Quantifying the Information Leak in Cache Attacks via Symbolic Execution

Sudipta Chattopadhyay  
Singapore University of Technology and Design

Ahmed Rezine  
Linköping University

Moritz Beck  
Saarland 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.

CCS CONCEPTS

- Security and privacy → Software security engineering;  
- Computer systems organization → Embedded software;

KEYWORDS

Side channel, Security, Cache, Symbolic execution

ACM Reference Format:

https://doi.org/10.1145/3127041.3127044

1 INTRODUCTION

Cache timing attacks [12] are among the best known side channel attacks [23] 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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

MEMOCODE ’17, September 29-October 2, 2017, Vienna, Austria  
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.  
ACM ISBN 978-1-4503-5093-8/17/09 ... $15.00  
https://doi.org/10.1145/3127041.3127044

Figure 1: For a fixed input message, the plot shows the number of keys leading to a given number of cache misses incurred by executing AES-128 encryption (sample size = 256000 keys)

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 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 [4] 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 subject 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.\(^1\)

It is this precision of its symbolic analysis that sets CHALICE apart from the state of the art. Existing works [22, 28] 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 exhibit substantial information leakage through side channels. Given a set of inputs (typically as part of a testing pipeline), CHALICE can precisely quantify the leak for each input, and thus provide a full spectrum that characterizes inputs with respect to information leakage.

The remainder of this paper is organized as follows. After giving an overview of CHALICE (section 2), we make the following contributions:

1. We present CHALICE, a new approach to precisely quantify information leak in execution and its usage in software testing (section 3).
2. We introduce a symbolic cache model to handle various cache configurations and instantiate CHALICE to detect cache side channel leakage (section 4). This is the first usage of symbolic execution to quantify information leakage by relating cache and program states.
3. We demonstrate generalizations across multiple observer models (section 5).
4. We provide an implementation based on LLVM and the KLEE symbolic virtual machine. Source code of CHALICE and all experimental data is publicly available: https://bitbucket.org/sudiptac/chalice
5. We evaluate our CHALICE approach (section 6) to show how we quantify the information leaked through execution in several libraries, including OpenSSL and Linux GDK libraries, and show that the information leak can be as high as 251\(^16\) for certain implementations [4] of AES-128.

After discussing related work (section 7), we close with conclusion and consequences (section 8).

## 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 data 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. Also for clarity, we assume that conditional checks, in this example, do not involve any access to the data cache (i.e. \(k\) is assigned to a register). However, it is worthwhile to note that our framework CHALICE handles arbitrary execution traces and it handles all instructions with arbitrary cache behaviours. 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: \(k = 0\) (hit/miss sequence = \((m,m,m))\) and \(k \in [1,127]\) (hit/miss sequence = \((m,m,h))\). A similar exercise for the code in Figure 2(b) results in the following partition of the sensitive input space: \(k \in [0,255]\) \(\land (k \mod 2 = 0)\) (hit/miss sequence = \((m,m,m))\) and \(k \in [0,255]\) \(\land (k \mod 2 \neq 0)\) (hit/miss sequence = \((m,m,h))\).

#### 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 \(k\) is odd. Such information still demands a probability of \(\frac{1}{128}\) 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 [22, 28] 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 (hence, observations), for arbitrary values of \(k\). Therefore, approaches based on static analysis [22, 28] will consider these two code fragments equivalent in terms of cache side-channel leakage. As a result, a crucial information leak scenario, such as the execution of code fragment in Figure 2(a) with \(k = 0\), will go completely unnoticed. Techniques based on verifying that programs execute in constant time typically check that memory accesses do not depend on sensitive inputs [6, 10]. 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. Hence, we track the relationship between input and cache performance through a symbolic model of the cache.

**Limitations of side-channel vulnerability metrics.** In contrast to existing works on measuring cache side-channel leakage [21], we do not aim to check the strength of an attacker to observe information through side channel. Although promising, this work [21] 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 [21] 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 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. In our earlier works [17] [11], we have shown how such an effective test suite can be generated automatically. Since the observation by an attacker (e.g. number of cache miss) corresponds to a (set of) test input(s), 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 program needs to be restructured to avoid such information leak.

**How CHALICE works.** Let us assume that we execute the code in Figure 2(a) with some input \( I \in [0, 255] \) and observe the trace \( t_I = \langle m, m, m \rangle \). 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 \( k \). Concretely, CHALICE constructs \( \Gamma(0 \leq k \leq 127) \) and \( \Gamma(128 \leq k \leq 255) \), which encode all cache hit/miss sequences for inputs satisfying \( 0 \leq k \leq 127 \) and \( 128 \leq k \leq 255 \), respectively. While exploring the path for inputs \( k \in [0, 127] \), we record a sequence of symbolic memory addresses \( \langle \&p[k], \&q[255-k], \&p[k] \rangle \), where \( \&x \) denotes the address of value \( x \). Since we started execution with an empty cache, the first access to \( p[k] \) inevitably incurs a cache miss, irrespective of the value of \( k \). The subsequent accesses can be cache hits, cold misses (first access to the respective cache line) or eviction misses (non-first access to the respective cache line). For instance, we check whether the “store” instruction suffers a cold data-cache miss as follows:

\[
(0 \leq k \leq 127) \land (\text{set}(\&p[k]) \neq \text{set}(\&q[255-k])) \land (\text{set}(\&p[k]) \neq \text{set}(\&p[k]))
\]

where \( \text{set}(\&x) \) captures the cache line where memory address \( \&x \) is mapped to. Intuitively, the constraint checks whether access to \( p[k] \) (via the “store” instruction) touches a cache line for the first time. Constraint (1) is clearly unsatisfiable, leading to the fact that the “store” instruction does not access a cache line for the first time during execution.

