Quantifying the Information Leakage in Cache Attacks via Symbolic Execution

SUDIPTA CHATTOPADHYAY, Singapore University of Technology and Design
MORITZ BECK, Saarland University
AHMED REZINE, Linköping University
ANDREAS ZELLER, Saarland University

Cache 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 leakage that can highlight critical cache side-channel leakage on arbitrary binary code. In our evaluation on real-world programs from OpenSSL and Linux GDK libraries, CHALICE effectively quantifies information leakage: 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**;

Additional Key Words and Phrases: Side channel, Security, Cache, Symbolic execution

ACM Reference Format:
Sudipta Chattopadhyay, Moritz Beck, Ahmed Rezine, and Andreas Zeller. 2017. Quantifying the Information Leakage in Cache Attacks via Symbolic Execution. 1, 1, Article 1 (October 2017), 25 pages.
https://doi.org/10.1145/3127041.3127044

1 INTRODUCTION

Cache attacks [11] are among the best known *side channel* attacks [26] to determine secret features of a program execution without knowing its input or output. Cache attacks can be timing-based [11] or access-based [30]. 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. Similarly, for access-based cache attacks, an observer monitors, for a known program, the specific cache lines being accessed. Then, such information is used to determine the input processed by the respective program. Recent cache attacks [33] show that access-based attacks are practical even in commodity embedded processors, such as in ARM-based embedded systems.

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

XXXX-XXXX/2017/10-ART1 $15.00
https://doi.org/10.1145/3127041.3127044

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 through timing attacks. For access-based attacks, CHALICE symbolically computes the number of cache lines accessed in each cache set. This is, then, used to derive which bits of encryption key may leak through an access-based cache attack.

The reason why CHALICE works is that the timings of cache hits and misses, as well as the cache access patterns, are not uniformly distributed; and therefore, some specific timings and cache access patterns may reveal more information than others. Figure 1 demonstrates the execution of an AES-128 implementation [3] 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.

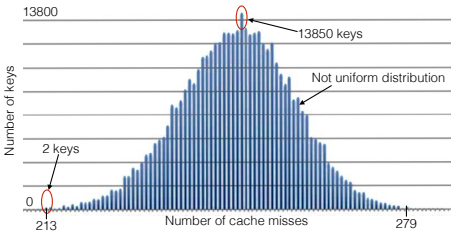


Fig. 1. For a fixed input message, the plot shows the number of keys leading to a given number of cache misses incurred by executing AES-128 encryption (sample size = 256000 keys)

It is this *precision of its symbolic analysis* that sets CHALICE apart from the state of the art. Existing works [23, 32] 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 leakage 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 on CHALICE (section 2), we make the following contributions:

(1) We present CHALICE, *a new approach to precisely quantify information leakage 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).

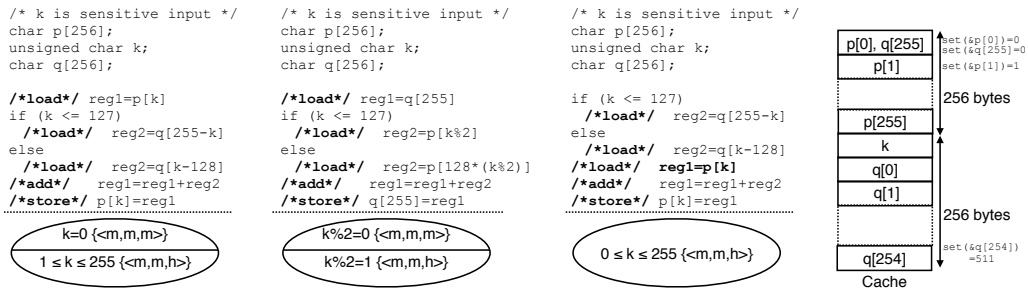


Fig. 2. k is a sensitive input. (a)-(c) three code fragments and respective partitions of the input space with respect to cache hit/miss sequence ($reg1$, $reg2$ represent registers), (d) mapping of program variables into a 512-byte direct-mapped cache ($q[255]$ and $p[0]$ conflict in the cache)

(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 leakage can be as high as 127 bits (out of 128 bits) for certain implementations [3] 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 leakage 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 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 = $\langle m, m, m\rangle$) and $k \in [1, 255]$ (hit/miss sequence = $\langle m, m, h\rangle$). A similar exercise for the code in Figure 2(b) results in the following partition of the sensitive input space: $k \in [0, 255] \wedge (k \ mod \ 2 = 0)$ (hit/miss sequence = $\langle m, m, m\rangle$) and $k \in [0, 255] \wedge (k \ mod \ 2 \neq 0)$ (hit/miss sequence = $\langle m, m, h\rangle$).

Key observation. In this work, we stress the importance of *quantifying information leakage 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 $\langle m, m, m\rangle$, 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 $\langle m, m, m\rangle$, 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 $^1/_{128}$ 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 $\langle m, m, m\rangle$ for Figure 2(a)). In order to fix the cache side-channel leakage in Figure 2(a), we can reorder the code as shown in Figure 2(c).

Limitations of static analysis. Existing works in static analysis [23, 32] 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 [23, 32] will consider these two code fragments *equivalent* in terms of cache side-channel leakage. As a result, a crucial information leakage 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 [7, 9]. 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.

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] [10], we have shown how such an effective test suite can be generated automatically. Since the observation by an attacker (*e.g.* number of cache misses) 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* 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 leakage.

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 $t_I \equiv \langle m, m, m\rangle$. *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:

$$(0 \leq k \leq 127) \wedge (set(\&p[k]) \neq set(\&q[255 - k])) \wedge (set(\&p[k]) \neq set(\&p[k])) \quad (1)$$

where $set(\&x)$ 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:

$$(0 \leq k \leq 127) \wedge (set(\&p[k]) = set(\&q[255-k])) \wedge (tag(\&p[k]) \neq tag(\&q[255-k])) \quad (2)$$

where $tag(\&x)$ 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:

$$(128 \leq k \leq 255) \wedge (set(\&p[k]) = set(\&q[k-128])) \wedge (tag(\&p[k]) \neq tag(\&q[k-128])) \quad (3)$$

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 $128 \leq k \leq 255$. As a result, the second access to $p[k]$ is not a cache miss for any input $k \in [128, 255]$.

From the observation $\langle m, m, m \rangle$, 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 $k \in [128, 255]$. 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 accessed from 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 $\langle m, m, m \rangle$. In other words, `CHALICE` accurately reports the information leakage (*i.e.* $k = 0$) for the observation $\langle m, m, m \rangle$.

`CHALICE` for debugging. In the preceding discussion, we observed that `CHALICE` reported accurate information leakage by checking the cache miss behaviour of the `store` instruction only. Consequently, a developer can invest more attention in fixing the leakage through the `store` instruction, such as forcing the store to be a cache hit for all inputs. As observed in Figure 4(c), such a fix involves moving the first `load` instruction in Figure 4(a). We note, however, that `CHALICE` does not provide capabilities to automatically fix cache side channels.

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 $\langle m, m, m \rangle$, then the uncertainty to guess k is 0 bits (as exactly one value of k is possible for this 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 $\langle m, m, m \rangle$ was observed by the attacker.

3 FRAMEWORK

In this section, we formally introduce the core capabilities implemented within `CHALICE`.

3.1 Foundation

3.1.1 Threat model. Side-channel attacks are broadly classified into synchronous and asynchronous attacks [41]. 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 [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 misses) 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 $\mathcal{P}$ on input I results in an execution trace t_I. t_I 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_I. We capture this quantification via $\mathcal{L}(t_I)$. 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 $\langle m, h, m, h, h\rangle$ and $\langle m, m, h, h, h\rangle$ to 2. It will therefore not differentiate them. The most precise observer would be the identity mapping on Σ^*. However, an observer that tracks prefixes of length two would be capable of differentiating $\langle m, h, m, h, h\rangle$ and $\langle m, m, h, h, h\rangle$.

We use the 0-1 variable $miss_i$ to capture the cache miss behaviour of the i-th memory access. 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 Φ_O. Φ_O is a symbolic constraint over variables $\{miss_1, miss_2, \ldots, miss_n\}$. For instance, $\Phi_O \equiv \left(\sum_{i=1}^{n} miss_i = 100\right)$ 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 Φ_O along a program path e. For example, $\Phi_{O,e} \equiv \left(\sum_{i=1}^{300} miss_i = 100\right)$ if $\Phi_O \equiv \left(\sum_{i=1}^{n} miss_i = 100\right)$ and the path e has 300 memory accesses. $\Phi_{O,e}$ amounts to $false$ if Φ_O requires a different number of memory accesses than those provided by the path e. Given only Φ_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 leakage detection is to capture the cache behaviour via symbolic constraints. Let us consider a set of inputs $\mathbb{I}$ that exercise the same execution path with n memory accesses. We use $\Gamma(\mathbb{I})$ to accurately encode all possible combinations of values of variables $\{miss_1, miss_2, \ldots, miss_n\}$. Therefore, if $\Gamma(\mathbb{I}) \wedge \Phi_O$ is *unsatisfiable*, we can deduce that the respective observation Φ_O *did not occur* for any input $I \in \mathbb{I}$. We now describe how $\mathcal{L}(t_I)$ is computed.

3.2 Quantifying Information Leakage in Execution

Figure 3 provides an outline of our entire framework. We symbolically execute a program $\mathcal{P}$ and compute the path condition [27] 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 behaviours.

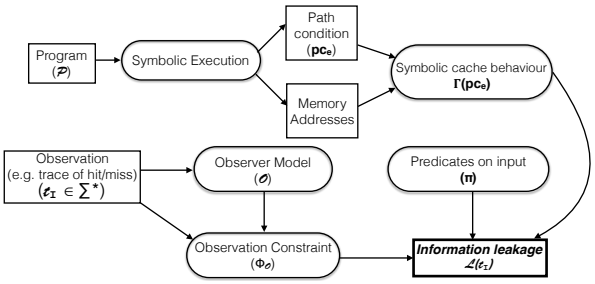


Fig. 3. The framework CHALICE

Recall that we use $\Gamma(\mathbb{I})$ to capture possible cache hit/miss sequences in an execution path, which was activated by a set of inputs $\mathbb{I}$. In an abuse of notation, we capture the set of inputs $\mathbb{I}$ 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 `If` 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 behaviour via variables $miss_i$. $miss_i$ is set to 1 (resp. 0) if and only if the i-th memory access along the path encounters a cache miss (resp. hit). Given n to be the total number of memory accesses along the path e, we formulate $\Gamma(pc_e)$ to bound the value of $\{miss_1, miss_2, \ldots, miss_n\}$. In particular, any solution of $\Gamma(pc_e) \wedge (miss_i = 1)$ captures a concrete input $I \models pc_e$ and such an input I leads to an execution where the i-th memory access is a cache miss. Therefore, if an observation Φ_O happens to be for input $I \models pc_e$, then $\Gamma(pc_e) \wedge \Phi_{O,e}$ is always satisfiable. Conclusively, we capture the information leakage through trace t_I as follows:

$$\mathcal{L}(t_I) = 2^N - \Big| \bigvee_{e \in Paths} \big(\Gamma(pc_e) \wedge \Phi_{O,e}\big) \Big|_{sol} \tag{4}$$

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

Relation to entropy. The secrecy of a sensitive input is usually measured as the uncertainty of an attacker to guess its value. This uncertainty can be measured using entropies such as Shannon entropy ($\mathcal{H}$) or min-entropy ($\mathcal{H}_\infty$) [20, 38]. Let $\mathbb{I}$ be the set of secret input values where each input $I \in \mathbb{I}$ has an a priori probability $p(I)$ to be used. The initial uncertainty is given by $\mathcal{H} = -\sum_{I \in \mathbb{I}} p(I) \log_2(p(I))$ for the Shannon entropy and by $\mathcal{H}_\infty = -\log_2(max\{p(I) \mid I \in \mathbb{I}\})$ for the min-entropy. In the particular case where all sensitive inputs are equally probable (i.e., $p(I) = {}^1/_{|\mathbb{I}|}$ for each $I \in \mathbb{I}$), then both measures coincide ($\mathcal{H} = \mathcal{H}_\infty = \log_2 |\mathbb{I}|$).

