that the timings of cache hits and misses are not uniformly distributed; and therefore, some specific timings may reveal more information than others. Figure [1] demonstrates the execution of an AES-128 implementation [1] for a fixed input and 256,000 different keys, inducing between 213 and 279 cache misses. We see that the distribution of cache misses is essentially Gaussian; if the number of cache misses is average, there are up to 13,850 keys which induce this very cache timing. If we have an extreme cache timing with 213 misses (the minimum) or 279 misses (the maximum), then there are only 2 keys that induce this very timing. CHALICE can determine that for these keys, 90 of 128 bits would leak if the execution were subjected to a cache attack, which in practice would mean that the remaining 38 bits could be guessed through brute force—whereas other “average” keys would be much more robust. For each key and input, CHALICE can precisely predict which bits would leak, allowing its users to determine and find the best alternative.

It is this precision of its symbolic analysis that sets CHALICE apart from the state of the art. Existing works [10] [21] use static analysis alone to provide an upper bound on the potential number of different observations that an attacker can make. This upper bound, however, does not suffice to choose between alternatives, as it ignores the distribution of inputs: It is possible that certain inputs may leak substantially more information than others. Not only that such an upper bound might be imprecise, it is also incapable to identify inputs that exhibits substantial information leakage through side channels. Given a set of inputs (typically as part of a testing pipeline), CHALICE

1. In the best of all worlds, one might have an implementation of every critical algorithm, such as an encryption routine, to have a uniform distribution over cache misses. But neither does such implementations exist that would be efficient, nor do we know whether such implementations can exist; and replacing a well-studied algorithm like AES by some other algorithm with uniform distribution may induce other, yet unknown risks.
2. Overview

In this section, we convey the key insight behind our approach through examples. In particular, we illustrate how CHALICE is used to quantify information leak from the execution trace of a program.

Motivating Example

Let us assume that our system contains a direct-mapped cache of size 512 bytes. Figures 2(a)-(c) show different code fragments executed in the system. For the sake of clarity, we use both assembly-level and source-level syntaxes. However, our framework takes a binary code as input, in order to accurately capture the memory behaviour of a program. For simplicity in the example, we assume that conditional checks do not involve any access to cache (i.e. $k$ is assigned to a register). The mapping of different variables into the cache is shown in Figure 2(d). Let us assume that the code fragments of Figures 2(a)-(c) are executed with some arbitrary (and unknown) value of $k$. Broadly, CHALICE answers the following question: Provided only the cache performance (e.g. cache hit/miss sequence) from such executions, how much information about the sensitive input $k$ is leaked?

The cache performance induces a partition on the program input space. Let us capture the cache performance via a sequence of hits ($h$) and misses ($m$). In Figure 2(a), for all values of $k$ between 0 and 127, we observe two cache misses due to the first two memory accesses, $p[k]$ and $q[255-k]$, respectively. The second access to $p[k]$ is a cache hit, for $k \in [1,127]$. However, if $k = 0$, the content of $p[k]$ will be replaced by $q[255-k]$, resulting in a cache miss at the second access of $p[k]$. For $k \in [128,255]$, $p[k]$ is never replaced once it is loaded into the cache. Therefore, the second access to $p[k]$ is a cache hit for $k \in [128,255]$. In other words, we observe the sequence of cache hits and misses to induce the following partition on the input space: $k \in [0,255] \land k \mod 2 = 0$ (hit/miss sequence = $\langle m,m,m \rangle$ and $k \in [0,255] \land k \mod 2 = 1$ (hit/miss sequence = $\langle m,m,h \rangle$).

Let us consider that a cache miss occurs at $q[255-k]$, for $k \leq 127$. Then, the respective partitions of the input space with respect to cache hit/miss sequence (reg1, reg2 represent registers) are shown in Figure 2(b) and Figure 2(c). A similar exercise for the cache miss at $q[k-128]$ for $k > 127$ is shown in Figure 2(c). Let us assume that the code fragments of Figures 2(a)-(c) are executed with some arbitrary (and unknown) value of $k$. Broadly, CHALICE answers the following question: Provided only the cache performance (e.g. cache hit/miss sequence) from such executions, how much information about the sensitive input $k$ is leaked?

The cache performance induces a partition on the program input space. Let us capture the cache performance via a sequence of hits ($h$) and misses ($m$). In Figure 2(a), for all values of $k$ between 0 and 127, we observe two cache misses due to the first two memory accesses, $p[k]$ and $q[255-k]$, respectively. The second access to $p[k]$ is a cache hit, for $k \in [1,127]$. However, if $k = 0$, the content of $p[k]$ will be replaced by $q[255-k]$, resulting in a cache miss at the second access of $p[k]$. For $k \in [128,255]$, $p[k]$ is never replaced once it is loaded into the cache. Therefore, the second access to $p[k]$ is a cache hit for $k \in [128,255]$. In other words, we observe the sequence of cache hits and misses to induce the following partition on the input space: $k \in [0,255] \land k \mod 2 = 0$ (hit/miss sequence = $\langle m,m,m \rangle$ and $k \in [0,255] \land k \mod 2 = 1$ (hit/miss sequence = $\langle m,m,h \rangle$).

**Key observation.** In this work, we stress the importance of quantifying information leaks from execution traces and not from the static representation of a program. To illustrate this, consider the input partitions created for code fragments...
in Figures 2(a)-(b). We emphasize that observing the cache hit/miss sequence \(⟨m, m, m⟩\), from an execution of the code fragment in Figure 2(a), results in complete disclosure of sensitive input \(k\). On the contrary, observing the sequence \(⟨m, m, m⟩\), from an execution of the code fragment in Figure 2(b), will only reveal the information that \(k \text{ is odd}\). Such information still demands a probability of \(\frac{1}{2}\) in order to correctly guess \(k\) at first attempt. This is in contrast to accurately guessing the correct value of \(k\) at first attempt (as happened through the sequence \(⟨m, m, m⟩\) for Figure 2(a)). In order to fix the cache side-channel leak in Figure 2(a), we can reorder the code as shown in Figure 2(c).

**Limitations of static analysis.** Existing works in static analysis and verification have aimed at quantifying side channel leaks [16], [21] and verifying constant-time implementations [7], [10]. These works correlate the number of possible observations (by an attacker) with the number of bits leaked through a side channel. We believe this view can be dangerous. Indeed, both code fragments in Figures 2(a)-(b) have exactly two possible cache hit/miss sequences, for arbitrary values of \(k\). Therefore, approaches based on static analysis [16], [21] will consider these two code fragments *equivalent* in terms of cache side-channel leakage. As a result, statically analyzing a program will not reveal crucial information leak scenarios, such as the execution of code fragment in Figure 2(a) with \(k = 0\). Techniques based on verifying that programs execute in constant time typically check that memory accesses do not depend on sensitive inputs. Yet, most implementations do not execute in constant time. Besides, programs such as in Figure 2(c) have accesses that may depend on sensitive inputs without leaking information about it to a cache-performance observer. Therefore, we not only check the dependency between accessed memory address and program inputs, but we also accurately track the information flow through cache performance.

**Can we use dynamic tainting?** In the preceding paragraph, we state the importance of dynamically tracking sensitive information flow through cache performance. Approaches based on dynamic tainting [13] can accomplish the task to detect information leak through standard functional outputs. However, such approaches fail to detect information leak through software non-functional outputs, such as cache performance, among others. Our methodology targets this angle of information leak detection by building a relationship between sensitive inputs and observed cache performance. In order to establish such a relationship, we leverage on symbolic analysis and constraint solving.

**Limitations of side-channel vulnerability metrics.** In contrast to existing works on measuring cache side channel leakage [15], we do not aim to check the strength of an attacker to *observe information through side channel*. Although promising, this work [15] fails to detect the information flow between sensitive inputs and observed performance. As a result, the side-channel vulnerability metric can only quantify how well an attacker can retrieve information from a system, but, does not highlight the information potentially leaked to the attacker. Of course, we believe our work is complementary to the metrics proposed in [15] and CHALICE could be combined with such metrics to build more advanced metrics for measuring side channel leakage. Such metrics could consider both information leaked by the system as well as the information that could be retrieved by an attacker.