Subsequently, we check whether the second access to \( p[k] \) can suffer an eviction miss. To this end, we check whether \( q[255-k] \) can evict \( p[k] \) from the cache as follows:

\[
(0 \leq k \leq 127) \land (\text{set}(\&p[k]) = \text{set}(\&q[255-k])) \land (\text{tag}(\&p[k]) \neq \text{tag}(\&q[255-k]))
\]

where \( \text{tag}(\&x) \) captures the cache tag associated with the accessed memory block. Intuitively, Constraint (2) is satisfied if and only if \( q[255-k] \) accesses a different memory block as compared to \( p[k] \), but \( q[255-k] \) and \( p[k] \) access the same cache line (hence, causing an eviction before \( p[k] \) was accessed for the second time). In this way, we collect Constraints (1)-(2) to formulate the cache behaviour of a memory access into \( \Gamma(0 \leq k \leq 127) \).

After constructing \( \Gamma(0 \leq k \leq 127) \), we explore the path for inputs \( k \in [128, 255] \) and record the sequence of memory accesses.
p[k], q[k − 128] and p[k]. Performing a similar exercise, we can show that the second access to p[k] cannot be a cold miss along this path. In order to check whether the second access to p[k] was an eviction miss along this path, we check whether q[k − 128] can evict p[k] from the cache as follows:

\[(128 \leq k \leq 255) \land (set(\&p[k]) = set(\&q[k − 128])) \land (tag(\&p[k]) \neq tag(\&q[k − 128]))\]  

Constraint (3) is used to formulate \(\Gamma(128 \leq k \leq 255)\) and is unsatisfiable. This is because only p[0] shares a cache line with q[255] (i.e. set(\&p[0]) = set(\&q[255])) and therefore, set(\&p[k]) = set(\&q[k − 128]) is evaluated false for 128 ≤ k ≤ 255. As a result, the second access to p[k] is not a cache miss for any input k ∈ [128, 255].

From the observation \((m, m, m)\), we know that the second access to p[k] was a miss. From the discussion in the preceding paragraph, we also know that this observation cannot occur for any inputs k ∈ [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 \((m, m, m)\).

Relation to entropy. In the preceding example, CHALICE computes the number of impossible values of k, for a given observation. This, in turn, can be used to compute the uncertainty to guess k, provided the respective observation occurred. For instance, if the attacker observes the sequence \((m, m, m)\), then the uncertainty to guess k is “0” bit (as exactly one value of k is possible for observation). If we assume that k was uniformly distributed, the initial uncertainty to guess k was 8 bits (since k is an 8-bit input in the example). This leads to a reduced uncertainty of 8 bits when the sequence \((m, m, m)\) was observed by the attacker.

3 FRAMEWORK
In this section, we formally introduce CHALICE.

3.1 Foundation

3.1.1 Threat model. Side-channel attacks are broadly classified into synchronous and asynchronous attacks [30]. In a 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 possibility is not available for asynchronous attacks. Synchronous attacks are clearly easier to perform, since the attacker does not need to infer the start and end of the targeted routine under attack. For instance, in a synchronous attack, the attacker can trigger encryption of known plaintext messages and observe the encryption-timing [12]. 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 on 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 of accessing the address space of the target routine.

3.1.2 Notations. The execution of program \(P\) on input \(I\) results in an execution trace \(I_t\). \(I_t\) 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 leakage through \(I_t\). We capture this quantification via \(L(I_t)\). We assess the information leakage with respect to an observer. An observer \(O\) is a mapping \(O : \Sigma^* \rightarrow \mathbb{N}\) where \(\mathbb{N}\) is a countable set. For instance, an observer \(O : \Sigma^* \rightarrow \mathbb{N}\) can count the number of misses and will associate both sequences \((m, h, m, h)\) and \((m, m, h, h)\) 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 length two would be enough to differentiate \((m, h, m, h)\) and \((m, m, h, h)\).

We use the variable \(\text{miss}_i\) 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 variables \(\text{miss}_1, \text{miss}_2, \ldots, \text{miss}_n\). For instance, \(\Phi_O \equiv \sum_{i=1}^{n} \text{miss}_i = 100\) 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 mean the interpretation of an observation constraint \(\Phi_O\) along a program path \(e\). For example, \(\Phi_{O,e} \equiv \sum_{i=1}^{300} \text{miss}_i = 100\) if \(\Phi_O \equiv \sum_{i=1}^{300} \text{miss}_i = 100\) and the path \(e\) has 300 memory accesses. \(\Phi_{O,e}\) amounts to false if \(\Phi_O\) requires a different number of memory accesses than those provided by the path \(e\). 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 behavior 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 \([\text{miss}_1, \text{miss}_2, \ldots, \text{miss}_n]\). Therefore, if \(\Gamma(I) \land \Phi_O\) is unsatisfiable, we can deduce that the respective observation \(\Phi_O\) did not occur for any input \(I \in I\).

We now describe how \(L(I_t)\) is computed based on the notations and the intuition mentioned in the preceding.
3.2 Quantifying Information Leak in Execution

Figure 3 provides an outline of our entire framework. We symbolically execute a program $P$ and compute the path condition [24] 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 $\Gamma(\ell)$ to capture possible cache hit/miss sequences in an execution path, which was activated by a set of inputs $\ell$. In an abuse of notation, we capture the set of inputs $\ell$ via path conditions. For instance, in Figure 2(a), we use $\Gamma(0 \leq k \leq 127)$ to encode all possible cache hit/miss sequences for inputs activating the $1 \text{st}$ branch.