Assume the observations are completely determined by the sensitive input. Suppose the attacker makes an observation O. Let $\mathbb{I}_O$ be the set of all sensitive inputs in $\mathbb{I}$ leading to the observation O. Let $p_O(I)$ be the probability that the sensitive input is I given the behaviour O is observed. Notice that $p_O(I) = {}^1/_{|\mathbb{I}_O|}$ in case sensitive inputs are equally probable. The remaining uncertainty is given by $\mathcal{H}(O) = -\sum_{I \in \mathbb{I}} p_O(I) \log_2(p_O(I))$ using the Shannon entropy and by $\mathcal{H}_\infty(O) = -\log_2(max\{p_O(I) \mid I \in \mathbb{I}\})$ using the min-entropy. These two measures coincide again for equally probable sensitive inputs ($\mathcal{H}(O) = \mathcal{H}_\infty(O) = \log_2 |\mathbb{I}_O|$). Observe that $\mathbb{I}_O$ is the set of all program inputs for which $\bigvee_{e \in Paths} \big(\Gamma(pc_e) \wedge \Phi_{O,e}\big)$ evaluates to true. Hence, if the attacker assumes that all values of the sensitive input are equally probable for observation constraint Φ_O, we get

$$\mathcal{H}(O) = \log_2 \Big| \bigvee_{e \in Paths} \big(\Gamma(pc_e) \wedge \Phi_{O,e}\big) \Big|_{sol} \tag{5}$$

In summary, less the number of satisfying solutions for the formula $\bigvee_{e \in Paths} \left(\Gamma(pc_e) \wedge \Phi_{O,e}\right)$, less the uncertainty to guess the sensitive input I given O is observed.

Example 3.1. Assume a program that takes a 4 bits long sensitive input. Suppose an attacker can observe three possible values: O_1 resulting from 12 input values $\mathbb{I}_{O_1}$, O_2 resulting from 3 input values $\mathbb{I}_{O_2}$, and O_3 resulting from a single input value $\mathbb{I}_{O_1}$. Assuming all input values are equally probable, the initial uncertainty is given by $\mathcal{H} = \log_2(2^4) = 4$ bits. The remaining uncertainty after an attacker observes O_1 is $\mathcal{H}(O_1) = \log_2(|\mathbb{I}_{O_1}|) = \log_2 12 = 3.58$, after observing O_2 is $\mathcal{H}(O_2) = \log_2(|\mathbb{I}_{O_2}|) = \log_2(3) = 1.58$ and after observing O_3 is $\mathcal{H}(O_3) = \log_2(|\mathbb{I}_{O_3}|) = \log_2 1 = 0$. The corresponding information leakage is therefore of $4 - 3.58 = 0.42$ bits for O_1, of $4 - 1.58 = 2.42$ bits for O_2, and of $4 - 0 = 4$ bits for O_3.

Given an observation O, our analysis gives sound upper bounds to $|\mathbb{I}_O|$ for a rich set of observations of cache behaviours. The longer we run the analysis, the tighter (i.e. smaller) upper bound we obtain. The smaller the upper bounds, the more information is established to be leaked by the given observation. Our analysis can for example be used to exclude adopting some sensitive inputs resulting in an observation O with a particularly small $|\mathbb{I}_O|$, i.e., leaking too much information.

In the next section, we show how to compute the number of solutions of $\bigvee_{e \in Paths} \left(\Gamma(pc_e) \wedge \Phi_{O,e}\right)$ (i.e., $|\mathbb{I}_O|$) in an incremental fashion.

3.3 Cartesian Bounding of Information Leakage

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 $\mathcal{L}(t_I)$. Computing the exact value of $\mathcal{L}(t_I)$ 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 [13] supporting non-linear constraints. Specifically, we evaluated a scalable, but approximate model counter [16] and discovered it is *incapable* of dealing 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. Intuitively, while evaluating leakage through an observation, we check for each input byte (bit, respectively) whether the byte can hold a specific value between 0 and 255 (0 or 1, respectively) via predicates π_{byte} (π_{bit}, respectively). This allows us to trade accuracy for the number of required solver calls. The π_{byte} and π_{bit} cases are clarified after lemma 3.2 in Section 3.3.2. We formalize a generalization of this idea. This allows us to use arbitrary predicates (not only π_{byte} and π_{bit}). Our formalization reflects on some valuable properties in terms of improving the accuracy of quantified leakage while requiring fewer solver calls. In addition, our proposed approach is inherently parallel, as each partition (resulting from predicate choices) can be checked independently. 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 checkers or weak keys in encryption routines. Finally, our proposed scheme also provides *strong guarantees* on the derived bound for $\mathcal{L}(t_I)$.

3.3.2 Input space partitioning to compute $\mathcal{L}(t_I)$. Consider a set $\mathbb{I}$ of program inputs containing all possible N-bit input values, i.e., $|\mathbb{I}| = 2^N$. A partition P of $\mathbb{I}$ is a set $\{\mathsf{P}[j] \mid 1 \leq j \leq |\mathsf{P}|\}$ of disjoint non-empty sets whose union coincides with $\mathbb{I}$. Here, we write $|\mathsf{P}|$ to mean the size of P, i.e. the number of subsets of $\mathbb{I}$ 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 $\mathsf{P}_1, \ldots, \mathsf{P}_K$ of the input set $\mathbb{I}$ for which no $(\mathsf{P}_1[i_1] \cap \mathsf{P}_2[i_2] \cap \ldots \cap \mathsf{P}_K[i_K])$ is empty, for any $i_j : 1 \leq i_j \leq |\mathsf{P}_j|$. A $\{\mathsf{P}_1, \ldots, \mathsf{P}_K\}$-based Cartesian partitioning of $\mathbb{I}$, written $(\mathsf{P}_1 \boxtimes \mathsf{P}_2 \boxtimes \ldots \boxtimes \mathsf{P}_K)$, is the partition of $\mathbb{I}$ that corresponds to the intersection of all partitions $\mathsf{P}_1, \ldots, \mathsf{P}_K$, i.e, whose elements are the sets $(\mathsf{P}_1[i_1] \cap \mathsf{P}_2[i_2] \cap \cdots \cap \mathsf{P}_K[i_K])$ where $i_j : 1 \leq i_j \leq |\mathsf{P}_j|$. For each tuple $(\mathsf{P}_1[i_1], \ldots, \mathsf{P}_K[i_K])$ of the cross product $(\mathsf{P}_1 \times \ldots \times \mathsf{P}_K)$, we write $[\![(\mathsf{P}_1[i_1], \ldots, \mathsf{P}_K[i_K])]\!]$ to mean the element $(\mathsf{P}_1[i_1] \cap \mathsf{P}_2[i_2] \cap \ldots \cap \mathsf{P}_K[i_K])$ of the Cartesian partitioning $(\mathsf{P}_1 \boxtimes \ldots \boxtimes \mathsf{P}_K)$. For a subset T of the cross product $(\mathsf{P}_1 \times \mathsf{P}_2 \times \ldots \times \mathsf{P}_K)$, we let $[\![T]\!]$ mean the union $\cup_{t \in T}[\![t]\!]$. A Cartesian partitioning $(\mathsf{P}_1 \boxtimes \ldots \boxtimes \mathsf{P}_K)$ is said to be *complete* if each $[\![(\mathsf{P}_1[i_1], \ldots, \mathsf{P}_K[i_K])]\!]$ is a singleton of $\mathbb{I}$. Observe that this means $|\mathbb{I}| = 2^N = |\mathsf{P}_1| \times \ldots \times |\mathsf{P}_K|$ holds. Given a subset S of $\mathbb{I}$ and a Cartesian partitioning $(\mathsf{P}_1 \boxtimes \ldots \boxtimes \mathsf{P}_K)$ of $\mathbb{I}$, we write $(S)_{|(\mathsf{P}_1 \boxtimes \ldots \boxtimes \mathsf{P}_K)}$ to mean the set of elements of $(\mathsf{P}_1 \boxtimes \ldots \boxtimes \mathsf{P}_K)$ whose denotations intersects S. Observe that $S = [\![(S)_{|(\mathsf{P}_1 \boxtimes \ldots \boxtimes \mathsf{P}_K)}]\!]$ in case $(\mathsf{P}_1 \boxtimes \ldots \boxtimes \mathsf{P}_K)$ is complete. The following lemma bounds information leakage by requiring only $\Sigma_{i:1 \leq i \leq K}|\mathsf{P}_i|$ solver calls (as opposed to $2^N = \Pi_{i:1 \leq i \leq K}|\mathsf{P}_i|$ when enumerating all inputs):

Lemma 3.2 (Cartesian leakage bound). *Assume a complete Cartesian partitioning $(\mathsf{P}_1 \boxtimes \ldots \boxtimes \mathsf{P}_K)$ of $\mathbb{I}$ and a trace t_I that results in the observation constraint Φ_O. If $\mathsf{U}^{\Phi_O}_{\mathsf{P}_i} \subseteq \mathsf{P}_i$ is the set of P_i elements for which Φ_O is unfeasible, then $\mathcal{L}(t_I) \geq 2^N - \prod_{1 \leq i \leq K}\left(|\mathsf{P}_i| - |\mathsf{U}^{\Phi_O}_{\mathsf{P}_i}|\right)$.*

Proof. Recall $\mathcal{L}(t_I) = 2^N - |\bigvee_{e \in Paths}\left(\Gamma(pc_e) \wedge \Phi_{O,e}\right)|_{sol}$. Observe $|\bigvee_{e \in Paths}\left(\Gamma(pc_e) \wedge \Phi_{O,e}\right)|_{sol}$ is the size of the set S_{Φ_O} of all program inputs I that exhibit the observation Φ_O, i.e., satisfying some path condition pc_e where $(\Gamma(I) \wedge \Phi_{O,e})$ holds. Observe that S_{Φ_O}, which coincides with the denotation of $\left(S_{\Phi_O}\right)_{|(\mathsf{P}_1 \boxtimes \ldots \boxtimes \mathsf{P}_K)}$, is included in the denotation of $\left(\left(S_{\Phi_O}\right)_{|\mathsf{P}_1} \times \left(S_{\Phi_O}\right)_{|\mathsf{P}_2} \times \ldots \times \left(S_{\Phi_O}\right)_{|\mathsf{P}_K}\right)$. Hence, $|S_{\Phi_O}| \leq \Pi_{i:1 \leq i \leq K}\left|\left(S_{\Phi_O}\right)_{|\mathsf{P}_i}\right|$. We conclude by observing that $\left(S_{\Phi_O}\right)_{|\mathsf{P}_i} = \mathsf{P}_i \setminus \mathsf{U}^{\Phi_O}_{\mathsf{P}_i}$ since $\mathsf{U}^{\Phi_O}_{\mathsf{P}_i}$ is the subset of P_i for which Φ_O is unfeasible. □