**The usage of CHALICE.** CHALICE is aimed to be used for validating security properties of software. Given a test suite (i.e. a set of concrete test inputs) for the software, CHALICE is used to quantify the information leaked for each possible observation obtained from this test suite. This is possible, as the observation by an attacker (e.g. number of cache miss) corresponds to a (set of) test inputs and CHALICE presents how much can be deduced about such inputs from the respective observation. In other words, our framework CHALICE fits the role of a test oracle [9] in the software validation process. For instance, if CHALICE reports substantial information leakage, the test inputs leading to the respective observation should be avoided (e.g. avoiding a “weak” encryption key) or the candidate program needs to be restructured to avoid such information leak. The generation of an effective and optimized test suite, in order to detect cache side channels, is an open problem. However, CHALICE can be instantiated to generate a witness for each possible observations made by an attacker. The set of all these witnesses forms a concise test suite and our proposed method in CHALICE can quantify information leak for each element in such a test suite. In this paper, we only focus on the quantification of information leak in a single test execution and not on the generation of a test suite.

**CHALICE should not** be used for verifying the absence of cache side-channel leakage. Implementations that must adhere to zero-leakage, may leverage on CHALICE during the early design, specifically to discover the severity of potential cache side-channel leaks and the program locations exhibiting such leaks. Nevertheless, CHALICE is aimed for testing arbitrary software and we envision that such a strategy becomes an integral component of software testing pipeline in the future.

**How CHALICE works.** Let us assume that we execute the code in Figure 2(a) with some input \(I \in [0, 255]\) and observed the trace \(t_I \equiv ⟨m, m, m⟩\). *Given only the observation* \(t_I\), CHALICE *quantifies how much information about program input* \(I\) *is leaked.* CHALICE symbolically executes the program and it tracks all memory accesses dependent on the sensitive input \(k\). For each explored path, CHALICE constructs a symbolic cache model, which accurately encodes all possible cache hit/miss sequences for the respective path. In this example, CHALICE constructs \(Γ(0 \leq k \leq 127)\) and \(Γ(128 \leq k \leq 255)\), which encode all cache hit/miss sequences for inputs satisfying \(0 \leq k \leq 127\) and \(128 \leq k \leq 255\), respectively. Let us consider the path explored for inputs \(k \in [0, 127]\). While exploring the path, we record a sequence of symbolic memory addresses...
is unsatisfiable. This is because only \( k \) the value of \( q \) cannot occur for any inputs the preceding paragraph, we also know that this observation cannot for any inputs \( k \in [128, 255] \). Therefore, the value of \( k \) must result in Constraint (2) satisfiable. Constraint (2) is unsatisfiable if we restrict the value of \( k \) between 1 and 127. This happens based on the fact that only \( p[0] \) is mapped to the same cache line as \( q[255] \) (cf. Figure 2(d)). As a result, CHALICE reports 255 (127 for the if branch and 128 for the else branch in Figure 2(a)) values being leaked for the observation \((m, m, m)\). In other words, CHALICE accurately reports the information leak (i.e. \( k = 0 \)) for the observation \((m, m, m)\).

3. Framework

In the following, we formally introduce the problem statement and provide an outline of our overall approach to solve this problem.

3.1. Foundation

Threat model. Side-channel attacks are broadly classified into synchronous and asynchronous attacks \([25]\). In synchronous attack, an attacker can trigger the processing of known inputs (e.g. a plain-text or a cipher-text for encryption routines), whereas such a phenomenon is not possible for asynchronous attacks. Synchronous attacks are clearly easier to perform, since the attacker does not need to compute the start and end of the targeted routine under attack. For instance, in synchronous attack, the attacker can trigger encryption of known plaintext messages and observe the encryption-timing \([11]\). Since CHALICE is a software validation tool with the aim of producing side-channel resistant implementations, we assume the presence of a strong attacker in this paper. Therefore, we consider the attacker can request and observe the execution (e.g. number of cache miss) of the targeted routine. We also assume that the attacker can execute arbitrary user-level code in the same processor running the targeted routine. This allows the attacker to flush the cache before the targeted routine starts execution and therefore, reduce the external noise in the observation. The attacker, however, is incapable to access the address space of the target routine.

Notations. The execution of program \( P \) on input \( I \) results in an execution trace \( t_{ij} \). \( t_{ij} \) is a sequence over the alphabet \( \Sigma = \{ h, m \} \) where \( h \) (respectively, \( m \)) represents a cache hit (respectively, cache miss). Our proposed method in CHALICE quantifies the information leaked through \( t_{ij} \). We capture this quantification via \( L(t_{ij}) \). We assess the information leakage with respect to an observer. An observer is a mapping \( O : \Sigma^* \rightarrow \mathbb{D} \) where \( \mathbb{D} \) is a countable set. For instance, an observer \( O : \Sigma^* \rightarrow \mathbb{N} \) can count the number of misses and will associate both sequences \((m, h, m, h, h) \) and \((m, m, h, h, h) \) to 2. It will therefore not differentiate them. The most precise observer would be the identity mapping on \( \Sigma^* \). However, an observer that tracks prefixes of some fixed lengths (for example 2) would be enough to differentiate the two aforementioned sequences.

We use the variable \( miss_i \) to capture whether or not the \( i \)-th memory access was a cache miss during execution. The observation by an attacker, over the execution for an arbitrary input and according to the observer model \( O \), is considered via the observation constraint \( \Phi_O \).
\( \Phi_O \) is a symbolic constraint over the set of variables \( \{miss_1, miss_2, \ldots, miss_n\} \), where \( n \) is the total number of memory accesses during an execution. For instance, \( \Phi_O \equiv (\sum_{i=1}^{n} miss_i = 100) \) accurately captures that the attacker observes 100 cache misses in an execution manifesting \( n \) memory accesses. For the sake of formulation, we use \( \Phi_{O,e} \) to define a projection of \( \Phi_O \) on an arbitrary program path \( e \). In particular, \( \Phi_{O,e} \) captures the observation constraint if program path \( e \) is executed. Given only \( \Phi_O \) to be observed by an attacker, CHALICE quantifies how much information about the respective program input is leaked.

The central idea of our information leak detection is to capture the cache behaviour via symbolic constraints. Let us consider a set of inputs \( I \) that exercise the same execution path with \( n \) memory accesses. We use \( \Gamma(I) \) to accurately encode all possible combinations of values of variables \( \{miss_1, miss_2, \ldots, miss_n\} \). Therefore, if \( \Gamma(I) \land \Phi_O \) is unsatisfiable, we can deduce that the respective observation \( \Phi_O \) did not occur for any input \( I \in \mathbb{I} \).

We now describe how \( L(t_I) \) is computed based on the notations and the intuition mentioned in the preceding.

### 3.2. Quantifying Information Leak in Execution

Figure 3.1 provides an outline of our entire framework. We symbolically execute a program \( P \) compute the path condition \([15]\) for each explored path. Such a path condition symbolically encodes all program inputs for which the respective program path was followed. Our symbolic execution based framework tracks all memory accesses on a taken path and therefore, enables us to characterize, for all symbolic arguments satisfying the path condition, the set of all associated cache behaviors.

Recall that we use \( \Gamma(I) \) to capture possible cache hit/miss sequences in an execution path, which was activated by a set of inputs \( I \). In an abuse of notation, we capture set of inputs \( I \) via path conditions. For instance, in Figure 3.1(a), we use \( \Gamma(0 \leq k \leq 127) \) to encode all possible cache hit/miss sequences for inputs activating the \( I_f \) branch.

For an arbitrary execution path, let us consider \( pc \) be the path condition. Along this path, we record each memory access and we consider its cache behaviour via variable \( miss_i, miss_{is} \) is set to 1 (resp. 0) if and only if the \( i \)-th memory access along the path encounters a cache miss (hit). Given \( n \) to be the total number of memory accesses along the path, we formulate \( \Gamma(pc) \) to bound the value of \( \{miss_1, miss_2, \ldots, miss_n\} \). In particular, any solution of \( \Gamma(pc) \land (miss_i = 1) \) captures a concrete input \( I \Rightarrow pc \) and such an input \( I \) leads to an execution where the \( i \)-th memory access is a cache miss. Therefore, if an observation \( \Phi_O \) happens to be for input \( I \Rightarrow pc \), \( \Gamma(pc) \land \Phi_O \) is always satisfiable.