For an arbitrary execution path $e$, let $pc_e$ be the path condition. Along this path, we record each memory access and we consider its cache behavior via variables $miss_1, miss_2, \ldots, miss_n$. In particular, any solution of $\Gamma(pc_e) \land (miss_i = 1)$ captures a concrete input $1 \models pc_e$ and such an input $1$ 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 $1 \models pc_e$, then $\Gamma(pc_e) \land \Phi_{O,e}$ is always satisfiable.

We capture the information leak through trace $t_f$ as follows:

$$L(t_f) = 2^N - \sum_{e \in \text{Paths}} |\Gamma(pc_e) \land \Phi_{O,e}|_{\text{sol}}$$

(4)

where $N$ is size of program input (in bits), $\Phi_{O,e}$ is the interpretation of the observation constraint on path $e$. $\text{Paths}$ is the set of all program paths and $pc_e$ is the path condition for program path $e$. $|X|_{\text{sol}}$ captures the number of solutions satisfied by predicate $X$. It is worthwhile to note that $|\sum_{e \in \text{Paths}} \Gamma(pc_e) \land \Phi_{O,e}|_{\text{sol}}$ accurately captures the number of program inputs that exhibit the observation satisfied by $\Phi_O$. In other words, Equation (4) quantifies the number of program inputs that do not exhibit the observation, as captured by $\Phi_O$. Hence, if the attacker observes $\Phi_O$, then she can deduce as many as $L(t_f)$ inputs were impossible for $\Phi_O$.

Relation to entropy. The secrecy of a sensitive input is usually measured as the uncertainty of an attacker to guess the respective input. This uncertainty can be computed via several metrics, such as Shannon entropy [19]. For a given distribution $\lambda$ for an $N$-bit sensitive input $k$, Shannon entropy is computed as follows:

$$H(\lambda) = - \sum_{k' \in \{0, 2^N\}} \lambda(k') \log_2 \lambda(k')$$

(5)

If $\lambda$ is a uniform distribution, initial uncertainty $H(\lambda) = N$.

For observation constraint $\Phi_O$, the remaining uncertainty can be computed as follows:

$$H(\lambda_{\Phi_O}) = - \sum_{k' \in \{0, 2^N\}} \lambda_{\Phi_O}(k') \log_2 \lambda_{\Phi_O}(k')$$

(6)

where $\lambda_{\Phi_O}(k')$ denotes the probability that the sensitive input value is $k'$ given the observation satisfies $\Phi_O$.

Recall that $|\sum_{e \in \text{Paths}} \Gamma(pc_e) \land \Phi_{O,e}|_{\text{sol}}$ captures the number of inputs satisfying observation constraint $\Phi_O$. Hence, if the attacker assumes that all values of the sensitive input are equally probable for observation constraint $\Phi_O$, we get

$$H(\lambda_{\Phi_O}) = \log_2 |\sum_{e \in \text{Paths}} \Gamma(pc_e) \land \Phi_{O,e}|_{\text{sol}}$$

(7)

In summary, less the number of satisfying solutions for the formula $\sum_{e \in \text{Paths}} \Gamma(pc_e) \land \Phi_{O,e}$, less the uncertainty to guess the sensitive input $k$ given $\Phi_O$ holds. In the next section, we show how to compute $|\sum_{e \in \text{Paths}} \Gamma(pc_e) \land \Phi_{O,e}|_{\text{sol}}$ in an incremental fashion.

3.3 Cartesian Bounding of Information Leak

We justify and describe in the following a simple technique to tackle the scalability challenges. These challenges are faced by model counting for quantifying side-channel leakage.

3.3.1 Challenges to compute $L(t_f)$. Computing the exact value of $L(t_f)$ can become infeasible for the targeted problems in this paper. For instance, the input domain of targeted programs are as big as $2^{128}$ and the number of CNF clauses for an equivalent #SAT problem varies from 550K to two Millions, with hundreds of thousands of variables. Such a problem scale is several orders of magnitude higher than the programs evaluated by state-of-the-art model counting tools [14] supporting non-linear constraints. Specifically, we evaluated a scalable, but approximate model counter [16] and discovered it is incapable to deal with the length and the number of clauses generated for our subject programs.

In order to solve the aforementioned scalability issues, we leverage the capability of our framework to symbolically reason about partitions over input values. This has two crucial advantages: 1) we have an anytime algorithm to quantify the cache side-channel leakage. This means the longer time CHALICE runs, the more accurately it quantifies the information leakage. Moreover, our proposed approach is inherently parallel. 2) CHALICE not only quantifies the leakage, it also characterizes the equivalence class of secrets for a given observation. This is critical to identify weak secrets, such as weak passwords in password checker or weak keys in encryption routines. Finally, our proposed scheme also provides strong guarantees on the derived bound for $L(t_f)$.

In future, we will explore the synergy between our input partitioning scheme and model counting. Specifically, we can use model counting to reduce the number of solver calls required for an input partition. Similarly, model counting approaches can use our proposed partitioning to limit the search in an input-partition and potentially speed up the counting process.

