CaSym: Cache Aware Symbolic Execution for Side Channel Detection and Mitigation

Robert Brotzman, Shen Liu, Danfeng Zhang, Gang Tan, Mahmut Kandemir
Department of Computer Science and Engineering
Pennsylvania State University
State College, PA
{*rcb44, sxl463, zhang, gtan, kandemir*}*@cse.psu.edu*

***Abstract*—Cache-based side channels are becoming an important attack vector through which secret information can be leaked to malicious parties. Previous work on cache-based side channel detection, however, suffers from the code coverage problem or does not provide diagnostic information that is crucial for applying mitigation techniques to vulnerable software. We propose CaSym, a cache-aware symbolic execution to identify and report precise information about where side channels occur in an input program. Compared with existing work, CaSym provides several unique features: (1) CaSym enables verification against various attack models and cache models, (2) unlike many symbolic-execution systems for bug finding, CaSym verifies all program execution paths in a sound way, (3) CaSym uses two novel abstract cache models that provide good balance between analysis scalability and precision, and (4) CaSym provides sufficient information on where and how to mitigate the identified side channels through techniques including preloading and pinning. Evaluation on a set of crypto and database benchmarks shows that CaSym is effective at identifying and mitigating side channels, with reasonable efficiency.**

***Keywords*-side-channels; symbolic execution; cache**

I. Introduction

Side-channel attacks exploit information gathered from the physical implementation of computer systems to steal sensitive data. Among all side channels, side channels resulting from shared data/instruction cache have shown to be extremely effective. The first cache-based attacks on crypto systems learn AES [1], [2] and RSA keys [3] by analyzing the timing variance due to cache usage of the victim, assuming the attacker shares the L1 cache with the victim. More realistic attacks have recently emerged showing the practicality of various timing attacks using the shared CPU cache, including the last-level cache. For example, in a multi-tenant cloud system, cache-based timing attacks are shown to be a potential risk across VMs [4], [5], [6] and across isolated applications in secure enclaves [7]; more evidence is emerging showing practical timing attacks that break crypto systems [8], [9], [10].

To counter cache-based side channels, developers of popular cipher implementations (e.g., RSA and AES in OpenSSL and Libgcrypt) have been manually identifying side channels and patching their software with counter measures.[1] However, the manual process often misses important side channels, as evidenced by recent discoveries of new side channels in crypto implementations [11]. Furthermore, the highly-specialized patches by crypto developers are of little help for non-experts to fix side channels in other crypto implementations and other security-sensitive applications.

Previous work has made good progress in automatic detection of cache-based side channels. Example systems include CacheAudit [12], [13] and CacheD [11]. While those systems have successfully identified side channels in real-world programs, they still face a couple of limitations.

First, although some of the work based on symbolic execution (e.g., [11]) naturally offers counterexamples that represent program executions that exhibit the identified side channels, they only explore one or multiple dynamic execution paths and suffer from the problem of code coverage. That is, they are unable to detect side channels in unexecuted code, and side channels due to branches conditioned on confidential data. Some systems [12], [13] use abstract interpretation, which covers all program paths. But they do not show developers where the side channels are and why they are side channels. More importantly, such missing information could be used to construct mitigations that fix the identified side channels.

Second, an adequate cache model is important for language-based analysis. To detect cache-based side channels, one approach is to directly check the existence or absence of data in the cache via architecture-dependent cache models (e.g., [12]). But the unpleasant consequence is that the security guarantee offered by those systems becomes architecture-dependent, and reasoning on concrete cache models is typically costly. Other systems (e.g., [13], [11]) check that the entire trace of accessed memory addresses is secret-independent, without using a concrete cache model. However, doing so can be too conservative. For example, AES implementations with preloading still exhibit key-dependent memory accesses, but they are secure since all

[1] See Section 5 of [2] for a discussion of common counter measures used in crypto implementations.

key-dependent memory accesses will result in a cache hit with preloading.

In this paper, we propose a novel framework, called CaSym, for detecting and mitigating cache-based side channels. CaSym tackles the aforementioned limitations via the following components.