Lemma 3.2 holds for any complete partitioning of the set of inputs. In practice, we can generate the K partitions by sampling the N-bit input into K equal segments. We then constrain the search space of the solver by restricting the value of each such input segment. For instance, let us assume k is the program input and k_i captures the i-th input segment. The i-th partition is defined using $2^{\frac{N}{K}}$ predicates $\pi_i[v] \equiv (k_i = v)$ for $v \in \left\{0, 1, \ldots, 2^{\frac{N}{K}} - 1\right\}$. For a segment i, the predicates in $\{\pi_i[v] \mid 0 \leq v < 2^{\frac{N}{K}}\}$ are pairwise unsatisfiable and partition all input values into $2^{\frac{N}{K}}$ elements. The obtained Cartesian partitioning is complete. We use each $\pi_i[v]$ to guide the solver and search for a solution only in the input space where the i-th input segment is v. Since, we have K different segments, we generate a total of $\left(K \cdot 2^{\frac{N}{K}}\right)$ different predicates, each characterizing an element of some partition. An appealing feature of this process is that all $K \cdot 2^{\frac{N}{K}}$ predicates can be generated independently and result in parallelizable unsatisfiability checks. Given a partition i, we compute $\mathsf{U}^{\Phi_O}_{\mathsf{P}_i}$ as the number of predicates $\pi_i[v]$ for which the following is *unsatisfiable*:

$$\bigvee_{e \in Paths}\left(\Gamma(pc_e) \wedge \Phi_{O,e} \wedge \pi_i[v]\right) \tag{6}$$

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 leakage 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 $\mathbb{I}$, $(\mathsf{P}_1 \boxtimes \ldots \boxtimes \mathsf{P}_K)$ and $(\mathsf{Q}_1 \boxtimes \ldots \boxtimes \mathsf{Q}_M)$. We say that $(\mathsf{P}_1 \boxtimes \ldots \boxtimes \mathsf{P}_K)$ refines, or is more precise than, $(\mathsf{Q}_1 \boxtimes \ldots \boxtimes \mathsf{Q}_M)$ if there is a surjective function $h : \{1, \ldots, M\} \rightarrow \{1, \ldots, K\}$ such that each partition P_i, for $i : 1 \leq i \leq K$, coincides with the Cartesian partitioning $(\mathsf{Q}_{j_1} \boxtimes \ldots \boxtimes \mathsf{Q}_{j_{|h^{-1}(i)|}})$ where $\left\{j_1, \ldots, j_{|h^{-1}(i)|}\right\} = h^{-1}(i)$.

Lemma 3.3 (Cartesian bound refinement). *Assume two complete Cartesian partitioning of* $\mathbb{I}$ *where* $(\mathsf{P}_1 \boxtimes \ldots \boxtimes \mathsf{P}_K)$ *refines* $(\mathsf{Q}_1 \boxtimes \ldots \boxtimes \mathsf{Q}_M)$. *For any trace* t_I *that results in the observation constraint* Φ_O, *the Cartesian leakage bound obtained with* $(\mathsf{P}_1 \boxtimes \ldots \boxtimes \mathsf{P}_K)$ *is always larger or equal than the one obtained with* $(\mathsf{Q}_1 \boxtimes \ldots \boxtimes \mathsf{Q}_M)$, *i.e.,* $\mathcal{L}(t_I) \geq 2^N - \prod_{1 \leq i \leq K} \left(|\mathsf{P}_i| - |\mathsf{U}_{\mathsf{P}_i}^{\Phi_O}|\right) \geq 2^N - \prod_{1 \leq i \leq M} \left(|\mathsf{Q}_i| - |\mathsf{U}_{\mathsf{Q}_i}^{\Phi_O}|\right)$.

Proof. Let S_{Φ_O} be the subset of program inputs I that exhibits the observation Φ_O, i.e., containing all program inputs I satisfying some path condition pc_e where $(\Gamma(I) \wedge \Phi_{O,e})$ holds. Observe that S_{Φ_O}, which coincides with the denotation of $\left(S_{\Phi_O}\right)_{|(\mathsf{P}_1 \boxtimes \mathsf{P}_2 \boxtimes \ldots \boxtimes \mathsf{P}_K)}$ is subset of the denotation of $\left(\left(S_{\Phi_O}\right)_{|\mathsf{P}_1} \boxtimes \left(S_{\Phi_O}\right)_{|\mathsf{P}_2} \boxtimes \ldots \boxtimes \left(S_{\Phi_O}\right)_{|\mathsf{P}_K}\right)$ which is subset of the denotation of $\left(\left(S_{\Phi_O}\right)_{|\mathsf{Q}_1} \times \left(S_{\Phi_O}\right)_{|\mathsf{Q}_2} \times \ldots \times \left(S_{\Phi_O}\right)_{|\mathsf{Q}_M}\right)$. This leads to the following crucial inequalities: $|S_{\Phi_O}| \leq \Pi_{i:1 \leq i \leq K} \left|\left(S_{\Phi_O}\right)_{|\mathsf{P}_i}\right| \leq \Pi_{i:1 \leq i \leq M} \left|\left(S_{\Phi_O}\right)_{|\mathsf{Q}_i}\right|$. We conclude by observing that $\left(S_{\Phi_O}\right)_{|\mathsf{P}_i} = \mathsf{P}_i \setminus \mathsf{U}_{\mathsf{P}_i}^{\Phi_O}$ since $\mathsf{U}_{\mathsf{P}_i}^{\Phi_O}$ is the subset of P_i for which Φ_O is unfeasible and similarly $\left(S_{\Phi_O}\right)_{|\mathsf{Q}_i} = \mathsf{Q}_i \setminus \mathsf{U}_{\mathsf{P}_i}^{\Phi_O}$. □

Due to the classic path explosion problem in symbolic execution, it is possible that only a subset of execution paths $\mathcal{E} \subseteq \mathit{Paths}$ can be explored within a given time budget. In such cases, we can quantify $\mathcal{L}(t_I)$ as follows.

Lemma 3.4 (Anytime information leakage). *Assume a complete Cartesian partitioning* $(\mathsf{P}_1 \boxtimes \ldots \boxtimes \mathsf{P}_K)$ *of* $\mathbb{I}$ *and a trace* t_I *that results in the observation constraint* Φ_O. *If* $\mathcal{E} \subseteq \mathit{Paths}$, *let* $\mathsf{U}_{\mathsf{P}_i}^{\Phi_O,\mathcal{E}} \subseteq \mathsf{P}_i$ *be the set of* P_i *elements for which the observation constraint* Φ_O *is impossible along the paths* $\mathcal{E}$. *The following holds:*

$$\mathcal{L}(t_I) \geq \left|\bigvee_{e \in \mathcal{E}} pc_e\right|_{sol} - \prod_{1 \leq i \leq K} \left(|\mathsf{P}_i| - |\mathsf{U}_{\mathsf{P}_i}^{\Phi_O,\mathcal{E}}|\right) \tag{7}$$

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 $\left|\bigvee_{e \in \mathcal{E}} pc_e\right|_{sol}$ and $\left|\bigvee_{e \in \mathit{Paths} \setminus \mathcal{E}} pc_e\right|_{sol}$. Similarly, we observe that $\left|\bigvee_{e \in \mathit{Paths}} \left(\Gamma(pc_e) \wedge \Phi_{O,e}\right)\right|_{sol}$ coincides with the sum of $\left|\bigvee_{e \in \mathcal{E}} \left(\Gamma(pc_e) \wedge \Phi_{O,e}\right)\right|_{sol}$ and $\left|\bigvee_{e \in \mathit{Paths} \setminus \mathcal{E}} \left(\Gamma(pc_e) \wedge \Phi_{O,e}\right)\right|_{sol}$. Therefore, information leakage $\mathcal{L}(t_I)$ coincides with: $|\bigvee_{e \in \mathcal{E}} pc_e|_{sol} + |\bigvee_{e \in \mathit{Paths} \setminus \mathcal{E}} pc_e|_{sol} - \left|\bigvee_{e \in \mathcal{E}} \left(\Gamma(pc_e) \wedge \Phi_{O,e}\right)\right|_{sol} - \left|\bigvee_{e \in \mathit{Paths} \setminus \mathcal{E}} \left(\Gamma(pc_e) \wedge \Phi_{O,e}\right)\right|_{sol}$. The result follows from both $|\bigvee_{e \in \mathit{Paths} \setminus \mathcal{E}} pc_e|_{sol} \geq \left|\bigvee_{e \in \mathit{Paths} \setminus \mathcal{E}} \left(\Gamma(pc_e) \wedge \Phi_{O,e}\right)\right|_{sol}$ and $\prod_{1 \leq i \leq K} \left(|\mathsf{P}_i| - |\mathsf{U}_{\mathsf{P}_i}^{\Phi_O,\mathcal{E}}|\right) \geq \left|\bigvee_{e \in \mathcal{E}} \left(\Gamma(pc_e) \wedge \Phi_{O,e}\right)\right|_{sol}$. □

For instance, using the segments defined before, $\mathsf{U}_{\mathsf{P}_i}^{\Phi_O,\mathcal{E}}$ is the number of values v in $0 \leq v < 2^{N/K}$ for which no $\Gamma(pc_e) \wedge \Phi_{O,e} \wedge (k_i = v)$ is satisfiable. Note the term $|\bigvee_{e \in \mathcal{E}} pc_e|_{sol}$ involves only path conditions. $|\bigvee_{e \in \mathcal{E}} pc_e|_{sol}$ can often be computed via model counting [5] in practice.

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

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 symbolic model to encode the performance of *direct-mapped caches* and *set-associative LRU caches*. To describe our model, we shall use the following notations:

- $2^{\mathcal{S}}$: The number of cache lines in the cache.
- $2^{\mathcal{B}}$: The size of a cache line (in bytes).
- $\mathcal{A}$: Associativity of a set-associative cache.
- $set(r_i)$: Cache line accessed by instruction r_i.
- $tag(r_i)$: 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 `If` 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_e and the execution trace Ψ_{pc_e} for each explored path e as follows:

$$\Psi_{pc_e} \equiv \langle (r_1, \sigma_1), (r_2, \sigma_2), \ldots, (r_{n-1}, \sigma_{n-1}), (r_n, \sigma_n) \rangle \tag{8}$$

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.

Modeling Symbolic Cache Access. Following the basic design principle of caches, we compute $set(r_i)$ and $tag(r_i)$ by manipulating the symbolic expression σ_i as follows:

$$set(r_i) = (\sigma_i \gg \mathcal{B}) \ \& \ \left(2^{\mathcal{S}} - 1\right); \ \ tag(r_i) = (\sigma_i \gg (\mathcal{B} + \mathcal{S})) \tag{9}$$

We note that $set(r_i)$ and $tag(r_i)$ might be symbolic expressions due to the symbolic nature of σ_i.