3.3.2 Input space partitioning to compute $L(t_f)$. Consider a set $\Pi$ of program inputs containing all possible $N$-bit input values, i.e., $|\Pi| = 2^N$. A partition $P$ of $\Pi$ is a set $\{P[j] | 1 \leq j \leq |P|\}$ of disjoint non-empty sets whose union coincides with $\Pi$. Here, we write $|P|$ to mean the size of $P$, i.e. the number of subsets of $\Pi$ defined by the partition $P$. For example, a possible partition of size 2 is the one that partitions program inputs into two sets depending on the value of their first bit. Assume $K$ partitions $P_1, \ldots, P_K$ of the input set $\Pi$ for which no $(P_{1}[i_1] \cap P_{2}[i_2] \cap \ldots \cap P_{K}[i_K])$ is empty, for any
It is worthwhile to note that setting $K = 1$ amounts to enumerating all solutions as in Equation (4). This yields an exact but expensive measure of information leakage. In contrast, choosing $K = N$ amounts to checking information leak at bit-level. This results in a scalable amount of solver calls (only $2N$) but yields a potentially much weaker bound on information leakage. Therefore, $K$ provides a tunable parameter for the detection of information leakage. We can formalize this observation by introducing the notion of Cartesian refinement. Assume two Cartesian partitioning of $I$, $(P_1 \times \ldots \times P_K)$ and $(Q_1 \times \ldots \times Q_M)$. We say that $(P_1 \times \ldots \times P_K)$ refines, or is more precise than, $(Q_1 \times \ldots \times Q_M)$ if there is a surjective function $h : \{1, \ldots, M\} \rightarrow \{1, \ldots, N\}$ such that each partition $P_i$, for $i : 1 \leq i \leq K$, coincides with the Cartesian partitioning $(Q_j \times \ldots \times Q_{h^{-1}(i)})$ where $\{j_1, \ldots, j_{h^{-1}(i)}\} = h^{-1}(i)$.

**Lemma 3.2 (Cartesian bound refinement).** Assume two complete Cartesian partitioning of $I$ where $(P_1 \times \ldots \times P_K)$ refines $(Q_1 \times \ldots \times Q_M)$. For any trace $t_i$ that results in the observation constraint $\Phi_O$, the leakage bound obtained with $(P_1 \times \ldots \times P_K)$ is always larger or equal than the one obtained with $(Q_1 \times \ldots \times Q_M)$, i.e., $\mathcal{L}(t_i) \geq 2^N - \Pi_{1 \leq i \leq K} \left( |P_i| - |P_i^{\Phi_O}| \right) \geq 2^N - \Pi_{1 \leq i \leq M} \left( |Q_i| - |Q_i^{\Phi_O}| \right)$.

Proof. Let $S_{\Phi_O}$ be the subset of program inputs $I$ that exhibits the observation $\Phi_O$, i.e., containing all program inputs $I$ satisfying some path condition $pc_e$ where $(\Gamma(I) \land \Phi_O(e))$ holds. Observe that $S_{\Phi_O}$, which coincides with the denotation of $\left( S_{\Phi_O} \right)_{(P_1 \times P_2 \times \ldots \times P_K)}$ is subset of the denotation of $\left( S_{\Phi_O} \right)_{(Q_1 \times Q_2 \times \ldots \times Q_M)}$, which is subset of the denotation of $\left( S_{\Phi_O} \right)_{Q_1 \times \ldots \times Q_M}$. This leads to the following crucial inequalities: $|S_{\Phi_O}| \leq \Pi_{1 \leq i \leq K} \left( |S_{\Phi_O}[P_i]| \right) \leq \Pi_{1 \leq i \leq M} \left( |S_{\Phi_O}[Q_i]| \right)$. We conclude by observing that $S_{\Phi_O}[P_i] = P_i \setminus U_{P_i}^{\Phi_O}$ since $U_{P_i}^{\Phi_O}$ is the subset of $P_i$ for which $\Phi_O$ is unsatisfiable. Due to the classic path explosion problem in symbolic execution, it is possible that only a subset of execution paths $E \subseteq Paths$ can be explored within a given time budget. In such cases, we can quantify $\mathcal{L}(t_i)$ as follows.

**Lemma 3.3 (Anytime information leakage).** Assume a complete Cartesian partitioning $(P_1 \times \ldots \times P_K)$ of $I$ and a trace $t_i$ that results in the observation constraint $\Phi_O$. If $E \subseteq Paths$, let $\Phi_O^{E} \subseteq \Phi_O$ be the set of $P_i$ elements for which the observation constraint $\Phi_O$ is impossible along the paths $E$. The following holds:

$$\mathcal{L}(t_i) \geq \left| \bigcup_{e \in E} pc_e \right|_{sol} - \Pi_{1 \leq i \leq K} \left( |P_i| - |U_{P_i}^{\Phi_O,E}| \right) \geq \left( |\bigcup_{e \in E} pc_e \setminus E| |sol| - \left| \bigcup_{e \in E} pc_e \setminus E \right|_{sol} \right)$$

Proof. Observe that the set of all path conditions defines a partition of the set of program inputs. Hence $2^N$ coincides with the sum of $\left| \bigcup_{e \in E} pc_e \setminus E \right|_{sol}$ and $\left| \bigcup_{e \in Paths \setminus E} pc_e \setminus E \right|_{sol}$ Similarly, we observe that $\left| \bigcup_{e \in Paths} \left( \Gamma(pc_e) \land \Phi_O(e) \right) \right|_{sol}$ coincides with the sum of $\left| \bigcup_{e \in E} \left( \Gamma(pc_e) \land \Phi_O(e) \right) \right|_{sol}$ and $\left| \bigcup_{e \in Paths \setminus E} \left( \Gamma(pc_e) \land \Phi_O(e) \right) \right|_{sol}$. Therefore, information leakage $\mathcal{L}(t_i)$ coincides with the following:

$$\left| \bigcup_{e \in E} pc_e \setminus E \right|_{sol} + \left| \bigcup_{e \in Paths \setminus E} pc_e \setminus E \right|_{sol} - \left| \bigcup_{e \in E} \left( \Gamma(pc_e) \land \Phi_O(e) \right) \right|_{sol} - \left| \bigcup_{e \in Paths \setminus E} \left( \Gamma(pc_e) \land \Phi_O(e) \right) \right|_{sol}.$$
We note that both

\[ \bigvee_{e \in \text{Path}(E)} \left( \Gamma(pce) \land \Phi_O,e \right) \big|_{\text{sol}}. \]