We capture the information leak through execution trace \( t_I \) as follows:

\[
L(t_I) = 2^N - \bigvee_{e \in Path} (\Gamma(pc_e) \land \Phi_{O,e})|_{sol}
\]

where \( N \) is size of program input (in bits), \( \Phi_{O,e} \) is the projection of the observation constraint on path \( e \). Path is the set of all program paths and \( pc_e \) is the path condition for program path \( e \). For input segment \( I \), \( |X|_{sol} \) captures the number of solutions satisfied by predicate \( X \). It is worthwhile to note that \( \bigvee_{e \in Path} (\Gamma(pc_e) \land \Phi_{O,e})|_{sol} \) accurately captures the number of program inputs that does not exhibit the observation satisfied by \( \Phi_O \). In other words, Equation (5) quantifies the number of program inputs that does not exhibit the observation, as captured by \( \Phi_O \). Hence, if the attacker observes \( \Phi_O \), she can deduce as many as \( L(t_I) \) inputs were impossible for the respective observation.

In practice, however, computing the exact value of \( L(t_I) \) might be infeasible, as it might require the enumeration of all solutions. In order to control such enumeration, we generate predicates on input variables. In particular, we sample an \( N \)-bit input into \( K \) equal segments, resulting in input segments of length \( \frac{N}{K} \). Subsequently, we constrain the search space of the solver by restricting the value of each such input segment to any possible value, that is, pointing to a value in the set \( \{0, 1, \ldots, 2^{\frac{N}{K}} - 1\} \). For instance, let us assume \( x \) is the program input and \( x_i \) captures the \( i \)-th input segment. A predicate \( \pi \equiv (x_i = 0) \) will guide the solver to search for a solution only in the input space where the \( i \)-th input segment is 0. Since, we have \( K \) different segments, we generate a total of \( K \cdot 2^{\frac{N}{K}} \) different predicates. For each such predicate \( \pi \), we record information leak if the following constraint is unsatisfiable:

\[
\bigvee_{e \in Path} (\Gamma(pc_e) \land \Phi_{O,e} \land \pi)
\]

Concretely, if Constraint (5) is unsatisfiable, we can accurately record that input \( I \), which leads to observation \( \Phi_O \) along some program path, satisfies the predicate \( \neg \pi \). An appealing feature of this process is that all \( K \cdot 2^{\frac{N}{K}} \) predicates can be generated independently and therefore, the unsatisfiability check of Constraint (5) can be performed in parallel for different predicates.

Let us assume, \( U_1, U_2, \ldots, U_K \) are the number of unsatisfiable solutions reported for each of the \( K \) input segments respectively. Therefore, we can estimate a lower bound on \( L(t_I) \) from these unsatisfiability checks as follows:

\[
L(t_I) \geq 2^N - \prod_{1 \leq i \leq K} \left( 2^{\frac{N}{K}} - U_i \right)
\]
Due to the classic path explosion problem in symbolic execution, it is possible that only a subset of paths $P' \subseteq P$ can be explored within a given time budget. In such cases, we can quantify $\mathcal{L}(t_i)$ as follows:

\[
\mathcal{L}(t_i) = 2^N - \left| \bigcup_{e \in \text{Path}} (\Gamma(pc_e) \land \Phi_{O,e}) \right|_{sol}
\]

\[
= | \bigcup_{e \in P} pc_e|_{sol} + | \bigcup_{e \in P \setminus P'} pc_e|_{sol} \\
- | \bigcup_{e \in P'} (\Gamma(pc_e) \land \Phi_{O,e})|_{sol} \\
- | \bigcup_{e \in P'} (\Gamma(pc_e) \land \Phi_{O,e})|_{sol} \\
\geq | \bigcup_{e \in P} pc_e|_{sol} - \bigcup_{e \in P'} (\Gamma(pc_e) \land \Phi_{O,e})|_{sol} \\
\geq | \bigcup_{e \in P} pc_e|_{sol} - \prod_{1 \leq i \leq K} \left( 2^S - U_i \right)
\]

This result follows from the fact that $\Gamma(pc_e) \land \Phi_{O,e} \Rightarrow \Gamma(pc_e) \Rightarrow pc_e$. The term $| \bigcup_{e \in P} pc_e|_{sol}$ involves only path conditions and it can be computed via model counting \[\text{[5]}\].

Finally, it is worthwhile to note that setting $K = 1$ is equivalent to enumerating all solutions as in Equation (4).

In contrast, setting $K = N$ is equivalent to checking information leak at bit-level (i.e., checking whether the value of a single bit can influence cache performance). Therefore, $K$ provides a tunable parameter for different levels of information leak detection. We have conducted evaluation for $K = 8$ and $K = N$. This means, we have checked how much information about a single byte and respectively, a single bit are leaked through observing cache performance.

In the next section, we will describe the construction of $\Gamma(pc)$ for an arbitrary path condition $pc$.

### 4. Generating Symbolic Cache Model

The technical contribution of our methodology is a symbolic model for cache behaviour – establishing a link between the program input and observed cache performance. To describe our model, we shall use the following notations throughout our discussions:

- $2^S$: The number of cache sets in the cache.
- $2^B$: The size of a cache line in bytes.
- $A$: Associativity of cache. For direct-mapped caches, $A = 1$.
- $set(r_i)$: Cache set accessed by instruction $r_i$.
- $tag(r_i)$: The tag stored in the cache for the memory block accessed by $r_i$.
- $ζ_i$: The cache state before executing instruction $r_i$ and after executing instruction $r_{i-1}$.

In the following, we will explain the different steps of generating the symbolic cache model.

#### 4.1. Intercepting Memory Requests

We symbolically execute a program $P$. During symbolic execution, we track the path condition and the sequence of memory accesses for each explored path. For instance, while symbolically exercising the $iff$ branch of Figure 2(a), we track the path condition $0 \leq k \leq 127$ and the sequence of memory addresses $\langle \&p[k], \&q[255-k], \&p[k] \rangle$. It is worthwhile to note that such memory addresses might capture symbolic expressions due to the dependency from program inputs. Concretely, we compute the path condition $pc$ and the execution trace $Ψ_{pc}$ for each explored path as follows:

\[
Ψ_{pc} \equiv \langle (r_1, σ_1), (r_2, σ_2), \ldots, (r_{n-1}, σ_{n-1}), (r_n, σ_n) \rangle
\]

where $r_i$ captures the $i$-th memory-related instruction executed along the path and $σ_i$ symbolically captures the memory address accessed by $r_i$.

#### 4.2. Modeling Symbolic Cache Access

In order to find the impact on caches, we need to find out the set of cache lines being accessed. This is accomplished by manipulating the expression $σ_i$, which was collected while executing each memory-related instruction $r_i$ (cf. Equation (8)). In particular, we formulate $set(r_i)$ as follows:

\[
set(r_i) = (σ_i \gg B) \& (2^S - 1)
\]

In Equation \[(9]\], “$\gg$” captures a bitwise-and operation and “$\&$” captures a right-shift operation.

Apart from the cache set a memory address is mapped to, we need to distinguish different memory addresses from which contents are stored into the cache. This is different from just checking the inequality between $σ_i$ values, as the memory controller groups contents of different memory addresses into a memory block and stores the memory block into a cache line. In order to distinguish different memory blocks mapped into the same cache lines, a tag is stored within each cache line. For instruction $r_i$, such a tag $tag(r_i)$ is captured as follows:

\[
tag(r_i) = (σ_i \gg (B + S))
\]

Therefore, if $tag(r_i) ≠ tag(r_j)$, we can conclude that $r_i$ and $r_j$ are accessing different memory blocks, even if $set(r_i) = set(r_j)$ holds.

It is worthwhile to note that both $set(r_i)$ and $tag(r_i)$ might be symbolic expressions due to the presence of symbolic expression $σ_i$ in Equations (8)-(10). Moreover, the computations of $set(r_i)$ and $tag(r_i)$ are independent of any cache replacement policy.