4.1 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* the memory block accessed by 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 such a cache miss *if and only if* the last access to $set(r_i)$ was from an instruction $r_j \in \{r_1, r_2, \ldots, r_{i-1}\}$, such that $tag(r_j) \neq tag(r_i)$.

4.1.1 ***Constraints to formulate cold cache misses.*** If a memory block 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 memory block for the first time during execution. In other words, we can check none of the instructions $r \in \{r_1, r_2, \ldots, r_{i-1}\}$ access the same memory block as r_i. Therefore r_i suffers a cold miss if and only if the following condition holds:

$$\Theta_i^{cold} \equiv \bigwedge_{p \in [1,i)} \left(set(r_p) \neq set(r_i)\right) \vee \left(tag(r_p) \neq tag(r_i)\right) \tag{10}$$

We note that the conditions for cold misses do not depend on the specifics of caches (e.g. direct-mapped or set-associative). In the next section, we show the formulation of eviction misses for direct-mapped and set-associative LRU caches.

4.1.2 ***Constraints to formulate cache evictions.*** In the following, we formulate a set of constraints to encode cache misses due to the eviction of memory blocks from caches.

Direct-mapped caches. In direct-mapped caches, each cache set has exactly one cache line, i.e., $\mathcal{A} = 1$. Therefore, each memory address is mapped to exactly one cache line.

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 the 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:

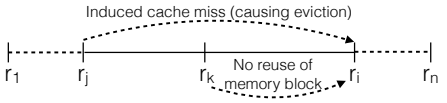


Fig. 4. r_j induces a cache miss at r_i if r_j accesses the same cache set as r_i and r_k does not load the block accessed by r_i

1) $\psi_{cnf}(j, i)$: r_i and r_j access the same cache line,

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

$$\psi_{cnf}\,(j,i) \equiv \big(set(r_j) = set(r_i)\big)\,; \quad \psi_{dif}\,(j,i) \equiv \big(tag(r_j) \neq tag(r_i)\big) \tag{11}$$

3) $\psi_{eqv}(j, i)$: 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 is not responsible for a cache miss at r_i. We formulate the following constraint to capture this condition:

$$\psi_{eqv}\,(j,i) \equiv \bigwedge_{k:\ j<k<i} \big(tag(r_k) \neq tag(r_i) \ \vee \ set(r_k) \neq set(r_i)\big) \tag{12}$$

Constraints (11)-(12) 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 (11)-(12) for any $r \in \{r_1, r_2, \ldots, r_{i-1}\}$. This can be captured via the following constraint:

$$\Theta_i^{emp} \equiv \left(\bigvee_{j:\ 1\leq j<i} \big(\psi_{cnf}\,(j,i) \wedge \psi_{dif}\,(j,i) \wedge \psi_{eqv}\,(j,i)\big) \right) \tag{13}$$

Instruction r_i will not suffer a cache miss due to eviction when, for all prior instructions, at least one of the Constraints (11)-(12) does not hold. This scenario is the negation of Constraint (13) and therefore, it is captured via $\neg\Theta_i^{emp}$.

We use the 0-1 variable $miss_i$ to capture the cache miss behaviour of r_i. As discussed in the preceding paragraphs, r_i suffers a cold miss (*i.e.* satisfying Constraint (10)) or the memory block accessed by r_i would be evicted due to the instructions executed before r_i (*i.e.* satisfying Constraint (13)). Using this notion, we formulate the value of $miss_i$ as follows:

$$\begin{aligned} \Theta_i^{m,dir} &\equiv \left(\left(\Theta_i^{emp} \vee \Theta_i^{cold} \right) \Rightarrow (miss_i = 1) \right) \\ \Theta_i^{h,dir} &\equiv \left(\left(\neg\Theta_i^{emp} \wedge \neg\Theta_i^{cold} \right) \Rightarrow (miss_i = 0) \right) \end{aligned} \tag{14}$$

Set-associative LRU caches. In $\mathcal{A}$-way set-associative caches, each cache set contains $\mathcal{A}$ cache lines. A memory block is mapped to a unique cache set, but it can be located at any of the $\mathcal{A}$ cache lines of the respective set. If the cache set is full and a new memory block mapped to the same cache set is accessed, then the least recently used memory block (LRU) is replaced from the set.

Modeling set-associative caches involves some unique challenges. To illustrate this, let us consider the following sequence of memory accesses in a two-way associative cache with the LRU replacement policy: $(r_1 : m_1) \rightarrow (r_2 : m_2) \rightarrow (r_3 : m_2) \rightarrow (r_4 : m_1)$. We assume both m_1, m_2 are accessed from the same cache set and the cache is empty before r_1 starts execution. We observe

that r_4 will still incur a *cache hit*. This is because r_4 suffers only one cache conflict from the memory block m_2. To incorporate this into our symbolic cache model, we only count cache conflicts from the closest access to a given memory block. Therefore, in our example, we count cache conflicts to r_4 from r_3 and discard the cache conflict from r_2. Formally, we introduce the following additional condition for instruction r_j to inflict a cache conflict to instruction r_i.

$\psi_{unq}(j,i)$: No instruction between r_j and r_i accesses the same memory block as r_j. This is to ensure that r_j is the closest to r_i in terms of accessing the memory block corresponding to the memory address σ_j (*cf.* Equation 8). We capture $\psi_{unq}(j,i)$ formally as follows:

$$\psi_{unq}(j,i) \equiv \bigwedge_{k:\ j<k<i} \left(tag(r_k) \neq tag(r_j) \vee set(r_k) \neq set(r_j)\right) \tag{15}$$

Hence, r_j inflicts a unique cache conflict to r_i only if $\psi_{unq}(j,i)$, $\psi_{eqv}(j,i)$, $\psi_{cnf}(j,i)$ and $\psi_{dif}(j,i)$ are all satisfiable. This is formally captured as follows:

$$\begin{aligned}
\Theta^{emp}_{j,i} &\equiv \left(\bigvee_{j:\ 1\leq j<i} \left(\psi_{cnf}(j,i) \wedge \psi_{dif}(j,i) \wedge \psi_{eqv}(j,i) \wedge \psi_{unq}(j,i)\right)\right) \rightarrow \left(\eta_{ji} = 1\right) \\
\Theta^{ehp}_{j,i} &\equiv \left(\bigwedge_{j:\ 1\leq j<i} \left(\neg\psi_{cnf}(j,i) \vee \neg\psi_{dif}(j,i) \vee \neg\psi_{eqv}(j,i) \vee \neg\psi_{unq}(j,i)\right)\right) \rightarrow \left(\eta_{ji} = 0\right)
\end{aligned} \tag{16}$$

Concretely, η_{ji} is set to 1 if r_j creates a unique cache conflict to r_i and η_{ji} is set to 0 otherwise.

If the number of unique cache conflicts to r_i exceeds the associativity ($\mathcal{A}$) of the cache, then r_i suffers a cache miss due to eviction. Based on this intuition, we formalize $miss_i$ for set-associative LRU caches as follows:

$$\begin{aligned}
\Theta^{m,lru}_{i} &\equiv \left(\sum_{j\in[1,i)} \eta_{ji} \geq \mathcal{A}\right) \vee \Theta^{cold}_{i} \Rightarrow (miss_i = 1) \\
\Theta^{h,lru}_{i} &\equiv \left(\sum_{j\in[1,i)} \eta_{ji} < \mathcal{A}\right) \wedge \neg\Theta^{cold}_{i} \Rightarrow (miss_i = 0)
\end{aligned} \tag{17}$$

We note that $\sum_{j\in[1,i)} \eta_{ji}$ accurately counts the number of unique cache conflicts to the instruction r_i (*cf.* Constraint (16)). Hence, the condition $\left(\sum_{j\in[1,i)} \eta_{ji} \geq \mathcal{A}\right)$ captures whether the memory block accessed by r_i was replaced from the cache before r_i. If r_i does not suffer a cold miss and $\left(\sum_{j\in[1,i)} \eta_{ji} < \mathcal{A}\right)$, then r_i will be a cache hit when executed, as captured by the condition $\Theta^{h,lru}_{i}$.

4.2 Putting it all together

Recall that $\Gamma(pc_e)$ captures the constraint system to encode the cache behaviour for all inputs $I \models pc_e$. In order to construct $\Gamma(pc_e)$, we gather constraints, as derived in the preceding sections, and the path condition. For direct-mapped caches, $\Gamma(pc_e)$ can simply be formulated as follows:

$$\boxed{\Gamma(pc_e) \equiv pc_e \wedge \bigwedge_{i\in[1,n]} \left(\Theta^{m,dir}_{i} \wedge \Theta^{h,dir}_{i}\right)} \tag{18}$$

For set-associative LRU caches, we need to additionally account for constraints capturing unique cache conflicts (i.e. Constraint (16)). Hence, $\Gamma(pc_e)$ is formalized via the following constraint:

$$\boxed{\Gamma(pc_e) \equiv pc_e \wedge \bigwedge_{i\in[1,n]} \left(\Theta^{m,lru}_{i} \wedge \Theta^{h,lru}_{i} \wedge \bigwedge_{j\in[1,i)} \Theta^{emp}_{j,i} \wedge \bigwedge_{j\in[1,i)} \Theta^{ehp}_{j,i}\right)} \tag{19}$$

4.3 Modeling Cache Access

In the preceding sections, we discuss our symbolic cache model $\Gamma(pc_e)$ (*cf.* Constraints (18)-(19)). $\Gamma(pc_e)$ encodes the *cache timing* behaviour for all inputs $I \models pc_e$. However, $\Gamma(pc_e)$ does not encode the *cache access* behaviour. This means that $\Gamma(pc_e)$ lacks the capability to compute the set of cache lines accessed for any input $I \models pc_e$. Computing such information is crucial to investigate the information leakage for access-based attacks [30]. The basic idea behind access-based attacks is to first fill up the cache with some data D. Then, based on the knowledge of the cache replacement policy, the attacker determines the cache line for each memory block in D. After the victim process (*e.g.* an encryption routine) finishes execution, the attacker repeats the process of accessing memory blocks in D and computes the cache lines accessed by the victim process. This is possible, as some memory blocks in D were replaced by the victim and the attacker process identifies that such memory blocks took longer time than the rest to access. Recent cache attacks [33] demonstrate the feasibility of access-based cache attacks in ARM-based embedded platforms. Figure 5 outlines the process behind an access-based attack for an LRU cache. In summary, the victim process leaks information if the set of cache lines being accessed is dependent on sensitive inputs (*e.g.* the encryption key in cryptographic routines).