Finally, the result follows from

\[ \bigvee_{e \in \text{Path}(E)} pce_{\text{sol}} \geq \bigvee_{e \in \text{Path}(E)} \left( \Gamma(pce) \land \Phi_O,e \right) \big|_{\text{sol}}. \]

as well as,

\[ \Pi_{1 \leq i \leq K} \left( p_{ij} - |\mathbb{P}_{p_{ij}}^{\Phi_O,E} | \right) \geq \bigvee_{e \in \text{E}} \left( \Gamma(pce) \land \Phi_O,e \right) \big|_{\text{sol}}. \]

For instance, using the segments defined before, \( \cup_{p_{ij}}^{\Phi_O,E} \) is the number of values \( v \) in \( 0 \leq v < 2^{N/K} \) for which no \( \Gamma(pce) \land \Phi_O,e \land (k_i = v) \) is satisfiable. Note the term \( \bigvee_{e \in \text{E}} pce_{\text{sol}} \) involves only path conditions. \( \bigvee_{e \in \text{E}} pce_{\text{sol}} \) can often be computed via model counting [2] in practice.

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

4 GENERATING SYMBOLIC CACHE MODEL

The technical contribution of our methodology is to establish a relation between a symbolic model for the cache and our leakage metric introduced in Section 3.2. In this section, we propose and formulate a novel and efficient symbolic model to encode the performance of direct-mapped caches. In direct-mapped caches, a given memory address can be mapped to exactly one cache line (as shown in Figure 2(d)). Of course, we evaluate CHALICE also for set-associative caches [18]. The symbolic models of set-associative caches are handled in a similar fashion as proposed in our previous work [17]. To describe our model, we shall use the following notations throughout our discussions:

- \( 2^S \) : The number of cache lines in the cache.
- \( 2^B \) : The size of a cache line (in bytes).
- \( \text{set}(r_i) \) : Cache line accessed by instruction \( r_i \).
- \( \text{tag}(r_i) \) : The tag that would be stored in the cache for the memory access by \( r_i \).

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 1F branch of Figure 2(a), we track the path condition \( 0 \leq k \leq 127 \) and the sequence of memory addresses \( \langle k[p, k] \rangle, \& p[255 - k] \rangle, \& p[k] \rangle. \) It is worthwhile to note that such memory addresses might capture symbolic expressions due to the dependency on program inputs. Concretely, we compute the path condition \( pce \) and the execution trace \( \Psi_{pce} \) for each explored path \( e \) as follows:

\[ \Psi_{pce} \equiv \left( \langle r_1, \sigma_1 \rangle, \langle r_2, \sigma_2 \rangle, \ldots, \langle r_{n-1}, \sigma_{n-1} \rangle, \langle r_n, \sigma_n \rangle \right). \]  (10)

where \( r_i \) captures the \( i \)-th memory-related instruction executed along the path and \( \sigma_i \) symbolically captures the memory address accessed by \( r_i \).

Modeling Symbolic Cache Access. Following the basic design principle of caches, we compute \( \text{set}(r_i) \) and \( \text{tag}(r_i) \) by manipulating the symbolic expression \( \sigma_i \). In particular, \( \text{set}(r_i) \) and \( \text{tag}(r_i) \) are formulated as follows:

\[ \text{set}(r_i) = (\sigma_i \Rightarrow \mathbb{B}) \land (2^S - 1); \text{tag}(r_i) = (\sigma_i \Rightarrow (\mathbb{B} + S)). \]  (11)

We note that both \( \text{set}(r_i) \) and \( \text{tag}(r_i) \) might be symbolic expressions due to the presence of symbolic expression \( \sigma_i \).

MEMOCODE ’17, September 29–October 2, 2017, Vienna, Austria

![Figure 4](image)

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

Modeling Cache Misses. We characterize cache misses into the following categories:

1. Cold cache misses. Instruction \( r_i \) 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. Instruction \( r_i \) 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 \( \text{tag}(r_j) \neq \text{tag}(r_i) \).

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 \( r_i \). Therefore, \( r_i \) suffers a cold miss if and only if the following condition holds:

\[ \Theta_i^{\text{cold}} \equiv \bigwedge_{p \in [1,i]} \left( set(p) \neq set(r_i) \right) \]  (12)

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 line. Therefore, we have the following:

\[ \psi_{\text{cnf}}(j, i) \equiv \left( set(r_j) = set(r_i) \right) \]  (13)

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

\[ \psi_{\text{dif}}(j, i) \equiv \left( \text{tag}(r_j) \neq \text{tag}(r_i) \right) \]  (14)

3. \( \psi_{\text{eqv}}(j, i) \): There does not exist any instruction \( r_k \) where \( k \in [1 + j, i) \), 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 \( r_j \). Since \( r_k \) is executed after \( r_j \) (in program order), \( r_j \) is not responsible for a cache miss at \( r_i \). We formulate the following constraint to capture this condition:

\[ \psi_{\text{eqv}}(j, i) \equiv \bigwedge_{k \in [1 + j, i)} \left( \text{tag}(r_k) \neq \text{tag}(r_i) \lor \text{set}(r_k) \neq \text{set}(r_i) \right) \]  (15)