#### 4.3. Direct-mapped Caches

In this section, we assume that the cache is direct-mapped. Therefore, each cache set holds exactly one cache line. In the next section, we extend our symbolic model for set-associative caches.
Induced cache miss (causing eviction) no reuse of memory block

Figure 4. Memory-access \( r_j \) induces a cache miss at \( r_i \), if \( r_j \) accesses the same cache set as \( r_i \) and \( r_k \) does not load the block accessed by \( r_i \).

We characterize cache misses into the following two categories:

1) Cold cache misses. \( r_i \) suffers a cold miss if and only if set(\( r_i \)) has not been accessed by any previous instruction \( r \in \{ r_1, r_2, \ldots, r_{i-1} \} \).

2) Cache misses due to eviction. \( r_i \) suffers a cache miss due to eviction if and only if the last access to set(\( r_i \)) had been from an instruction \( r_j \in \{ r_1, r_2, \ldots, r_{i-1} \} \), such that tag(\( r_j \)) \( \neq \) tag(\( r_i \)).

Constraints to formulate cold cache misses. If a cache line is accessed for the first time, such an access will inevitably incur a cache miss. Let us consider that we want to check whether instruction \( r_i \) accesses a cache line for the first time during execution. In other words, we can check none of the instruction \( r \in \{ r_1, r_2, \ldots, r_{i-1} \} \) touches the same cache line as \( r_j \). Therefore \( r_i \) suffers a cold miss if and only if the following condition holds:

\[
\Theta_i^{\text{cold}} \equiv \bigwedge_{p \in [1,i)} (\text{set}(r_p) \neq \text{set}(r_i))
\]

Constraints to formulate cache evictions. In the following, we formulate a set of constraints to encode cache misses other than cold cache misses. Such cache misses occur due to the eviction of memory blocks from caches.

To illustrate different cache-miss scenarios clearly, let us consider the example shown in Figure 4. Assume that we want to check whether \( r_i \) will suffer a cache miss due to eviction. This might happen only due to instructions appearing before (in the program order) \( r_i \). Consider one such instruction \( r_j \), for some \( j \in [1, i) \). Informally, \( r_j \) is responsible for a cache miss at \( r_i \), only if the following conditions hold:

1) \( \psi_{\text{cnf}}(j, i) \): \( r_i \) and \( r_j \) access the same cache set. Therefore, we have the following constraint:

\[
\psi_{\text{cnf}}(j, i) \equiv (\text{set}(r_j) = \text{set}(r_i))
\]

2) \( \psi_{\text{dir}}(j, i) \): \( r_i \) and \( r_j \) access different memory-block tags. This can be formalized as follows:

\[
\psi_{\text{dir}}(j, i) \equiv (\text{tag}(r_j) \neq \text{tag}(r_i))
\]

3) \( \psi_{\text{eqv}}(j, i) \): There does not exist any instruction \( r_k \) where \( k \in [j+1, i) \), such that \( r_k \) accesses the same memory block as \( r_i \). It is worthwhile to note that the existence of \( r_k \) will load the memory block accessed at \( r_i \). Since \( r_k \) is executed after \( r_j \) (in program order), \( r_j \) must not be responsible for a cache miss at \( r_i \). We formulate the following constraint to capture this condition:

\[
\psi_{\text{eqv}}(j, i) \equiv \bigwedge_{k: j < k < i} (\text{set}(r_k) \neq \text{set}(r_i)) \lor (\text{set}(r_k) \neq \text{set}(r_i))
\]

Constraints (12)-(14) capture necessary and sufficient conditions for instruction \( r_j \) to replace the memory block accessed by \( r_i \) (where \( j < i \)) and the respective block not being accessed between \( r_j \) and \( r_i \). In order to check whether \( r_i \) suffers a cache miss due to eviction, we need to check Constraints (12)-(14) for any \( r \in \{ r_1, r_2, \ldots, r_{i-1} \} \). This can be captured via the following constraint:

\[
\Theta_i^{\text{emp}} \equiv \left( \bigvee_{j: 1 \leq j < i} (\psi_{\text{cnf}}(j, i) \land \psi_{\text{dir}}(j, i) \land \psi_{\text{eqv}}(j, i)) \right)
\]

\( r_i \) will not suffer a cache miss due to eviction when at least one of the Constraints (12)-(14) does not hold for all prior instructions of \( r_i \). This scenario is the negation of Constraint (15) and therefore, it is captured via \( -\Theta_i^{\text{emp}} \).

We use variable miss_{\( i \)} to capture whether instruction \( r_i \) suffers a cache miss. As discussed in the preceding paragraphs, \( r_i \) suffers a cold miss (i.e. satisfying Constraint (11)) or the memory block accessed by \( r_i \) would be evicted due to instructions executed before \( r_i \) (i.e. satisfying Constraint (15)). Using this notion, we formulate the value of miss_{\( i \)} as follows:

\[
\Theta_i^{\text{emp,dir}} \equiv (\Theta_i^{\text{emp}} \lor \Theta_i^{\text{cold}})
\]

\[
\Theta_i^{\text{mp,dir}} \equiv (\Theta_i^{\text{mp,dir}} \Rightarrow (\text{miss}_{\( i \)} = 1))
\]

\[
\Theta_i^{\text{h,dir}} \equiv (-\Theta_i^{\text{mp,dir}} \Rightarrow (\text{miss}_{\( i \)} = 0))
\]

Putting it all together. Recall that \( \Gamma(\text{pc}) \) captures the constraint system to encode the cache behaviour for all inputs \( I \Rightarrow \text{pc} \). In order to construct \( \Gamma(\text{pc}) \), we gather constraints, as derived in the preceding sections, and the path condition into \( \Gamma(\text{pc}) \) as follows:

\[
\Gamma(\text{pc}) \equiv \text{pc} \land \bigwedge_{i \in [1,n]} (\Theta_i^{\text{mp,dir}} \land \Theta_i^{\text{h,dir}})
\]

4.4. Set-associative Caches

In direct-mapped caches, exactly one memory-block tag is contained by a cache set. As a result, this memory block is replaced by any instruction accessing the same cache set, but accessing a different memory-block tag. In contrast, set-associative caches group multiple cache lines into a cache set. Therefore, evicting a memory block from a cache set might require multiple accesses to the respective cache set. The number of such accesses, as required to evict a memory block from a cache set, is determined by the relative position of the same block within the cache set. This relative position is updated during execution according to a cache
replacement policy. In this paper, we instantiate CHALICE for set-associative caches with LRU replacement policy.

From technical perspective, we need to modify Constraints \[13\]–\[18\] to reflect the working principle of set-associative caches. Before discussing such modifications, we introduce the concept of cache conflict, which is crucial for formulating the cache behaviour of set-associative caches.

**Definition 4.1. (Cache Conflict):** \(r_j\) generates a cache conflict to \(r_i\) only if executing \(r_j\) can influence the relative position of memory block accessed by \(r_i\) within the cache state \(\zeta\) (i.e. the cache state before \(r_i\) and after \(r_{i−1}\)).

In order to check whether \(r_i\) suffers a cache miss, we distinguish between the following two scenarios:

1) \(r_i\) accesses a memory block for the first time. Hence, \(r_i\) will suffer a cold cache miss.

2) The number of unique cache conflicts generated to \(r_i\) is sufficient to evict the memory block accessed by \(r_i\). Hence \(r_i\) will suffer a cache miss.

**Constraints to formulate cold cache misses.** If \(r_i\) accesses a memory block for the first time, the following condition must hold:

\[
\Theta_i^{cold} \equiv \bigwedge_{1 \leq k < i} (\text{tag}(r_k) \neq \text{tag}(r_i)) \bigwedge \text{set}(r_k) \neq \text{set}(r_i))
\]

(20)

Informally, Constraint (20) states that every instruction \(r \in \{r_1, r_2, \ldots, r_{i−1}\}\) either accesses a different cache set than \(\text{set}(r_i)\) or the accessed memory block has a different tag compared to \(\text{tag}(r_i)\). This leads to a cold cache miss at \(r_i\).