- *Cache-aware symbolic execution (Section V).* CaSym takes a program in a compiler IR (specifically, LLVM IR) as input and performs symbolic execution to track both program and cache states. The symbolic state produced by CaSym's symbolic execution is used to construct a formula fed to an SMT solver. A satisfying solution to the formula produces public values as well as two sensitive values that demonstrate the existence of a cache-based side channel: those values trigger two program executions that result in different cache states. Unlike previous work that uses symbolic execution for detecting cache-based side channels [11], CaSym overestimates all program paths and therefore does not suffer from the code-coverage problem. CaSym soundly translates loops into loop-free code before performing symbolic execution (at the cost of some precision loss).
- *Cache models (Section VI).* For generality, CaSym takes any cache model that defines two operations (initialize and update) and an equality test. Besides the concrete cache models (such as LRU) used in previous work, CaSym also employs two novel cache models: the infinite cache model and the age model. Unlike other existing models, they hide implementation details and provide good balance between precision and generality. The infinite cache model represents an optimistic view of cache: if there is a side channel under the this model, then the side channel likely will exist in other more realistic cache models. The age model represents a pessimistic view of the cache: if there is a side channel under this model, then the side channel likely will exist in some realistic cache models. Compared with concrete cache models (e.g., LRU), the infinite cache model and the age model offer significantly better scalability and comparable precision.
- *Localizing and fixing side channels (Section VII).* From the counterexample (i.e., values that trigger two program executions that result in different cache states), CaSym utilizes the solver solution to localize the causes of the identified side channels and reports the problematic program points. From those problematic program points, mitigation mechanisms, such as preloading and pinning, can then be applied to eliminate those side channels in a straightforward manner.

We have applied CaSym on a set of crypto benchmarks using both symmetric and asymmetric ciphers as well as database benchmarks (Section IX). The experiments confirmed that CaSym can identify known side channels in these benchmarks, report precise information about where the side channels are and why they are side channels, as well as verify that some benchmarks are side-channel free (based on realistic attack and cache models). We also present new vulnerabilities, which to the best of our knowledge have not been found before in the glibc library code and the PostgreSQL database.

```
1  void Example() {
2    ... // computing RK[0..3] from key
3    RK[4]=RK[0]^Sbox[(RK[3] >> 8) & 0xFF];
4    res = res * res;
5    res = res % mod;
6    if(bit_set_at_i(key[0],i)) {
7        res = base * res;
8        res = res % mod;
9    }
10 }
```

Figure 1: Example program composed of snippets of real code which demonstrate the two kinds of side channels.

II. Background Information

A. Cache-based side channels

Side channels are information channels that were not intended to convey information. These come in many forms, such as timing, power consumption, network traffic, etc. In this paper, we consider cache-based side channels. Fig. 1 illustrates two kinds of cache-based side channels: secret-dependent array accesses and secret-dependent branches[2].

A secret-dependent array access occurs when the value of the index into an array is dependent on secrets. In this case, given two different secret values, different memory addresses will be loaded, causing different data to be stored into or evicted from cache. For example, line 3 in Fig. 1 is a secret-dependent array access, since the index into Sbox is part of the round key. The round key in AES can reveal the decryption key. Such vulnerabilities have led to real cache-based side channel attacks on AES (e.g., [14], [2], [15]).

A secret-dependent branch occurs when the outcome of a branch condition depends on secrets. In this case, given two different secret values, different branches can be taken, causing different data to be stored into or evicted from cache if branches have different memory access patterns. For example, the branch condition at line 6 in Fig. 1 is secret dependent since the outcome depends on the i-th bit of `key[0]`. Only when the i-th bit is set, variables `res`, `base` and `mod` are accessed. Such vulnerabilities have led to real cache-based side channel attacks on RSA and ElGamal (e.g., [9], [10]).

[2]Although the code is highly simplified, the code is composed of real code from cipher implementations: lines 2 to 3 resembles a snippet of the AES implementation in mbed TLS, and lines 6 to 10 resembles a snippet of RSA implementation in Libgcrypt.

In summary, this paper targets cache-based side channels: information leakage due to cache variants depending on confidential data. Informally, a program is free of cache-based side channels, if secrets in a program do not influence the cache state. We note that while much work has been done on timing channels in general (e.g., recent work of [16], [17], [18], [19]), most of them model execution time as a function of the number of instructions being executed. Since such a simplified timing model ignores the effect of cache on timing, they do not detect cache-based side channels.