Constraints (13)-(15) 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
In order to check whether \( r_i \) suffers a cache miss due to eviction, we need to check Constraints (13)-(15) for any \( r \in \{ r_1, r_2, \ldots, r_{i-1} \} \). This can be captured via the following constraint:

\[
\Theta_i^{emp} \equiv \bigvee_{j: 1 \leq j < i} \left( \psi_{cnf} (j, i) \land \psi_{diff} (j, i) \land \psi_{eqv} (j, i) \right) \tag{16}
\]

Instruction \( t_i \) will not suffer a cache miss due to eviction when, for all prior instructions, at least one of the Constraints (13)-(15) does not hold. This scenario is the negation of Constraint (16) and therefore, it is captured via \( -\Theta_i^{emp} \).

We use variable \( miss_i \) to capture whether instruction \( t_i \) suffers a cache miss. As discussed in the preceding paragraphs, \( t_i \) suffers a cold miss (i.e. satisfying Constraint (12)) or the memory block accessed by \( t_i \) would be evicted due to instructions executed before \( t_i \) (i.e. satisfying Constraint (16)). Using this notion, we formulate the value of \( miss_i \) as follows:

\[
\Theta_i^{m,dir} \equiv \left( \Theta_i^{emp} \lor \Theta_i^{cold} \right) \Rightarrow \left( \neg miss_i = 1 \right) \tag{17}
\]

\[
\Theta_i^{h,dir} \equiv \left( \neg \Theta_i^{emp} \land \neg \Theta_i^{cold} \right) \Rightarrow \left( miss_i = 0 \right) \tag{18}
\]

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

\[
\Gamma(pce) \equiv pce \land \bigwedge_{i \in [1, n]} \left( \Theta_i^{m,dir} \land \Theta_i^{h,dir} \right) \tag{19}
\]

5 CHECKING INFORMATION LEAK

In this section, we instantiate CHALICE by formulating \( \Phi_O \) for two different observer models. We assume that \( t_j \) is the observed execution trace for input \( I \) and we wish to quantify how much information about input \( I \) is leaked through \( t_j \).

Observation via total miss count. In this scenario, an attacker can observe the number of cache misses of executions [12]. The observer \( O : \Sigma^* \rightarrow \mathbb{N} \) 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 \( t \).

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:

\[
\bigvee_{e \in \text{Paths}} \left( \Gamma(pce) \land \sum_{i \in [1, n_e]} \text{miss}_i = O(t_j) \right) \land \pi \tag{20}
\]

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 (20) is unsatisfiable, we can establish that the information \( \neg \pi \equiv \text{true} \) is leaked through the execution trace \( t_j \). By performing such unsatisfiability checks over the entire program input space, we quantify the information leak \( L(t_j) \) through execution trace \( t_j \) (cf. Section 3.3).

**Observation via hit/miss sequence.** For an execution trace \( t \in \Sigma^* \), an observer can monitor hit/miss sequences from \( t \) [5]. 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: \( O(t) = 1 \) if \( t_{o_i} = m \) and \( O(t) = 0 \) otherwise. \( O(t) \) captures the \( i \)-th bit of \( O(t) \) and similarly, \( t_{o_i} \) captures the \( o_i \)-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:

\[
\bigvee_{e \in \text{Paths}} \left( \Gamma(pce) \land \bigwedge_{i \in [1, 2, \ldots, k]} \left( \neg o_i \leq n_e \right) \right) \land \pi \tag{21}
\]

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 \( L(t_j) \) (cf. Section 3.3). In Constraint (21), our general information leak checker in Constraint (8) is instantiated with \( \Phi_{O, e} \) being set to \( \bigwedge_{i \in [1, 2, \ldots, k]} \left( \neg o_i = O(t_{o_i}) \right) \).

In general, CHALICE can be instantiated for any observer model expressed via symbolic constraints over \( miss_i \).

6 EVALUATION

**Experimental setup.** We implemented CHALICE on top of the KLEE symbolic virtual machine [3]. We have engineered KLEE to symbolically execute the PISA [7] binary code – a MIPS like architecture. This is because, cache performance is captured accurately only in the executable binary code. To evaluate the effectiveness of CHALICE, we have chosen cryptographic applications from the OpenSSL library [1] and other software repositories [4], as well as applications from the Linux GDK library (cf. Table 1). The choice of our programs is motivated by the critical importance of validating security-related properties in these applications. We have performed all experiments on an Intel i7 machine with 8GB of RAM and running a Debian operating system.

<table>
<thead>
<tr>
<th>Program</th>
<th>Lines of C code</th>
<th>Lines of MIPS code (disassembled version)</th>
<th>Input size</th>
<th>Max. #Memory access</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES [4]</td>
<td>800</td>
<td>4842</td>
<td>16 bytes</td>
<td>2134</td>
</tr>
<tr>
<td>AES (1)</td>
<td>1432</td>
<td>1700</td>
<td>16 bytes</td>
<td>420</td>
</tr>
<tr>
<td>DES [4]</td>
<td>552</td>
<td>3430</td>
<td>8 bytes</td>
<td>334</td>
</tr>
<tr>
<td>RC4 [4]</td>
<td>169</td>
<td>660</td>
<td>8 bytes</td>
<td>1538</td>
</tr>
<tr>
<td>RC4 (1)</td>
<td>1745</td>
<td>1140</td>
<td>16 bytes</td>
<td>710</td>
</tr>
<tr>
<td>keyval_to_unicode (GDK)</td>
<td>1195</td>
<td>248</td>
<td>8 bytes</td>
<td>114</td>
</tr>
<tr>
<td>keyval_name (GDK)</td>
<td>1195</td>
<td>1400</td>
<td>4 bytes</td>
<td>12</td>
</tr>
</tbody>
</table>