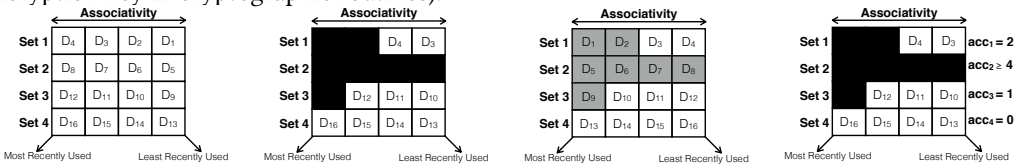


Fig. 5. (a) Attacker fills up the cache with memory blocks $D_1 \ldots D_{16}$ and constructs the state of each cache line with the knowledge of LRU policy, (b) shaded cache lines captures the lines accessed by the victim process, (c) the attacker resumes execution and accesses memory blocks in the order $D_4 \rightarrow D_3 \rightarrow D_2 \rightarrow D_1$ for the first cache set, in the order $D_8 \rightarrow D_7 \rightarrow D_6 \rightarrow D_5$ for the second cache set and so on. The attacker observes that accessing $D_1, D_2, D_5 \ldots D_9$ took longer than cache hits. From this, she can compute the cache lines accessed by the victim process, (d) acc_i captures the number of memory blocks accessed by the victim process from cache set i. We can accurately compute the cache lines accessed by the victim process via $acc_1 \ldots acc_4$.

4.3.1 ***Modeling access-based observers.*** An access-based observation can be modeled by counting the unique memory blocks accessed from each cache set (*cf.* Figure 5(d)). For instance, consider the first cache set in Figure 5(b). Since two memory blocks from the victim process are accessed from the first cache set, an access-based observer will compute that data elements D_1 and D_2 took longer to access than elements D_3 and D_4. However, if the number of memory blocks accessed from the second cache set is *at least* the associativity of the cache, then the access-based observer determines that accessing all elements between $D_5 \ldots D_8$ took longer than a cache hit.

Based on the intuition mentioned in the preceding paragraph, we model an access-based observer via the mapping $O : \Sigma^* \rightarrow (\mathbb{S} \rightarrow [0, \mathcal{A}])$, where Σ is an alphabet defined over cache hits (h) and misses (m), $\mathbb{S}$ is the set of all cache sets and $\mathcal{A}$ is the associativity of a cache. For direct-mapped caches, $\mathcal{A}$=1. Intuitively, for a given execution trace, such an observer computes the number of memory blocks accessed from each cache set. We note that given the number of memory blocks (of a victim process) accessed from each cache set, an access-based observer can accurately compute the set of cache lines accessed by the victim process (*cf.* Figure 5(d)). In the next sections, we formulate $\Gamma(pc_e)$ to encode the number of memory blocks accessed from each cache set. Subsequently, we show the usage of $\Gamma(pc_e)$ to quantify the information leaked via access-based observers.

4.3.2 ***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)). We use the symbolic variable acc_s to capture the

number of cache lines accessed in cache set s. For direct-mapped caches, acc_s can either be one or zero. Concretely, if any of the accessed memory blocks is accessed from cache set s, then acc_s is set to one. Otherwise, acc_s is set to zero. This is formally captured as follows:

$$\Theta_s^{acc^+} \equiv \bigvee_{i\in[1,n]} (set(r_i) = s) \Rightarrow (acc_s = 1); \quad \Theta_s^{acc^-} \equiv \bigwedge_{i\in[1,n]} (set(r_i) \neq s) \Rightarrow (acc_s = 0) \tag{20}$$

Finally, we construct $\Gamma(pc_e)$ by combining the constraints gathered for all cache sets as follows:

$$\boxed{\Gamma(pc_e) \equiv pc_e \wedge \bigwedge_{s\in[1,2^S]} \left(\Theta_s^{acc^+} \wedge \Theta_s^{acc^-}\right)} \tag{21}$$

4.3.3 ***Set-associative caches with LRU policy.*** In LRU caches, each cache set can hold as many memory blocks as the associativity ($\mathcal{A}$) of the cache. Therefore, unlike direct-mapped caches, it is necessary to count unique memory blocks accessed from a given cache set. To this end, we introduce a symbolic variable $acc_{s,i}$ for each cache set s and for each instruction r_i. $acc_{s,i}$ is set to one if and only if the following conditions hold:

(1) Instruction r_i accesses cache set s. Therefore, we have $(set(r_i) = s)$.
(2) Any instruction $r_j \in \{r_1, r_2, \ldots, r_{i-1}\}$ either does not access cache set s or each of r_j and r_i access memory blocks with different tags. This is formally captured as follows:

$$\psi_{acc}(i) \equiv \bigwedge_{j\in[1,i)} \left(set(r_j) \neq s\right) \vee \left(tag(r_j) \neq tag(r_i)\right) \tag{22}$$

Intuitively, if $\psi_{acc}(i)$ is satisfiable, then it ensures that r_i is the first instruction to read from or write to the accessed memory block. Based on this intuition, $acc_{s,i}$ is symbolically bounded as follows:

$$\Theta_{s,i}^{acc^+} \equiv (set(r_i) = s) \wedge \psi_{acc}(i) \Rightarrow \left(acc_{s,i} = 1\right) \tag{23}$$

$$\Theta_{s,i}^{acc^-} \equiv (set(r_i) \neq s) \vee \neg\psi_{acc}(i) \Rightarrow \left(acc_{s,i} = 0\right) \tag{24}$$

Finally, we compute the number of unique memory blocks accessed from cache set s by summing up the value of $acc_{s,i}$ for all memory-related instructions. Let us assume acc_s captures the number of unique memory blocks accessed from cache set s. Based on this notation, our symbolic cache model $\Gamma(pc_e)$ is formalized as follows:

$$\boxed{\Gamma(pc_e) \equiv pc_e \wedge \bigwedge_{s\in[1,2^S]} \left(\bigwedge_{i\in[1,n]} \left(\Theta_{s,i}^{acc^+} \wedge \Theta_{s,i}^{acc^-}\right) \wedge \left(acc_s = \sum_{j\in[1,n]} acc_{s,j}\right)\right)} \tag{25}$$

We note that the values of $acc_{s,i}$ in Constraint (25) is dictated via the constraints $\Theta_{s,i}^{acc^+}$ and $\Theta_{s,i}^{acc^-}$. The variable acc_s captures the number of unique memory blocks accessed from cache set s.

4.4 Size of Constraints

The size of $\Gamma(pc_e)$ in Constraint (18)-(19) is bounded by $O(n^3)$. Here n is the number of memory accesses. The dominating factor in this constraint system is the set of constraints generated from Constraint (13) and Constraint (16). 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 the overall constraint system to be $O(n^3)$. For access-based observers, the size of $\Gamma(pc_e)$ is bounded by $O(n \cdot 2^S)$ for direct-mapped caches (*cf.* Constraint 21) and it is bounded by $O(n^2 \cdot 2^S)$ for set-associative caches (*cf.* Constraint 25). For set-associative caches, additional constraints are required for checking unique memory block accesses (*cf.* Constraint 22).

5 CHECKING INFORMATION LEAKAGE

In this section, we instantiate CHALICE by formulating Φ_O for three different observer models. We assume that t_I is the observed execution trace for input I and we wish to quantify how much information about input I is leaked through t_I.

Observation via total miss count In this scenario, an attacker can observe the number of cache misses of executions [11]. 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 trace $t_I \in \Sigma^*$ associated to an input I, $O(t_I)$ captures the number of cache misses in t_I.

Recall that we use the 0-1 variable $miss_i$ to capture the cache miss behaviour of the i-th memory access. We check the unsatisfiability of the following logical formula to record information leakage:

$$\bigvee_{e \in Paths} \left(\Gamma(pc_e) \wedge \left(\sum_{i \in [1, n_e]} miss_i = O(t_I) \right) \wedge \pi \right) \tag{26}$$

where n_e is the number of memory accesses occurring along path e and π is a predicate defined on program inputs. Concretely, if Constraint (26) is unsatisfiable, we can establish that the information "$\neg\pi \equiv true$" is leaked through t_I. By performing such unsatisfiability checks over the entire program input space, we quantify the information leakage $\mathcal{L}(t_I)$ through execution trace t_I (*cf.* Section 3.3).

Observation via hit/miss sequence For an execution trace $t_I \in \Sigma^*$, an observer can monitor hit/miss sequences from t_I [6]. Concretely, let us assume $\{o_1, o_2, \ldots, o_k\}$ is the set of positions in trace t_I where the observation occurs. If n is the total number of memory accesses in t_I, 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: $O(t_I)_i = 1$ if $t_{o_i} = m$ and $O(t_I)_i = 0$ otherwise. $O(t_I)_i$ captures the i-th bit of $O(t_I)$ and similarly, t_{o_i} captures the o_i-th element in the execution trace t_I. 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 to record information leakage:

$$\bigvee_{e \in Paths} \left(\Gamma(pc_e) \wedge \bigwedge_{i \in \{1, 2, \ldots, k\}} \left(\begin{array}{c} o_i \leq n_e \\ \wedge\ miss_{o_i} = O(t_I)_i \end{array} \right) \wedge \pi \right) \tag{27}$$

where π is a predicate on program inputs. By generating such predicates over the input space, we quantify the information leaked about input I via $\mathcal{L}(t_I)$ (*cf.* Section 3.3). In Constraint (27), our general information leakage checker in Constraint (6) is instantiated with $\Phi_{O,e}$ being set to $\bigwedge_{i \in \{1, 2, \ldots, k\}} \left(miss_{o_i} = O(t_I)_i \right)$.

Observation via cache accesses An access-based observer [30] monitors the cache lines being accessed by a victim process. As shown in Figure 5, this can be computed via the number of memory blocks accessed from each cache set. Therefore, an access-based observer is modeled as follows. $O : \Sigma^* \rightarrow (\mathbb{S} \rightarrow [0, \mathcal{A}])$. For an execution trace $t_I \in \Sigma^*$, $O(t_I)(s)$ (written $O_{t_I}(s)$) captures the number of cache lines touched by the victim process within cache set s. It is worthwhile to note that $0 \leq O_{t_I}(s) \leq \mathcal{A}$.

For access-based observers, the unsatisfiability of the following records information leakage:

$$\bigvee_{e \in Paths} \left(\Gamma(pc_e) \wedge \bigwedge_{s \in [1, 2^S]} \left(\begin{array}{c} \left(O_{t_I}(s) \leq \mathcal{A} - 1\right) \Rightarrow \left(acc_s = O_{t_I}(s)\right) \\ \wedge \left(O_{t_I}(s) = \mathcal{A}\right) \Rightarrow \left(acc_s \geq O_{t_I}(s)\right) \end{array} \right) \wedge \pi \right) \tag{28}$$

where π is a predicate on program inputs. By generating such predicates over the input space, we quantify the information leaked about input I via $\mathcal{L}(t_I)$ (*cf.* Section 3.3). In Constraint (28), we abstract away all scenarios where the number of memory blocks accessed from a cache set is at

least the associativity of the cache (*i.e.* $acc_s \geq \mathcal{A}$). This is to align with the point of view of observer O. Concretely, for any scenario satisfying $acc_s \geq \mathcal{A}$, the observer can *only* determine that all cache lines of set s were touched by the victim process.

6 EVALUATION

Implementation setup: We implemented CHALICE on top of the KLEE symbolic virtual machine [2] based on LLVM. We engineered KLEE to symbolically execute the PISA [8] binary code (compiled with `gcc 2.7.2.3`) – a MIPS like architecture. This is because, cache performance is captured accurately only in the executable binary code. To keep CHALICE modular and extensible for other target binaries, we implemented a translator that converts PISA binary code to the LLVM bitcode. Such a translator is unique in the sense that it focuses on preserving both the memory behaviour and the functionality during the translation, whereas existing disassemblers only preserve the functionality. Some salient features of our translation are as follows: First, we ensure that each load/store instruction in the binary code to have a functionally equivalent load/store instruction in the translated bitcode. Secondly, we preserve the static-single-assignment (SSA) form of LLVM bitcode by systematically inserting *Phi* functions. Thirdly, several instructions at the machine code, e.g., LWL and LWR, may require multiple LLVM instructions to implement. 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.