**Constraints to formulate cache evictions.** The eviction of a memory block from the cache is critically influenced by cache conflict. Therefore, we need to consider all scenarios where a cache conflict might be generated. For LRU caches, \(r_j\) generates a cache conflict to \(r_i\) (where \(j < i\)) only if the following conditions hold:

1) \(\psi_{cnf}(j, i), \psi_{dif}(j, i)\) and \(\psi_{eqv}(j, i)\) hold (cf. Constraints \[12\]–\[14\]). This ensures that \(r_j\) and \(r_i\) access the same cache set, but different memory-block tags. Additionally, \(\psi_{eqv}(j, i)\) ensures that there does not exist any instruction between \(r_j\) and \(r_i\) that loads the memory block accessed by \(r_i\).

2) Note that multiple accesses may influence the cache content in set-associative caches. Therefore, we need to distinguish unique memory accesses in order to formulate cache conflict. For instance, consider the following memory accesses in sequence: \(r_1; m_1 \rightarrow r_2; m_2 \rightarrow r_3; m_2 \rightarrow r_4; m_4\), where \(r_i\) captures the instruction and \(m_j\) captures the respective memory block being accessed. If \(m_1\) and \(m_2\) map to the same cache set in a 2-way LRU cache, \(r_4\) will still be a cache hit. This is because \(r_4\) suffers cache conflict only once, from the access to memory block \(m_2\), even though \(m_2\) has been accessed twice (at \(r_2\) and at \(r_3\)). In order to account

unique cache conflicts, we only record the cache conflict from the closest access to different memory blocks. For instance, in the preceding example, we only record cache conflict from \(r_3\) to \(r_4\). Formally, we need additional constraints to distinguish such closest accesses. We use the constraint \(\psi_{unq}(j, i)\) for such purpose. \(\psi_{unq}(j, i)\) is satisfiable if and only if there does not exist any instruction between \(r_j\) (where \(j \in \{1, i\}\)) and \(r_i\) that accesses the same memory block as \(r_j\). \(\psi_{unq}(j, i)\) is formalized as follows:

\[
\psi_{unq}(j, i) \equiv \bigwedge_{k: j < k < i} (\text{tag}(r_j) \neq \text{tag}(r_k)) \bigwedge \text{set}(r_j) \neq \text{set}(r_k)
\]

(21)

Constraints \[12\]–\[14\] and Constraint (21) accurately capture scenarios where \(r_j\) (\(j \in \{1, i\}\)) will create a unique cache conflict to \(r_i\). Let us assume \(\Psi_{i,j}^{evt}\) captures whether \(r_j\) creates a unique cache conflict to \(r_i\). Using the intuition described in the preceding paragraph, we can formulate the following constraints to set the value of \(\Psi_{i,j}^{evt}\):

\[
\Theta_{j,i}^{\text{cnf}} \equiv (\psi_{cnf}(j, i) \land \psi_{dif}(j, i) \land \psi_{eqv}(j, i) \\
\land \psi_{unq}(j, i)) \Rightarrow (\Psi_{i,j}^{evt} = 1)
\]

(22)

If any of the conditions in Constraints \[12\]–\[14\] and in Constraint (21) is not satisfied between \(r_j\) and \(r_i\), then we do not account for the cache conflict between \(r_j\) and \(r_i\), as captured by the following formulation:

\[
\Theta_{j,i}^{\text{ch}} \equiv (\lnot\psi_{cnf}(j, i) \lor \lnot\psi_{dif}(j, i) \lor \lnot\psi_{eqv}(j, i) \\
\lor \lnot\psi_{unq}(j, i)) \Rightarrow (\Psi_{i,j}^{evt} = 0)
\]

(23)

We use variable \(\text{miss}_i\) to capture whether \(r_i\) is a cache miss. Therefore, \(\text{miss}_i\) is set to 1 if \(r_i\) is a cache miss, and is set to 0 otherwise. We formulate the value of \(\text{miss}_i\), using the following constraints:

\[
\Theta_i^{\text{mp},\text{lru}} \equiv \left(\sum_{j \in \{1, i\}} \Psi_{i,j}^{evt} \geq \mathcal{A}\right) \lor \Theta_i^{\text{cold}}
\]

(24)

\[
\Theta_i^{\text{mp},\text{lru}} \equiv (\Theta_i^{\text{mp},\text{lru}} \Rightarrow (\text{miss}_i = 1))
\]

(25)

\[
\Theta_i^{\text{lru}} \equiv (\Theta_i^{\text{mp},\text{lru}} \Rightarrow (\text{miss}_i = 0))
\]

(26)

In Constraint (24), \(\mathcal{A}\) captures the associativity of the cache. Once a memory block is loaded into the cache, it requires at least \(\mathcal{A}\) unique cache conflicts to evict the block. If \(\Psi_{i,j}^{\text{evt}} \geq \mathcal{A}\), \(r_j\) has suffered at least \(\mathcal{A}\) unique cache conflicts since the last access of the memory block referenced by \(r_j\) – resulting \(r_i\) to be a cache miss. If \(r_i\) is not a cold miss (i.e. \(\lnot\Theta_i^{\text{cold}}\) holds) and it has not suffered \(\mathcal{A}\) unique cache conflicts, \(r_i\) will be a cache hit, as captured by Constraint (26).
Putting it all together. To derive the symbolic cache behavior \( \Gamma(pc) \), we gather all constraints over \( \{r_1, \ldots, r_n\} \) and the path condition \( pc \) as follows:

\[
\Gamma(pc) \equiv pc \land \bigwedge_{i \in [1, n]} \left( \Theta_i^{m,iru} \land \Theta_i^{h,iru} \land \bigwedge_{j \in [1, i]} \Theta_j^{em} \land \bigwedge_{j \in [1, i]} \Theta_j^{eh} \right) \tag{27}
\]

\( \Theta_i^{m,iru} \) and \( \Theta_i^{h,iru} \) together bound the value of \( miss_i \), which, in turn captures whether \( r_i \) is a cache miss. However, \( \Theta_i^{em} \) and \( \Theta_i^{eh} \) are dependent on symbolic variables \( \Psi_{j} \) where \( j \in [1, i] \). The bound on \( \Psi_{j} \) is captured via \( \Theta_j^{em} \) and \( \Theta_j^{eh} \) (Constraints (22)-(23)). Hence, the formulation of \( \Gamma(pc) \) includes both \( \Theta_j^{em} \) and \( \Theta_j^{eh} \) for \( j \in [1, i] \).

Complexity of constraints. The size of our constraint system, in order to check cache side-channel leaks, is \( O(n^3) \). Here \( n \) is the number of memory accesses. The dominating factor in our constraint system is the set of constraints generated from Constraint (13) and Constraint (22). In general, we generate constraints for each pair of memory accesses that may potentially conflict in the cache, leading to \( O(n^2) \) pairs in total. For each such pair, the constraint may have a size \( O(n) \) — making the size of overall constraint system to be \( O(n^3) \). However, our evaluation reveals that such a bound is pessimistic and the constraint system can be solved efficiently for real-life embedded programs.

5. Checking Information Leak

In this section, we instantiate CHALICE for two different observer models. In particular, we show the formulation of Equation (5) by leveraging on our symbolic cache model \( \Gamma(pc) \) (as described in Sections 4.3 and 4.4) and instantiating \( \Phi_o \) for different observer models. We assume that \( t_f \) is the observed execution trace for input \( I \) and we wish to quantify how much information about input \( I \) is leaked through \( t_f \).

Observation via total miss count. In this scenario, an attacker can observe the number of cache misses in different executions [11]. The observer \( O : \Sigma^* \rightarrow \text{Natural} \) is a function, where a sequence of cache hits and misses are mapped to a non-negative integer capturing the number of cache misses. Therefore, for a given trace \( t \in \Sigma^* \), \( O(t) \) captures the number of cache misses in the trace \( t_f \).