**Table 1:** Salient features of the evaluated subject programs

**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, we assume that the entire input is sensitive and we make all input bits to be symbolic.
Let us assume an arbitrary N-byte program input $k$. We sample $k$ into $N$ equal segments and use $k_i$ to capture the $i$-th segment. We generate the following predicates on inputs for quantifying information leak $\mathcal{L}(t)$ (cf. Section 3.3):

$$

\pi_{\text{bit}} = \{ k_i = v \mid i \in [1, N], v \in \{0, 1\} \}
$$

$$

\pi_{\text{byte}} = \{ k_i = v \mid i \in [1, \frac{N}{8}], v \in \{0, 255\} \}
$$

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

**Key Results.** Table 2 outlines the key results obtained from CHALICE. For all evaluations in Table 2, we used an 8 KB direct-mapped cache with a line size of 32 bytes (refer to the extended version [18] for detailed experiments with different cache configurations). For a randomly generated execution, Table 2 documents the quantified information leak with respect to predicates $\pi_{\text{bit}}$ and $\pi_{\text{byte}}$. We make the following observations from Table 2. For all scenarios, $\mathcal{L}(t)$ is zero when predicates $\pi_{\text{bit}}$ is used. Therefore, we can prove the absence of any dependency between the cache performance (i.e. the number of cache miss or hit/miss sequence) and the value of an arbitrary bit of the key, for all the observations in Table 2. However, we observe the presence of substantial leak with respect to $\pi_{\text{byte}}$, when the number of cache misses were observed. For instance, we established that as many as 251 values (out of 256) are leaked for each byte of the AES key (in the implementation [4]). This means, there exist at least 251 values (out of a total $2^{128}$) 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 [4], we can observe from Table 2 that the implementation of AES from OpenSSL exhibits substantially fewer information leaks. DES exhibits severe cache side-channel leakage when the adversaries observe cache misses, whereas RC4 exhibits lower leakage as compared to AES and DES, with respect to the respective observation.

<table>
<thead>
<tr>
<th>Subject Program</th>
<th>Observation via total miss count w.r.t. $\pi_{\text{bit}}$</th>
<th>Observation via total miss count w.r.t. $\pi_{\text{byte}}$</th>
<th>Observation via hit/miss of an arbitrary access w.r.t. $\pi_{\text{bit}}$</th>
<th>Observation via hit/miss of an arbitrary access w.r.t. $\pi_{\text{byte}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES [4]</td>
<td>0 $\approx 2^{16}$</td>
<td>0 $\approx 2^{16}$</td>
<td>0 $\approx 2^{16}$</td>
<td>0 $\approx 2^{16}$</td>
</tr>
<tr>
<td>AES [1]</td>
<td>0 $\approx 2^{16}$</td>
<td>0 $\approx 2^{16}$</td>
<td>0 $\approx 2^{16}$</td>
<td>0 $\approx 2^{16}$</td>
</tr>
<tr>
<td>DES [1]</td>
<td>0 $\approx 2^{24}$</td>
<td>0 $\approx 2^{24}$</td>
<td>0 $\approx 2^{24}$</td>
<td>0 $\approx 2^{24}$</td>
</tr>
<tr>
<td>RC4 [1]</td>
<td>0 $\approx 2^{24}$</td>
<td>0 $\approx 2^{24}$</td>
<td>0 $\approx 2^{24}$</td>
<td>0 $\approx 2^{24}$</td>
</tr>
<tr>
<td>GDB Library</td>
<td>0 $\approx 2^{24}$</td>
<td>0 $\approx 2^{24}$</td>
<td>0 $\approx 2^{24}$</td>
<td>0 $\approx 2^{24}$</td>
</tr>
</tbody>
</table>

Table 2: Leakage $\mathcal{L}(t)$ quantified w.r.t. predicates $\pi_{\text{bit}}$ and $\pi_{\text{byte}}$

We also investigated on adversaries who can observe the sequence of cache hits and misses, instead of just the overall number of cache misses. 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. Both AES and DES exhibit substantial leakage with respect to this adversary. RC4, in contrast, exhibits less leakage in the respective observation, as shown in Table 2.

For RC5, CHALICE did not report any symbolic memory address or symbolic branch conditions. Hence, the cache performance of RC5 is unrelated to input and we leverage this report to verify the absence of cache side-channel leak in RC5, with respect to the observer models studied in CHALICE.

**Experiments with set-associative caches.** In this section, we include the evaluation report for set-associative caches employing least-recently-used (LRU) replacement policy. Due to space constraints, we only report the evaluation for AES implementations [1, 4] and for miss-count based observer model. A detailed evaluation on all the subject programs can be found in an extended version of this paper [18].

In Figures 5(a)-(b), we outline the number of values leaked per byte of the sensitive input. Hence, the horizontal axis in these figures capture the individual bytes of the sensitive input space. For instance, consider the evaluation shown in Figure 5 for 2-way, 8 KB cache. In this scenario, the AES implementation [4] leaks 212 values per byte of the secret key, for certain observations. This means, for the respective set of observations, a potential attacker can eliminate the possibility of at least $2^{1216}$ possible keys. In other words, at most $44^{16}$ keys are possible for the respective observation using a 2-way, 8 KB cache.