Experimental setup: To evaluate the effectiveness of CHALICE, we have chosen cryptographic applications from the OpenSSL library [4] (OpenSSL 1.1.0-pre6-dev) and other software repositories [3], as well as applications from the Linux GDK (version 3.0) 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 eight-core 4.00GHz i7-6700K CPU with 8GB of RAM and running Debian 8.4 operating system.

Program	Lines of C code	Lines of MIPS code (disassembled version)	Input size	Max. #Memory access
AES [3]	800	4842	16 bytes	2134
AES [4]	1428	1700	16 bytes	420
DES [4]	552	3480	8 bytes	334
RC4 [4]	160	660	8 bytes	1538
RC5 [4]	256	1740	16 bytes	410
GDK library	2650	2700	4 bytes	126

Table 1. Salient features of the evaluated subjects

Program	Observation via miss count		Observation via hit/miss of an arbitrary access	
	$\mathcal{L}(t_I)$ π_{bit}	$\mathcal{L}(t_I)$ π_{byte}	$\mathcal{L}(t_I)$ π_{bit}	$\mathcal{L}(t_I)$ π_{byte}
AES	0	$\approx 2^{127}$	0	$\approx 2^{127}$
AES	0	$\approx 2^{37}$	0	$\approx 2^{8}$
DES	0	$\approx 2^{62}$	0	$\approx 2^{42}$
RC4	0	$\approx 2^{6}$	0	$\approx 2^{8}$
RC5	0	0	0	0
GDK	0	$\approx 2^{31}$	0	$\approx 2^{31}$

Table 2. $\mathcal{L}(t_I)$ quantified w.r.t. π_{bit} and π_{byte}

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 K equal segments and use k_i to capture the i-th segment. We generate the following predicates on inputs for quantifying information leakage $\mathcal{L}(t_I)$ (*cf.* Section 3.3):

$$\pi_{\textbf{bit}} = \{k_i = v \mid i \in [1, N], v \in [0, 1]\}\,; \quad \pi_{\textbf{byte}} = \left\{k_i = v \mid i \in [1, \frac{N}{8}], v \in [0, 255]\right\}$$

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

6.1 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. For each subject program, we generated a set of executions by selecting a concrete value of the sensitive inputs (e.g. secret key in AES-128) uniformly at random. All other inputs to the subject program (e.g. plaintext in AES-128) were fixed while generating these executions. For such a randomly generated execution, Table 2 demonstrates the quantified information leakage with respect to predicates π_{bit} and π_{byte}. We make the following observations from Table 2. For all scenarios, $\mathcal{L}(t_I)$ is zero when predicates π_{bit} is used. Therefore, we can prove *the absence of any dependency between the cache performance (i.e. the number of cache misses 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 leakage with respect to π_{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 [3]). *This means, there exist at least* 251^{16} *(*$\approx 2^{127}$*) 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 basic AES implementation [3], we observe (*cf.* Table 2) that the OpenSSL version of AES exhibits substantially fewer information leakage. DES exhibits severe information leakage when the adversaries observe cache misses, whereas RC4 exhibits lower leakage as compared to AES and DES, with respect to the respective observation.

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 leakage in RC5, with respect to the observer models studied.

Coverage: Except routines chosen from the GDK library, the subjects in our evaluation are single-path programs. This is standard for cryptographic routines, as they aim to avoid input-dependent branches. Consequently, CHALICE covers 100% of the statements for AES, DES, RC4 and RC5. For routines involving multiple paths, e.g. routines from the GDK library, CHALICE should be used for multiple test inputs. In our evaluation, these inputs were generated randomly and they obtained 80% statement coverage in the routines tested from the GDK library. We note that code coverage is not a suitable metric to evaluate the information leakage. For instance, in AES, we obtained 100% code coverage, yet the quantified information leakage was zero with respect to predicate π_{bit}. This only reflects that the observation is not influenced by values of an individual bit in isolation. However, as observed with respect to predicate π_{byte}, the amount of information leakage can be substantial, as the observation is heavily influenced by values of an individual byte in isolation.

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 single 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, RC4 exhibits such behaviour.

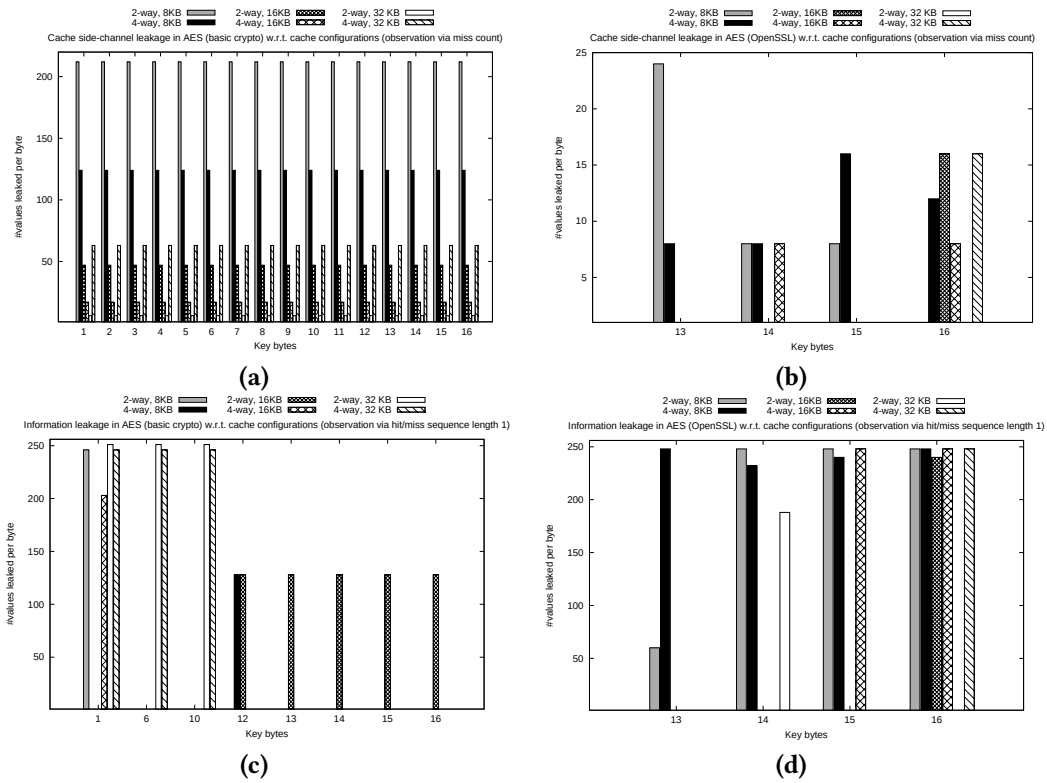


Fig. 6. Information leakage in AES [3, 4] w.r.t. different observers (only bytes leaking information are shown)

6.2 Sensitivity of Leakage w.r.t. Cache

In this section, we evaluate CHALICE for a variety of cache configurations. For each experiment, we show the number of values leaked per input byte. Concretely, for a given observation, CHALICE deduces how many values per byte can be ruled out due to the dependency between input and cache behaviour. Therefore, a higher bar indicates higher leakage. In other words, the output generated by CHALICE can be used to compute the set of potential inputs that led to the given observation.

Experience with AES: In Figures 6(a)-(d), we outline the number of values leaked per byte of the AES secret key. The horizontal axis in these figures capture the individual bytes of the AES secret key. For instance, consider the evaluation shown in Figure 6 for 2-way, 8KB cache. In this scenario, the AES implementation [3] 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 212^{16} possible keys.

Increasing cache size (or associativity) may have two contrasting effects as follows. For a given cache size, consider a subset of the input space $\mathbb{I}_{=C} \subseteq \mathbb{I}_{<C} \cup \mathbb{I}_{=C} \cup \mathbb{I}_{>C}$ (where $\mathbb{I}_{<C} \cup \mathbb{I}_{=C} \cup \mathbb{I}_{>C}$ is the entire input space) which leads to C cache misses. Increasing cache size reduces cache conflict. Therefore, it is possible that some input $i \in \mathbb{I}_{>C}$, 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 $i \in \mathbb{I}_{=C}$ may have less than C cache misses with increased cache size. This reduces the number of inputs having C cache misses, thus increasing the potential leakage through the observation of C cache misses. In Figure 6(a), $\mathcal{L}(t_I)$ reduces for cache sizes up to 16 KB, while it increases for a 4-way, 32KB cache due to the aforementioned effects.

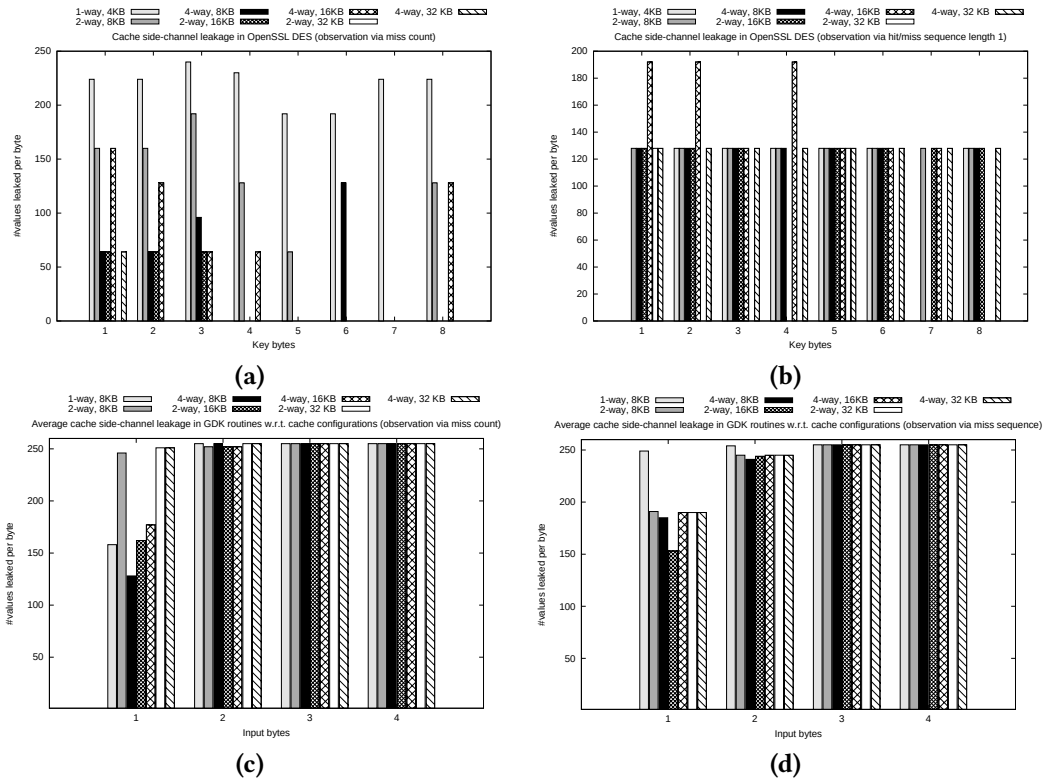


Fig. 7. Information leakage in DES and GDK library with respect to different observers

Experience with DES: Data Encryption standard (DES) [4] is a symmetric key algorithm for electronic data. It encrypts 64 bit plaintext blocks with a 64 bit secret key (56 bits effective key with one bit assigned for each byte as a parity check). Using 8KB caches, for example, DES leaks more than 120 values for several key bytes (*cf.* Figures 7(a)-(b)). Our results summarize the potentially insecure nature of DES, even if we only consider security leaks through cache behaviour.