B. Threat model

We consider an attacker who shares the same hardware platform with the victim, a common scenario in the era of cloud computing. The attacker has no direct access to the memory bus. Hence, he cannot directly observe the memory access traces from CPU. However, we assume the attacker can *probe* the shared data cache state, in order to detect whether some victim's data is in cache or not. This model captures most cache-based side channel attacks in the literature, such as an attacker who observes cache accesses by measuring the latency of the victim program (e.g., cache reuse attacks [20], [21], [22], [14] and evict-and-time attack [2]), or the latency of the attacker's program (e.g., prime-and-probe attacks [2], [15], [21], [8], [10]).

Based on at which point of the victim's program the attacker may probe the shared cache, there are two kinds of attackers (here, we follow the terminology used in [12][3]):

- access-based: when an attacker can probe the cache (i.e., determine if data are in cache or not) only upon the termination of the victim program.
- trace-based: when an attacker can probe the cache after each program point in the victim program.

III. Related Work

We next discuss related work on identifying and mitigating cache-based side channels.

Detecting and mitigating cache-based side channels: CacheAudit [12] and its extension [13] statically provide an upper bound of cache-based information leakage in a program. To do so, they abstract concrete cache states and, on top of that, estimate all possible final cache states using abstract interpretation. While a sound estimation of leakage bound is very useful for estimating the severeness of side channels in a program, such a leakage bound provides little insights on how to fix the side channels.

Recent work of CacheD [11] takes a concrete program execution trace as input and symbolizes secret values during a symbolic execution to identify secret-dependent memory accesses. Since all values except secrets are concrete in the analysis, CacheD is likely to be more precise than a static program analysis (i.e., have fewer false positives). However, CacheD explores only the same execution path as the input dynamic trace. Hence, vulnerabilities in the unexplored code or those that are due to secret-dependent branches cannot be detected by CacheD.

Moreover, we note that those analyses work on binary-level code, while CaSym works on IR-level code. Low-level details, such as memory allocation, could make a binary-level analysis more accurate. For example, the work by Doychev and Köpf [13] shows that some optimization level of gcc removes a side channel that shows up with other levels. However, using low-level details also makes the security guarantee compiler-dependent, which is undesirable for crucial software such as cipher implementations.

Some work uses program transformation to equalize the memory access pattern of secret-dependent branches, either by padding those branches to follow similar memory access patterns [23], [24] or rewriting those branches to be sequential programs [25], [26]. However, such mechanisms do not handle secret-dependent array accesses, which are the root cause of attacks on AES (e.g., [1], [2]).

Zhang et al. [27] propose a timing contract that enables a software-hardware co-design for mitigating cache-based side channels. Based on the contract, they show that full-system security can be enforced by a sound type system as well as contract-aware hardware. Their system requires specialized hardware, while our analysis identifies potential side channels on *commodity hardware*.

Detecting other kinds of timing channels: Prior work also uses static analysis to detect information leakage via program execution time. However, they are largely orthogonal to this paper: their timing model ignores the effects of cache, the root cause of cache-based side channel attacks.

For example, recent work [16], [17] applies symbolic execution to synthesize concrete public inputs that lead to maximum leakage via timing channels. However, their analyses do not model the effects of cache; execution time is modeled as the number of instructions being executed.

Chen et al. [19] prove ϵ-bounded noninterference for a program with regard to observable attributes such as execution time, memory usage, and response size. Blazer [18] has a similar goal, but it uses a novel decomposition technique to break a program up into partitions. However, these works only handle side channels whose effects can be modeled as a constant "cost" for each instruction. But an accurate modeling of cache requires more expressive power in program analysis.

OS- and architecture-level techniques: To mitigate cache-based side channels, one direction is to either physically or logically partition the data cache. At the OS level, Raj et al. [28] statically partition the last level cache into regions and allow VMs to use different regions by partitioning physical memory pages accordingly. Shi et al. [29]

[3]The work [12] also considers a time-based attacker, who may only observe the overall execution time of victim program. We do not consider this model since it is weaker than cache-probing attacks we consider in this paper.