Recall that we use variable \( miss_i \) to capture whether the \( i \)-th memory access was a cache miss. We check the unsatisfiability of the following logical formula to record information leak:

\[
\bigvee_{e \in \text{Path}} \left( \Gamma(pc_e) \land \left( \sum_{i \in [1, n_e]} miss_i = O(t_f) \right) \land \pi \right) \tag{28}
\]

where \( n_e \) is the number of memory accesses occurring along path \( e \) and \( \pi \) is a predicate defined on program inputs. Concretely, if Constraint (28) is unsatisfiable, we can establish that the information \( \neg \pi \equiv \text{true} \) is leaked through the execution trace \( t_f \). By performing such unsatisfiability checks over the entire program input space, we quantify the information leak \( \mathcal{L}(t_f) \) through execution trace \( t_f \) (cf. Equation (6)).

Observation via hit/miss sequence. For an execution trace \( t \in \Sigma^* \), an observer can monitor hit/miss sequences from \( t \). Concretely, let us assume \( \{o_1, o_2, \ldots, o_k\} \) is the set of positions in trace \( t \) where the observation occurs. If \( n \) is the total number of memory accesses in \( t \), we have \( o_i \in [1, n] \) for each \( i \in [1, k] \).

We define the observer \( O : \Sigma^* \rightarrow \{0, 1\}^k \) as a projection from the execution trace onto a bitvector of size \( k \). Such a projection satisfies the following conditions: \( O(t)_i = 1 \) if \( t_{o_i} = m \) and \( O(t)_i = 0 \) otherwise. \( O(t)_i \) captures the \( i \)-th bit of \( O(t) \) and similarly, \( O(t)_i \) captures the \( i \)-th element in the execution trace \( t \). Note that a strong observer could map the entire execution trace to a bitvector of size \( n \).

For such an observer, we check the unsatisfiability of the following formula to record information leak:

\[
\bigvee_{e \in \text{Path}} \left( \Gamma(pc_e) \land \left( \bigwedge_{i \in [1, \ldots, k]} (o_i \leq n_e) \land \neg miss_{o_i} = O(t_f)_i \right) \land \pi \right) \tag{29}
\]

where \( \pi \) is a predicate on program inputs. By generating such predicates over the input space, we quantify the information leaked about input \( I \) via \( \mathcal{L}(t_f) \) (cf. Equation (6)).

Although we instantiate CHALICE for two observer models, we believe that our framework is generic to capture a wide range of such models. In particular, we can tune CHALICE for any observer model that is expressed via symbolic constraints over variables \( miss_i \).

6. Implementation Aspects

In this section, we discuss some crucial implementation aspects for the efficiency and effectiveness of CHALICE.

Implementation setup. We implemented CHALICE on top of the KLEE symbolic virtual machine [2]. However, in order to design such an implementation, we faced the following challenges.

(a) Binary code. Assume all loads are loading bytes

```
/* load local variable */
load reg1, 16[80p]
beq reg1, reg1, 127, L2
L1: load reg2, 24[80p]  
   br L3
L2: load reg2, 32[80p]  
L3: add reg1, reg2  

/* register spill */
store reg1, 40[80p]  
sub reg1, reg2
```

(b) Translated LLVM code, some details are removed for clarity

```
/* load local variable */
load reg1, 16[80p]
beq reg1, reg1, 127, L2
L1: load reg2, 24[80p]
   br L3
L2: load reg2, 32[80p]
L3: add reg1, reg2
```

Figure 5. Translation from binary code to LLVM code
KLEE works on LLVM bitcode \cite{Klee13}. Considering cache performance, at the level of LLVM bitcode, introduces several inaccuracies. For instance, LLVM bitcode uses an unbounded number of virtual registers. In contrast, any given execution platform only contains a finite number of physical registers. In order to understand how this impacts memory performance, consider the example in Figure 5.

In Figure 5, we assume that the execution platform contains only two physical registers. As a result, a register spill is required in the binary code to preserve the functionality of the LLVM bitcode. In general, aggressive compiler optimizations may change the structure and memory behaviour of the LLVM bitcode dramatically, when translated into native binary.

In order to solve this challenge and still use the power of symbolic execution on target-independent LLVM bitcode, we have designed a translator that converts binary code to LLVM bitcode. Such a translation must preserve the following properties to produce a valid LLVM bitcode. First, we ensure that each load/store instruction in the binary code to have a functionally equivalent load/store instruction in the translated bytecode. Secondly, we preserve the static-single-assignment (SSA) form of LLVM bitcode by systematically inserting Phi functions. Thirdly, several instructions at the machine code may require multiple LLVM instructions to implement. The LWL and LWR are such machine-level instructions for MIPS architecture. Finally, LLVM bitcode is strongly typed. As a result, LLVM bitcode uses different instructions for pointer arithmetic as compared to general-purpose arithmetic. We use a lightweight type inference on the binary code and compute the appropriate LLVM instruction for a given machine-level instruction. Figure 5(b) demonstrates how the example binary code is translated into LLVM bitcode. The instruction `getelemptr` handles pointer arithmetic in the LLVM bitcode.

From a technical point of view, we have designed a translator that converts PISA binaries (a MIPS like architecture) into LLVM bitcode. Such a translator is unique in the sense that it focuses on preserving the memory behaviour during the translation. Nevertheless, our translator may introduce additional instructions to preserve the SSA semantics of LLVM bitcode. Such additional instructions are not part of the binary code. In order to exclude such instructions from our analysis, we annotate the LLVM bitcode with a mapping from each memory-related instruction in the binary to the respective memory-related instruction in the LLVM bitcode. As a result, CHALICE accurately captures the cache side-channel leaks for applications compiled into PISA binaries.

Our translator currently does not handle indirect jump instructions. However, we can use a lightweight static analysis to compute the potential targets for indirect jumps and the translator can easily be modified to take this into account. Besides, CHALICE is modular in the sense that it can easily be adapted for a different architecture. This can be accomplished only by extending the translator to convert the respective machine code into LLVM bitcode.

### Table 1. Salient Features of the Evaluated Subject Programs

| Program                  | Lines of C code | Lines of LLVM code | Max. Memory access |
|-------------------------|-----------------|--------------------|--------------------|
| AES                     | 800             | 4950               | 2134               |
| AES                    | 1428            | 1800               | 420                |
| DES                     | 552             | 1990               | 334                |
| RC4                     | 160             | 668                | 1538               |
| RC5                     | 256             | 1820               | 410                |

#### 7. Evaluation

**Experimental setup.** In order to evaluate the effectiveness of CHALICE, we have chosen cryptographic applications from OpenSSL library \cite{OpenSSL} and other software repository \cite{LinuxGDK}, as well as applications from Linux GDK library. The choice of our subject programs is motivated by the critical importance of validating security-related properties in these applications. Some salient features of the evaluated subject programs is outlined in Table 1. We have performed all our experiments on an Intel I7 machine with 8GB of RAM and running Debian as operating systems.

#### 7.1. Generating Predicates on Inputs

Using CHALICE, we can select an arbitrary number of bits in the program input to be symbolic. These symbolic bits capture the high sensitivity of the input subspace and our framework focuses to quantify the information leaked about this subspace. For instance, in encryption routines, the bits of private input (e.g. a secret key) can be made symbolic. Without loss of generality, in the following discussion, we assume that the entire input is sensitive and we make all input bits to be symbolic.

Let us assume \( N \)-byte program input. We use the notation \( k[i] \) to capture the \( i \)-th byte of an arbitrary input \( k \). Similarly, we use \( k[i,j] \) to capture the \( j \)-th bit of the \( i \)-th
byte in \( k \). We generate the following predicates on inputs for quantifying information leak \( \mathcal{L}(t_i) \) (cf. Equation (6)).

\[
P_{\text{bit}} = \{ k[i, j] = v \mid i \in [1, N], j \in [1, 8], v \in [0, 1] \}
\]

\[
P_{\text{byte}} = \{ k[i] = v \mid i \in [1, N], v \in [0, 255] \}
\]

It is worthwhile to mention that for a 16-byte sensitive input (e.g. in AES-128), \( P_{\text{bit}} \) and \( P_{\text{byte}} \) lead to 256 and 4096 calls to the solver, respectively to quantify \( \mathcal{L}(t_i) \).