Experience with RC4: We analyzed the OpenSSL version of RC4 (a stream cipher) implementation with 64 bits keys for several cache configurations (from 8KB to 64KB). `CHALICE` highlights substantial information being leaked about the first byte (in certain cases 254 values out of a total of 256). For bigger cache sizes (*e.g.* > 16KB), such information leakage disappears, as the executions of RC4 only suffer the minimum number of misses to load all the memory blocks into the cache.

Experience with GDK Library: Figures 7(c)-(d) present the average information leakage discovered in routines `gdk_keyval_to_unicode` and `gdk_keyval_name` from the Linux GDK library. We observe several scenarios leading to a complete disclosure of information for the third and the fourth input bytes (*i.e.* 255 out of 256 values are leaked). 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. Since we include path condition pc within our symbolic cache model $\Gamma(pc)$ (*cf.* Constraint (18)), we can accurately quantify the information leakage even in the presence of multiple program paths.

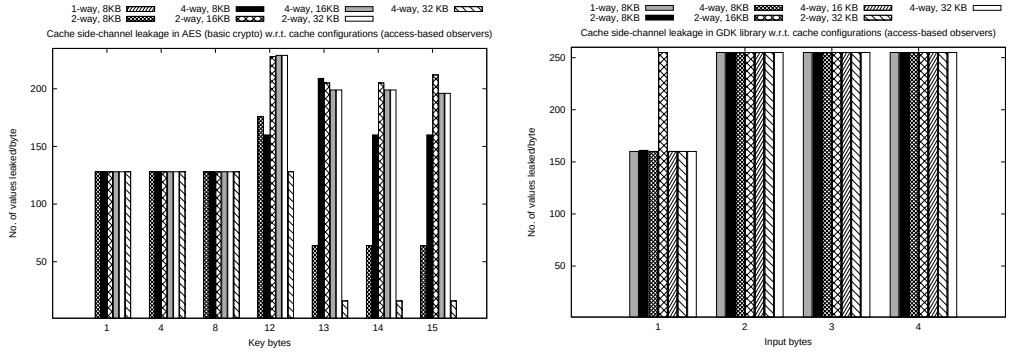
Evaluating constant-time implementation: Constant-time programming is the current standard to counter timing-related leakage in cryptographic software. To evaluate this line of countermeasures via `CHALICE`, we chose eight elliptic curve routines (total 12K lines of MIPS code) from

Subject program	Observation via total miss count					Observation via hit/miss of an arbitrary access				
	Formula size	Peak mem.	T_1	T_{byte}	T_{all}	Formula size	Peak mem.	T_1	T_{byte}	T_{all}
AES [3]	144072	261M	$\approx$ 20 sec	1 hour	16 hours	1580	105M	$<$ 1 sec	$\approx$ 1 min	16 min
AES [4]	21444	129M	$\approx$ 18 sec	77 min	20 hours	265	90M	$<$ 1 sec	$\approx$ 2 min	45 min
DES [4]	53808	127M	$\approx$ 10 sec	50 min	8 hours	1809	35M	$<$ 1 sec	$\approx$ 1 min	12 min
RC4 [4]	38622	1.1G	$\approx$ 4 sec	15 min	4 hours	490	32M	$<$ 1 sec	$\approx$ 1 min	16 min
RC5 [4]	0	28.3M	$\approx$ 15 sec	$\approx$ 15 sec	$\approx$ 15 sec	0	29.2M	$\approx$ 14 sec	$\approx$ 14 sec	$\approx$ 14 sec
GDK	21	102M	$<$ 1 sec	$<$ 1 sec	$\approx$ 2 min	21	100M	$<$ 1 sec	$<$ 1 sec	$\approx$ 1 min

Table 3. T_1 and T_{byte} capture the average time for one solver call and to check information leakage for one input byte, respectively. T_{all} captures the time taken to check information leakage via all predicates in $\mathbf{P_{byte}}$.

Cache	Observation via total miss count					Observation via hit/miss of an arbitrary access				
	Constraint size	Peak mem.	T_1	T_{byte}	T_{all}	Constraint size	Peak mem.	T_1	T_{byte}	T_{all}
2-way, 8 KB	2510964	496M	$\approx$ 2 min	8 hours	127 hours	37477	178M	$<$ 1 sec	$\approx$ 6 min	1 hour
4-way, 8 KB	2507608	570M	$\approx$ 2 min	8 hours	128 hours	27548	999M	$<$ 1 sec	$\approx$ 1 min	19 min
2-way, 16 KB	2518182	655M	$\approx$ 21 sec	1.5 hours	24 hours	28533	618M	$<$ 1 sec	$\approx$ 3 min	53 min
4-way, 16 KB	2511030	487M	$\approx$ 33 sec	2.3 hours	37 hours	74060	475M	$<$ 1 sec	$\approx$ 12 min	3.2 hour
2-way, 32 KB	2518052	556M	$<$ 1 sec	3 min	47 min	76304	485M	$<$ 1 sec	$\approx$ 2 min	30 min
4-way, 32 KB	2518118	820M	$\approx$ 45 sec	3.2 hours	50 hours	78689	349M	$<$ 1 sec	$\approx$ 3 min	41 min

Table 4. Analysis time w.r.t. cache configurations. T_1, T_{byte} and T_{all} have the same interpretation as Table 3.



(a) **(b)**

Fig. 8. Information leakage w.r.t. access-based observers (only bytes leaking information are shown)

`FourQLib` [1] – namely `eccmadd`, `eccnorm`, `pt_setup`, `eccdouble`, `R1_to_R2`, `R1_to_R3`, `R2_to_R4` and `R5_to_R1`. For all routines, `CHALICE` reported zero leakage within five minutes.

Analysis time: Tables 3-4 outline the analysis time for timing-based observers. In most cases, a single call to the solver, which reports information leakage via unsatisfiability checks (*e.g.* via checking Constraint (6)), is efficient. Due to the repeated calls to the solver, checking the information leakage, for the entire input space, takes significant time. For a given program, this time (i.e. T_{all}) is approximately the product of T_{byte} and the number of bytes in the input space. This is because, our analysis time mostly remains the same for any arbitrary input byte of a given program. However, since `CHALICE` incorporates an *anytime* strategy, the bounds on the quantification of $\mathcal{L}(t_I)$ are valid for any explored subsets of the program paths and input space. Finally, the analysis time can be improved if we assign independent threads to check information leaked about each input byte.

6.3 Evaluation with Access-based Observers

Figures 8(a)-(b) outline our evaluation for AES [3] and GDK library with access-based observers. For 8KB direct-mapped caches, we observed that all cache lines are filled up irrespective of the value of AES key. Hence, `CHALICE` did not discover any information leakage in AES for direct-mapped 8KB caches (*cf.* Figure 8(a)). However, for set-associative caches, the occupied cache lines are dependent

on the AES key. This leads to significant information being leaked from certain bytes, e.g., more than 200 values (out of 256) on average were leaked from bytes 12 . . . 15, with a 4-way, 16KB cache. For GDK-library routines, we observed that each program path often has a unique cache-access behaviour (in terms of which cache lines are accessed). As a result, provided only the accessed cache lines, an observer can accurately compute the executed program path. In our evaluation, CHALICE highlights substantial information leakage from all input bytes of GDK-library routines (*cf.* Figure 8(b)). This set of evaluations highlight that the core capabilities implemented within CHALICE are effective to quantify side-channel leakage for both timing-based and access-based observer models. For RC4 and RC5, CHALICE does not report any leakage. For the Openssl version of AES and DES, CHALICE reports a leakage (i.e. $\mathcal{L}(t_I)$) of 2^{11} and 2^{39}, respectively, on average.

Analysis time: Table 5 outlines the time taken by CHALICE for access-based observers. We use AES [3] for this set of experiments, as it takes the maximum time. Since $\Gamma(pc_e)$ depends on the cache size for access-based observers, we note that the constraint size changes (*cf.* Table 5) with respect to cache size. Nevertheless, the time taken for each solver call remains short. As a result, due to the incremental nature of our computation, CHALICE effectively quantifies the information leakage via access-based observation within reasonable time.

Cache	Observation via cache line accesses				
	Constraint size	Peak mem.	T_1	T_{byte}	T_{all}
1-way, 8 KB	9736	60M	≈ 2 sec	8 min	2.2 hours
2-way, 8 KB	423213	67M	≈ 6 sec	24 min	6.1 hours
4-way, 8 KB	220973	110M	≈ 7 sec	30 min	8.3 hours
2-way, 16 KB	827693	59M	≈ 11 sec	47 min	12 hours
4-way, 16 KB	423213	59M	≈ 7 sec	28 min	7.5 hours
2-way, 32 KB	1636516	67M	≈ 6 sec	25 min	6.7 hours
4-way, 32 KB	827556	59M	≈ 6 sec	27 min	7.1 hours

Table 5. Analysis time for access-based observers. T_1, T_{byte} and T_{all} have the same interpretation as Table 3.

6.4 Analysis Sensitivity w.r.t. Observation

Since we quantify leakage from execution trace t_I, the leakage $\mathcal{L}(t_I)$ depends on the observed cache behaviour, e.g., observed miss count. Figure 9 captures the information leakage with respect to observed miss count. Although we observe that $\mathcal{L}(t_I)$ mostly decreases with increased miss count, there is no direct correlation between the observed miss count and the computed leakage. It is worthwhile to note that results reported in Figure 9 are consistent with the observation at the tail of the essentially gaussian distribution captured in Figure 1.

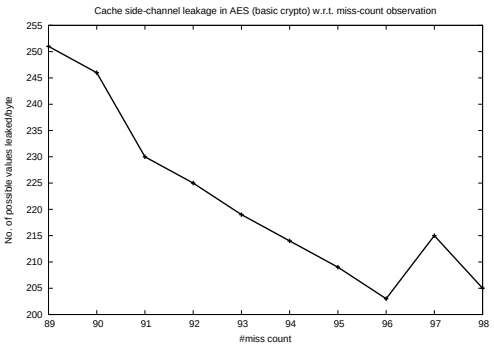


Fig. 9. $\mathcal{L}(t_I)$ in AES [3] w.r.t. observed miss count (1-way, 8KB cache)

7 RELATED WORK

In the following, we will position our work in the research area by reviewing the related literature.

Static analysis of caches. Static cache analysis [19, 40] has been an active research topic for the last two decades. Compared to static cache analysis [19, 40], CHALICE has significant flavors of testing and debugging. Concretely, CHALICE can highlight poor choices of secret keys in implementing encryption standards. CHALICE also highlights memory accesses that leak substantial information. This can be leveraged to drive security-related optimizations.