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 $I_{\text{EC}} \subseteq I_{\text{LC}} \cup I_{\text{MC}} \cup I_{\text{SC}} \cup I_{\text{DC}}$ (where $I_{\text{EC}} \cup I_{\text{LC}} \cup I_{\text{MC}}$ is the entire input space) which leads to C cache misses. Increasing cache size reduces cache conflict. Therefore, it is possible that some input $i \in I_{\text{LC}}$, 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 $i \in I_{\text{EC}}$ 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 5(a), the reduction in cache side-channel leakage is visible for cache sizes up to 16 KB, for certain AES implementation [4]. 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.

**Analysis sensitivity w.r.t. observation.** Since we quantify information leak from execution trace $t$, the leakage $\mathcal{L}(t)$ depends on the observed cache behaviour. For observer models that record the hit/miss statistics of an arbitrary access, we compute $\mathcal{L}(t)$ for an arbitrary memory access to be a cache miss or a cache hit. For observer models based on total miss count, the computed leakage $\mathcal{L}(t)$ depends on the observed miss count. Figure 6 captures the information leakage with respect to observed miss count. Although we observe that $\mathcal{L}(t)$ mostly decreases with increased miss count, there is no direct correlation between the observed miss count and the computed leakage. In Figure 6, the distribution of information leakage is similar to the tail of the essentially gaussian distribution captured in Figure 1.

**Analysis Time.** Table 3 outlines the analysis time for a direct-mapped 8 KB cache. In most cases, a single call to the solver, which
reports information leak via unsatisfiability check (e.g. via checking Constraint (8)), is efficient. Due to the repeated calls to solver, checking the information leakage, for the entire input space, takes significant time. However, since CHALICE incorporates an anytime strategy, the bounds on the quantification of $\ell(t_f)$ are valid for any explored subsets of the program paths and input space. Moreover, the size of our symbolic encoding does not vary significantly with respect to the cache size (see the extended version [18]). Finally, the performance of CHALICE can be improved drastically if we assign one or more independent threads to check information leaked about each input byte. We plan to implement this in the future.

Discussion. In our evaluation, we observe that CHALICE generally reports higher information leakage when miss count was observed, as compared to observing the cache behaviour of an arbitrary access. This is expected, as cache behaviour of a memory access, in general, also affects the total miss count. However, if cache behaviours of a pair of memory accesses are inversely correlated (e.g. one being a cache hit and another being a cache miss for any input), then such accesses do not affect the total miss count. Yet, these accesses may leak information when their cache behaviours were observed individually. In our experiments, such a phenomenon was discovered for RC4.

For the sake of brevity, we have only presented the quantification of information leak discovered via CHALICE. 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. Potential debugging strategies will be to restructure the code, selectively bypassing the cache or using software-controlled memory for certain memory accesses.

7 RELATED WORK

Approaches based on static analysis [8, 22, 27, 28] are incapable to highlight critical scenarios when a particular observation leaks substantially more information than the rest [13]. CHALICE quantifies information leak from execution traces and it does not suffer from the aforementioned limitation. Moreover, CHALICE targets arbitrary software binaries and it is not limited to the verification of constant-time cryptographic software [6, 10]. Existing work based on symbolic execution [29] quantifies side-channel leakage via counting the number of observations [22, 28], and it ignores the effect of micro-architectural entities, such as caches.

CHALICE is complimentary to our recently proposed approach CATAPULT [17], which obtains a coverage of all possible cache behaviour via symbolic execution. In contrast to CATAPULT, our goal is to quantify the leakage of information for a given cache behaviour and we instantiate our framework via a novel and efficient symbolic model of direct-mapped caches.

The observer models used in this paper are based on existing cache attacks [5, 12]. However, we believe that CHALICE is generic to incorporate more advanced attack scenarios [15, 25, 26], as long as the attacks are expressed via the intuition given in Section 5. CHALICE is orthogonal to approaches proposing countermeasures
to thwart side-channel attacks [20, 31]. Of course, CHALICE can be used to validate countermeasureings mitigating cache side channels.

In summary, we propose a new approach to quantify cache side-channel leakage from execution traces and 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 discover by competitive static or logical analysis.

8 CONCLUDING REMARKS

Threats to validity. In CHALICE, we assume an attacker model where the cache architecture (i.e. the number of cache sets, line size, associativity and replacement policy) is known to the adversary. We also assume that the adversary can clearly distinguish the execution profile of victim software. In practice, however, an adversary may not accurately know the cache architecture or the execution profile of the victim software. Hence, she may not be able to retrieve as much information as computed via CHALICE. Since CHALICE is aimed for security testing, we believe that threats involving a strong attacker model is justified. CHALICE currently does not handle concurrent adversaries that aim to retrieve information by investigating cache states [26]. However, our powerful symbolic reasoning framework allows us to consider such adversaries in a future extension of CHALICE.

Perspective. In this paper, we have shown that the mechanism of CHALICE is highly desirable for quantifying the amount of information that can leak through memory performance. Besides security testing, CHALICE can be used to discover bugs while writing constant-time cryptographic applications. We demonstrate the usage of CHALICE to highlight critical information leak scenarios in OpenSSL and Linux GDK libraries, among others.

CHALICE provides a platform to lift the state-of-the-art in security testing via detecting and quantifying side-channel vulnerabilities. We envision to extend CHALICE for side channels beyond 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 also influence the activities in testing software regressions.

Acknowledgments. We thank the anonymous reviewers, Ke Jiang, Arian Maghazeh and Ezekiel Olamide Soremekun for their insightful comments. This work is partially supported by CENIT organization (project 12.04) and SUTD research grant SRIS17123.

REFERENCES