### 7.2. Experience with AES-128

We used two different implementations \([1], [3]\) of the Advanced Encryption Standard (AES). AES is a widely used encryption standard in achieving confidential communication. AES has been of great importance for delivering security in embedded systems because of its sound protection even on credit cards. Therefore, it is crucial to validate security-related properties, such as side-channel resistance, for AES.

AES has input-dependent memory accesses. In particular, different encryption rounds of AES resolve around accessing an \textit{sbox} – a matrix-like structure kept in main memory (DRAM). During encryption, AES code accesses varied locations in the \textit{sbox}. The location of the \textit{sbox} being accessed, for a given instruction, depends on the secret key. That is, the sequence of memory blocks, accessed during encryption, is dependent on the value of the secret key. As a result, we potentially obtain different cache performance for different secret keys.

#### 7.2.1. Key result

We used an 8 KB direct-mapped cache with a line size of 32 bytes. This size is big enough to keep the entire \textit{sbox} of AES in the cache. We executed AES in simplescalar simulator \([8]\) (cf. Figure 6) with a test suite and obtained the respective set of observations (e.g. number of cache misses). For such observations, we intended to check how much information is leaked through a bit or a byte, by generating predicates \( P_{\text{bit}} \) and \( P_{\text{byte}} \) (as described in Section 7.2), respectively.

For the collected set of observations, CHALICE quantifies \( \mathcal{L}(t_i) \) to be 0 when the set of predicates \( P_{\text{bit}} \) is used. Therefore, each observation (e.g. the number of cache misses) is possible irrespective of whether an arbitrary bit of the AES input (in both implementations of AES \([1], [3]\)) is “1” or “0”. Therefore we deduce, for the given set of observations, there does not exist any dependency between the cache performance and the value of an arbitrary bit of the key.

Figure 6(a) captures an outline of information leak highlighted via CHALICE, for two different implementations of AES \([1], [3]\). For each byte of the 16-byte secret key, we show the amount of information leaked through the number of cache misses. For instance, we establish for certain observations, that as many as 251 values (out of 256) are leaked for each byte of AES key (in the implementation \([1]\)).

\textbf{This means, there exists at least 2^{251} possible keys (out of a total 2^{128}) that can be eliminated just by observing the cache misses.} Such an information gives the designer valuable insights when designing embedded systems, both in terms of choosing an AES key and a cache architecture, in order to avoid serious security breaches. In contrast to the implementation of \([1]\), we can observe from Figure 6(a) that the implementation of AES from OpenSSL exhibits substantially fewer information leaks. For instance, certain key bytes of OpenSSL AES do not leak any information through the number of cache misses.

In our framework, we also investigated on adversaries who can observe the sequence of cache hits and misses, instead of just the overall number of cache misses. However, to simplify our evaluation, we focused on sequences of length 1, and considered all the memory accesses. Our goal is to check the dependency between the AES-key and the hit/miss characteristics of an arbitrary memory access.

Figure 6(b) captures a snapshot of dependencies between AES-key bytes and the cache behavior of different memory accesses. For instance, the maximum values leaked through a byte can be as high as 235, as shown via Figure 6(b). Similar to Figure 6(a), we also observe that the AES implementation from OpenSSL leaks substantially less information, as compared to the implementation in \([1]\), when cache behavior is observed individually for each memory access.

#### 7.2.2. Sensitivity of Information Leak w.r.t Cache Size

Figures 7(a)-(b) capture the sensitivity of information leakage with respect to different configurations. For all experiments, the replacement policy is set to LRU and the cache-line size is set to a fixed 32 bytes. Figures 7(a)-(b) captures the information-leakage-sensitivity for observations via a given number of cache misses. Increasing cache size (or associativity) may have two contrasting effects as follows.

For a given cache size, let us assume a subset of the input space \( I_C \subseteq I_{\text{ent}} \cup I_{\text{conf}} \cup I_{\text{obs}} \) (where \( I_{\text{ent}} \cup I_{\text{conf}} \cup I_{\text{obs}} \) is the entire input space) which leads to \( C \) cache misses. Increasing cache size reduces cache conflict. Therefore, it is possible that some input \( i \in I_{\text{ent}} \), which leads to more than \( C \) cache misses with a smaller cache, produces \( C \) cache misses with the increased cache size. This tends to increase the number of inputs leading to \( C \) cache misses, thus reducing the amount of information leaked through observing \( C \) misses. Secondly, some input \( i \in I_{\text{conf}} \) may have less than \( C \) cache misses with increased cache size. This may reduce the number of inputs having \( C \) cache misses, thus increasing the potential leakage through the observation of \( C \) cache misses. In Figure 7(a), the reduction in cache side-channel leakage is visible for cache sizes up to 16 KB, for AES implementation from \([1]\). However, for a 4-way 32 KB cache, we observe the increase in information leakage. This is because the number of possible keys, leading to a given observation, is reduced considerably.

Figures 8(a)-(b) capture the sensitivity of information leakage (w.r.t. cache size) for an adversary who can observe the cache behavior of an arbitrary memory access. Concretely, consider the bars in Figure 8(a) for 8 KB and 32 KB caches. For a 2-way, 8 KB cache, a significant information about the first key byte is leaked. With 32 KB
caches, the number of cache conflicts reduces substantially, but we observe substantial leakage of information about key bytes one, six, and ten. Therefore, even though the increased cache size improves performance, it might make the overall system potentially less secure, as shown in Figure 8(b). In summary, we believe such insights are valuable for designers to build secure systems.

The evaluation also reveals that the AES implementation from OpenSSL exhibits information leak, as shown in Figure 8(b). Figure 8(b) highlights the last four bytes of the key to experience more leakage of information as compared to other key bytes.

7.3. Experience with DES

Data Encryption standard (DES) [3] is a symmetric key algorithm for electronic data. The OpenSSL implementation of DES encrypts 64 bit message with a 64 bit secret key. Figures 9(a)-(b) summarize our result on quantifying information leak in DES. Figure 9(a) reports information leaks through observing miss count. For instance, using 8KB caches, DES leaks more than 150 values for several key bytes. In contrast, information leak in the OpenSSL version of AES is relatively sparse and it generally leaks less information about key bytes (cf. Figure 7(b)). In Figure 9(b), we observe a similar trend, as DES continues to suffer from information leak when the cache behavior of an arbitrary memory access is observed. Our results summarize potentially insecure nature of DES, even if we only consider security leaks through cache behaviour.

7.4. Experience with RC4

RC4 [3] is a stream cipher. It uses variable length key (between 40 and 2048 bits) and it is considered to be vulnerable in many applications. In our evaluation, we studied how RC4 leaks information through cache side channels. We analyzed the OpenSSL version of RC4 implementation, where we fixed the size of key to be 64 bits. Figures 10(a)-(b) outline our findings. Figure 10(a) summarizes our results for miss-count-based observer models. CHALICE highlights information being leaked about the first byte. For bigger
7.5. Experience with RC5

RC5 [3] is a symmetric block cipher, which is suitable for both software and hardware implementation. RC5 has a variable word size and a variable-length secret key. In our evaluation, we analyzed the RC5 implementation available in the OpenSSL library and we fixed size of both the plaintext and the secret key to be 16 bytes.

During the execution of RC5, our tool CHALICE does not report any symbolic memory address. This means, for any memory-related instruction, the referenced address is independent of secret key. As a result, cache performance (i.e. number of cache misses or the sequence of hits and misses) of RC5 is unrelated to input and does not exhibit information leak with respect to the observer models studied in this paper. It is also worthwhile to mention that RC5 does not have any key-dependent branches. Therefore, we can turn the report generated by CHALICE into verification, meaning that we can prove the absence of information leak according to the observer models explored in this paper.

7.6. Experience with GDK Library

Figures 11(a)-(b) present the average information leak discovered in routines gdk_keyval_to_unicode and gdk_keyval_name from the Linux GDK library. As shown in Figure 11 several scenarios lead to a complete disclosure of information for the third and the fourth byte of input (i.e. 255 out of 256 values are leaked for these bytes). Moreover, the reported information leak persists across a wide-range of cache configurations. Upon close inspection, we discovered that the cache behaviour of
gdk_keyval_to_unicode and gdk_keyval_name is primarily dominated by the number of cold cache misses, which, in turn is heavily influenced by the path executed in the respective routine. As a result, observing the cache performance may lead to a disclosure of the (set of) paths taken in gdk_keyval_to_unicode and gdk_keyval_name. Since we include path condition pc within our symbolic cache model \( \Gamma(p_c) \) (cf. Constraint (27)), we can accurately quantify the information leak even in the presence of multiple program paths.