Analysis of side channel. The closest to our work are approaches based on static analysis [23, 31, 32]. However, these analyses fail to detect critical scenarios when a particular observation leaks

substantially more information than the rest [12]. CHALICE quantifies information leakage 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 [7, 9]. However, due to the dynamic nature of the analysis embodied within CHALICE, it only provides bounds on the information leakage for tested inputs. Existing work based on symbolic execution [36] quantifies side-channel leakage via counting the number of observations [23, 32], and it ignores the effect of micro-architectural entities, such as caches. CHALICE formulates cache side-channel leakage via logical constraints, in contrast to the probabilistic modeling of cache and prefetching [37]. Consequently, CHALICE provides deterministic bounds on the information being leaked through the cache. Besides, contrary to the existing work [37], CHALICE targets arbitrary programs beyond block ciphers. Existing work on side-channel vulnerability metric [22] quantifies *how well* an attacker can retrieve information from a system, but, *does not highlight the information leaked to the attacker*. CHALICE is complementary to the proposed metric [22] and CHALICE can be combined with such metrics to build more advanced metrics for measuring side channel leakage.

Information leakage quantification. We note that existing techniques [15, 35] aim to quantify information leakage via satisfiability checking and counting. These works are not targeted towards micro-architectural side channels and they do not provide an *anytime* strategy to quantify the information leakage. In contrast, CHALICE provides an *anytime* algorithm based on input space partitioning and the refinement of these partitions. The refinement strategy guarantees to improve the accuracy of computed cache side-channel leakage.

Software testing. In contrast to testing side-channel leakage [10, 17], our goal is to quantify the leakage of information for a given cache behaviour. The overarching goal of CHALICE is to combine its capability with the approaches based on software testing [10, 17]. To this end, CHALICE quantifies the cache side-channel leakage for test cases generated via such approaches. In contrast to a preliminary version of CHALICE [18], which only considers timing-based observers, our current work accounts for both timing-based [6, 11] and access-based observers [30]. Moreover, we perform a detailed analysis to show the sensitivity of cache side-channel leakage and analysis time with respect to different cache configurations and for three different observer models.

Cache side-channel attacks. Over the last few decades, cache-based side-channel attacks have emerged to be a prevalent class of security breaches for many systems [26]. The observer models used in this paper are based on existing cache attacks [6, 11, 30]. However, in contrast to these approaches, CHALICE does not aim to engineer new cache attacks. Based on a configurable observer model, CHALICE aims to quantify the information leakage for a *given* cache attack. We believe that CHALICE is generic to incorporate advanced attack scenarios [14, 28, 29, 34, 43, 44] that are currently not handled in this paper. In general, as long as cache attacks are expressed via the intuition given in Section 5, we can instantiate CHALICE to quantify the information leakage via such attacks.

Countermeasures against side-channel attacks. CHALICE is orthogonal to approaches proposing countermeasures [21, 39, 42] via hardware [42], compiler [39] or runtime environment [21]. Of course, CHALICE can validate the proposed countermeasures 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 can highlight critical information leakage 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 involving a strong attacker model is justified. However, we note that CHALICE should be used together with a test generation tool [17] that obtains sufficient coverage of observations made by the attacker. This is because, CHALICE quantifies leakage from a given observation. In its current state, CHALICE does not provide capabilities to fix cache side channels. Hence, the output generated by CHALICE needs to be investigated manually for reducing information leakage. Potential debugging strategies will be to restructure the code, selectively bypassing the cache or using software-controlled memory for certain memory accesses. Finally, the effectiveness and efficiency of CHALICE depends on the granularity of checking (i.e. π_{bit} or π_{byte} in Table 2). We observed that for realistic programs, the choice π_{byte} provides valuable insight on the information leakage. Although we can incrementally refine the granularity of checks based on the obtained results, it is currently not implemented and can be considered in the future release of CHALICE.

Attack models. CHALICE currently does not handle active adversaries that aim to retrieve information by manipulating cache states on-the-fly [44] or target shared last-level caches [34]. Moreover, as CHALICE does not incorporate the capabilities for out-of-order and speculative execution, currently, CHALICE does not quantify the information leakage for recently discovered Meltdown [24] and Spectre [25]. Nevertheless, our powerful symbolic reasoning framework allows us to consider additional micro-architectural features (e.g. shared caches, out-of-order execution and speculation) while formulating $\Gamma(pc_e)$ (cf. subsection 4.1). This can be considered in a future extension of CHALICE. We note that the central idea behind our information leakage quantification, as described in subsection 3.2-subsection 3.3, can be leveraged without any change for such extension.

Perspective. In this paper, we have shown that the mechanism of CHALICE is essential for quantifying the amount of information that can leak through memory performance and cache-access statistics. 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 leakage scenarios in OpenSSL and Linux GDK libraries, among others. In future work, we will explore the synergy between our input partitioning scheme and model counting to reduce the number of calls to solvers and speed up the quantification process. We hope that the core idea of CHALICE impacts regular activities in software engineering including regression testing.

REFERENCES

[1] [n. d.]. FourQLib Library. ([n. d.]). https://github.com/Microsoft/FourQlib/ (Date last accessed 20-October-2017).
[2] 2008. KLEE LLVM Execution Engine. (2008). https://klee.github.io/.
[3] 2015. AES Implementation. (2015). https://github.com/B-Con/crypto-algorithms.
[4] 2016. OpenSSL Library. (2016). https://github.com/openssl/openssl/tree/master/crypto.
[5] 2016. UC Davis, Mathematics. Latte integrale. (2016). https://www.math.ucdavis.edu/~latte/.
[6] Onur Acıiçmez and Çetin Kaya Koç. 2006. Trace-driven cache attacks on AES. In *Information and Communications Security*. 112–121.
[7] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying Constant-Time Implementations. In *USENIX*. 53–70.
[8] Todd Austin, Eric Larson, and Dan Ernst. 2002. SimpleScalar: An infrastructure for computer system modeling. *Computer* 35, 2 (2002).
[9] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Carlos Daniel Luna, and David Pichardie. 2014. System-level Non-interference for Constant-time Cryptography. In *CCS*. 1267–1279.
[10] Tiyash Basu and Sudipta Chattopadhyay. 2017. Testing Cache Side-Channel Leakage. In *ICST Workshops*. 51–60.
[11] Daniel J Bernstein. 2005. Cache-timing attacks on AES. (2005).
[12] Nataliia Bielova. 2016. Dynamic Leakage: A Need for a New Quantitative Information Flow Measure. In *PLAS*. 83–88.
[13] Mateus Borges, Quoc-Sang Phan, Antonio Filieri, and Corina S. Pasareanu. 2017. Model-Counting Approaches for Nonlinear Numerical Constraints. In *NFM*. 131–138.

[14] Billy Bob Brumley and Risto M Hakala. 2009. Cache-timing template attacks. In *ASIACRYPT*. 667–684.

[15] Miguel Castro, Manuel Costa, and Jean-Philippe Martin. 2008. Better bug reporting with better privacy. In *ASPLOS*. 319–328.

[16] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. 2016. Algorithmic Improvements in Approximate Counting for Probabilistic Inference: From Linear to Logarithmic SAT Calls. In *IJCAI*. 3569–3576.

[17] Sudipta Chattopadhyay. 2017. Directed Automated Memory Performance Testing. In *TACAS*. 38–55.

[18] Sudipta Chattopadhyay, Moritz Beck, Ahmed Rezine, and Andreas Zeller. 2017. Quantifying the information leak in cache attacks via symbolic execution. In *MEMOCODE*. 25–35.

[19] Sudipta Chattopadhyay and Abhik Roychoudhury. 2013. Scalable and precise refinement of cache timing analysis via path-sensitive verification. *Real-Time Systems* 49, 4 (2013), 517–562.

[20] Thomas M. Cover and Joy A. Thomas. 2006. *Elements of Information Theory (Wiley Series in Telecommunications and Signal Processing)*.

[21] Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. 2015. Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity.. In *NDSS*.

[22] John Demme, Robert Martin, Adam Waksman, and Simha Sethumadhavan. 2012. Side-channel vulnerability factor: A metric for measuring information leakage. In *ISCA*. 106–117.

[23] Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. 2015. CacheAudit: a tool for the static analysis of cache side channels. *TISSEC* 18, 1 (2015), 4.

[24] Moritz Lipp et al. 2018. Meltdown. *ArXiv e-prints* (2018). arXiv:1801.01207

[25] Paul Kocher et al. 2018. Spectre Attacks: Exploiting Speculative Execution. *ArXiv e-prints* (2018). arXiv:1801.01203

[26] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. 2016. A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware. In *Cryptology ePrint Archive*. https://eprint.iacr.org/2016/613.pdf/.

[27] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *PLDI*. 213–223.

[28] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security*. 897–912.

[29] Roberto Guanciale, Hamed Nemati, Christoph Baumann, and Mads Dam. 2016. Cache Storage Channels: Alias-Driven Attacks and Verified Countermeasures. In *IEEE Symposium on Security and Privacy*. 38–55.

[30] David Gullasch, Endre Bangerter, and Stephan Krenn. 2011. Cache games–bringing access-based cache attacks on AES to practice. In *IEEE Symposium on Security and Privacy*. 490–505.

[31] Boris Köpf and David A. Basin. 2007. An information-theoretic model for adaptive side-channel attacks. In *CCS*. 286–296.

[32] Boris Köpf, Laurent Mauborgne, and Martín Ochoa. 2012. Automatic quantification of cache side-channels. In *CAV*. 564–580.

[33] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security Symposium*. 549–564.

[34] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *IEEE Symposium on Security and Privacy*. 605–622.

[35] Ziyuan Meng and Geoffrey Smith. 2011. Calculating bounds on information leakage using two-bit patterns. In *PLAS*. 1.

[36] Corina S. Pasareanu, Quoc-Sang Phan, and Pasquale Malacaria. 2016. Multi-run side-channel analysis using Symbolic Execution and Max-SMT. In *CSF*. 387–400.

[37] Chester Rebeiro and Debdeep Mukhopadhyay. 2015. A Formal Analysis of Prefetching in Profiled Cache-Timing Attacks on Block Ciphers. *IACR Cryptology ePrint Archive* 2015 (2015), 1191.

[38] Geoffrey Smith. 2009. *On the Foundations of Quantitative Information Flow*. Springer Berlin Heidelberg, Berlin, Heidelberg, 288–302. https://doi.org/10.1007/978-3-642-00596-1_21

[39] Deian Stefan, Pablo Buiras, Edward Z Yang, Amit Levy, David Terei, Alejandro Russo, and David Mazières. 2013. Eliminating cache-based timing attacks with instruction-based scheduling. In *ESORICS*. 718–735.

[40] Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. 2000. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Systems* 18, 2-3 (2000).

[41] Eran Tromer, Dag Arne Osvik, and Adi Shamir. 2010. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology* 23, 1 (2010), 37–71.

[42] Zhenghong Wang and Ruby B. Lee. 2007. New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. In *ISCA*. 494–505.

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

[44] Yuval Yarom, Daniel Genkin, and Nadia Heninger. 2017. CacheBleed: a timing attack on OpenSSL constant-time RSA. *J. Cryptographic Engineering* 7, 2 (2017), 99–112.