### 7.7. Analysis Time

Table 2 outlines the analysis time for different subject programs while using a direct-mapped 8KB cache. In most cases, a single call to the solver, which reports information leak via unsatisfiability check (e.g., Constraint (5)), is efficient. Due to the repeated calls to solver, checking the information leakage, for the entire input space, takes significant time. However, CHALICE incorporates anytime strategy, meaning the more time it runs the more accurately it can quantify the information leak. Besides, the timing reported in Table 2 is either consistent or decreases with
increasing cache size. This is due to the fact that our symbolic cache model uses the notion of cache conflict to encode the cache behavior and the size of our model does not vary dramatically with increasing cache size. Finally, the performance of CHALICE can be improved drastically using a parallel implementation. For instance, we can assign one or more independent threads to check information leaked about each input byte. We plan to implement such a parallel version of CHALICE in the future. Table 2 only reports timing for a sequential implementation in this paper.

7.8. Discussion

For the sake of brevity, we have only presented the quantification of information leak discovered through CHALICE. Of course, due to the symbolic nature of our analysis, CHALICE not only quantifies information leak, it also highlights which values might leak through a potential cache attack. Furthermore, for each memory-related instruction, CHALICE highlights the set of input values that may leak for a given execution. In summary, the report generated by CHALICE can be leveraged for debugging and fixing critical information leak scenarios. Some of the potential debugging strategies would be to restructure the code, suppressing or enabling compiler optimizations (such as bypassing the cache for certain memory blocks or using software-controlled memory) and choosing an appropriate hardware platform. In future, we plan to use CHALICE to study the impact of such hardware/software transformations on information leak.

8. Related Work

The closest to our work are approaches based on static analysis [16], [21]. These works quantify information leak from the static representation of a program. In particular, the information leak is quantified via the unique number of observations made by an attacker. As a result, these works are incapable to highlight critical information leak scenarios when a particular observation leaks substantially more information than the rest. CHALICE quantifies information leak from execution traces and therefore, it does not suffer from the aforementioned limitation. Our work is orthogonal to approaches proposed in [7], [10]. In particular, our approach targets arbitrary software binaries and it is not limited to the verification of constant-time cryptographic software. Besides, our approach has a significant flavor of testing and debugging, as we highlight information leak scenarios directly from execution traces. A recent work [22] aims to quantify side-channel leakage via symbolic execution and Max-SMT. However, this work does not take into account side-channel leaks through micro-architectural entities, such as caches.

Over the last few decades, cache-based side-channel attacks have emerged to be a prevalent class of security breaches for many systems. A detailed account on these attacks can be found in the survey [17]. The observer models used in this paper are based on existing cache attacks [6], [11]. However, we believe that CHALICE is generic to incorporate more advanced attack scenarios [12], [19], [20], as long as the attacks are expressed via the intuition given in Section 5.

To defend against cache-based side-channel attacks, several countermeasures have been proposed over the past few years. Some of these countermeasures require hardware support, such as designing new cache architecture [26] or compiler support, such as devising new instruction-based scheduling [23]. More recently, the approach described in [14] leverages on software diversity at runtime to randomize the cache behavior and hence, reducing the probability for a potential cache side-channel attack. Our proposal is orthogonal to approaches proposing countermeasures. Of course, we believe that our framework can be leveraged as a valuable tool to discover potential flaws in countermeasures proposed to mitigate cache side channels.

Finally, static cache analysis [24] is still an active research topic. Compared to static cache analysis, our approach has significant flavors of testing and debugging. Moreover, our approach can highlight memory accesses that leak significant information via side-channels. This can be leveraged to drive security-related optimizations.

In summary, we propose a new approach to quantify cache side-channel leakage from execution traces and not from the static representation of the program. We demonstrate that such an approach clearly has benefits over approaches based on static or logical analysis. This is because CHALICE can highlight critical information leak scenarios that are impossible to be discovered by competitive static or logical analysis.

9. Concluding Remarks

In this paper, we propose a new approach to quantify cache side-channel leakage. To the best of our knowledge, CHALICE is the first work to categorize input segments with respect to memory performance. We illustrate that the mechanism of CHALICE is highly desirable for security testing of arbitrary software, specifically, to detect the amount of information that can leak through memory performance. Besides security testing, CHALICE can also be used to discover bugs while writing constant-time cryptographic applications. We demonstrate the usage of CHALICE to highlight critical information leak scenarios in real-world software – including applications from OpenSSL and Linux GDK libraries.

We believe CHALICE provides a platform to lift the state-of-the-art in security testing, in particular, detecting security-related flaws due to side channels. As a result, we envision to extend CHALICE for side channels other than caches and use it to detect the potential of advanced side-channel attacks not investigated in this paper. We hope that the core idea of CHALICE would influence regular activities in software testing and in testing regressions.
References

[1] Advanced Encryption Standard Implementation. https://github.com/B-Con/crypto-algorithms.
[2] KLEE LLVM execution engine. https://klee.github.io/.
[3] OpenSSL Library. https://github.com/openssl/openssl/tree/master/crypto.
[4] The LLVM compiler infrastructure. http://llvm.org/.
[5] UC Davis, Mathematics. Latte integrale. https://www.math.ucdavis.edu/~latte/.
[6] Onur Acıic¸mez and Çetin Kaya Koç. Trace-driven cache attacks on AES. In Information and Communications Security. Springer, 2006.
[7] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In USENIX, pages 53–70, 2016.
[8] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An infrastructure for computer system modeling. Computer, 35(2), 2002.
[9] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shabbaz, and Shin Yoo. The oracle problem in software testing: A survey. IEEE transactions on software engineering, 41(5):507–525, 2015.
[10] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Carlos Daniel Luna, and David Pichardie. System-level non-interference for constant-time cryptography. In CCS, pages 1267–1279, 2014.
[11] Daniel J Bernstein. Cache-timing attacks on AES, 2005.
[12] Billy Bob Brumley and Risto M Hakala. Cache-timing template attacks. In ASIACRYPT. Springer, 2009.
[13] James Clause, Wanchun Li, and Alessandro Orso. Dytan: a generic dynamic taint analysis framework. In ISSTA. ACM, 2007.
[14] Stephen Crane, Andrei Homescu, Stefan Brumthaler, Per Larsen, and Michael Franz. Thwarting cache side-channel attacks through dynamic software diversity. In NDSS, 2015.
[15] John Demme, Robert Martin, Adam Waksman, and Simha Sethumadhavan. Side-channel vulnerability factor: A metric for measuring information leakage. In ISCA, 2012.
[16] Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. CacheAudit: a tool for the static analysis of cache side channels. TISSEC, 18(1):4, 2015.
[17] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. In Cryptology ePrint Archive, 2016. https://eprint.iacr.org/2016/613.pdf/.
[18] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In PLDI, 2005.
[19] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In USENIX Security, 2015.
[20] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games—bringing access-based cache attacks on AES to practice. In IEEE Symposium on Security and Privacy. IEEE, 2011.
[21] Boris Köpf, Laurent Mauborgne, and Martín Ochoa. Automatic quantification of cache side-channels. In CAV. Springer, 2012.
[22] Corina S. Pasareanu, Quoc-Sang Phan, and Pasquale Malacaria. Multi-run side-channel analysis using symbolic execution and maxsmt. In CSF, 2016.
[23] Deian Stefan, Pablo Buiras, Edward Z Yang, Amit Levy, David Terei, Alejandro Russo, and David Mazières. Eliminating cache-based timing attacks with instruction-based scheduling. In ESORICS, pages 718–735. Springer, 2013.
[24] Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. Real-Time Systems, 18(2-3), 2000.
[25] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on aes, and countermeasures. Journal of Cryptology, 23(1):37–71, 2010.
[26] Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. In ISCA, pages 494–505, 2007.