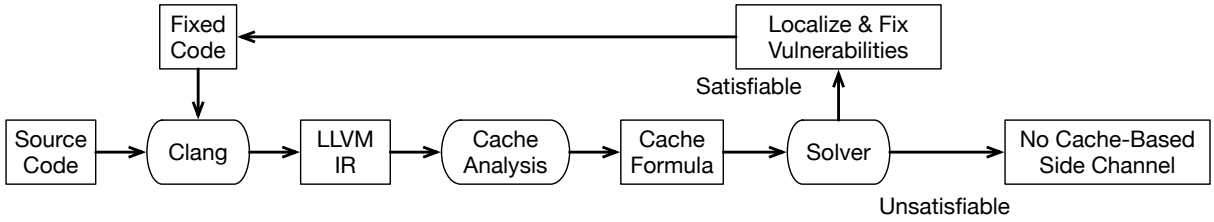

Figure 2: System flow of CaSym.

show that dynamic page coloring helps to establish strong isolation between different applications in terms of cache usage. StealthMem [30], [31] manages a set of locked cache lines per core, which are never evicted from the cache, and efficiently multiplexes them so that each VM can load its own sensitive data into the locked cache lines. Cache partitioning has also been explored at the hardware level [20], [32], [33], [34]. Some other previous work explores injecting noises to the timing signal, such as Düppel [35] at the system level, as well as RPCache [32], Newcache [36], and random fill cache [37] at the hardware level.

IV. System Overview

Fig. 2 depicts the system flow of CaSym. It takes the source code of the input program and uses LLVM's front end, Clang, to convert the program into LLVM IR code. CaSym then performs cache analysis on the IR code to build a cache formula (verification condition) that represents how the program manipulates the cache state. The formula is fed to an SMT solver. A satisfiable answer of the formula means a side channel. In this case, CaSym uses the solution from the solver to localize the error and report it; this information can then be used for mitigation. An unsatisfiable cache formula means no cache-based side channel exists in the input program.

CaSym's cache analysis is based on symbolic execution. At a high level, the symbolic execution takes an input program and outputs a symbolic state that models how the program relates the program's initial program and cache states to its final program and cache states.

To discuss the process more formally, we introduce some notation used throughout this paper. Assume the input program has n program variables, X_1 to X_n, as well as some arrays with statically known sizes (e.g., A[16] and B[1024]). We use small-case letters x_1 to x_n for logical variables that represent the *symbolic* initial values of X_1 to X_n.[4] Therefore, at the beginning of the program, we have $X_1 = x_1 \wedge \ldots \wedge X_n = x_n$. At the end of the program, the final values in X_1 to X_n may have changed and are represented as symbolic expressions that may contain occurrences of x_1 to x_n. As an abbreviation, we write $\overline{x}$ for $x_1, \ldots, x_n$ and $\overline{X}$ for $X_1, \ldots, X_n$.

We use c to represent symbolically the initial cache state and C for an implicit variable that tracks the current symbolic cache state; initially, $C = c$ and at the end of the program the cache state C is a symbolic expression that may contain occurrences of c and x_1 to x_n.

According to the input program's semantics, CaSym's symbolic execution then builds a formula that represents the final program and cache states, using the initial program and cache states $(\overline{x}, c)$:

$$\sigma(\overline{x}, c, \overline{X}, C)$$

Assume at the beginning of the program, K_0 is a set of secret variables whose values should be protected from side-channel attacks.[5] A variable not in K_0 is considered a public input whose value does not need protection. Informally, a program is free of side channels if K_0 has no influence on the final cache state C. Following the definition of noninterference [38], we formalize the verification condition for cache-based side channels as follows.

Given formula σ and K_0, CaSym issues the following verification condition to an SMT solver:

$$\begin{aligned} \mathrm{VC}(\sigma, K_0) \triangleq\ & \exists \overline{x}, \overline{x'}, \overline{X}, \overline{X'}, c, c', C, C', \\ & \neg(\forall X \in K_0, x = x') \wedge (\forall X \notin K_0, x = x') \\ & \wedge c = c' \wedge \sigma(\overline{x}, c, \overline{X}, C) \wedge \sigma(\overline{x'}, c', \overline{X'}, C') \\ & \wedge \neg \mathrm{eq_C}(C, C') \end{aligned} \tag{1}$$

For the special case when $K_0 = \{X_1\}$, the formula requires $x_1 \neq x_1' \wedge x_2 = x_2' \wedge \ldots \wedge x_n = x_n'$; i.e., two different values for X_1 and the same value for other variables. When equation (1) is satisfiable, then it is possible to run the program twice with two different secrets, the same public inputs, and the same initial cache state, and get two different final cache states C and C'. That is, by observing the final cache state, an attacker can learn information about the secret.

Compared with previous work based on abstract interpretation [12], [13], a benefit of checking the verification condition above is that a solution of equation (1) leads to two program executions that exhibit the existence of side

[4]For scalability, CaSym does not track symbolic values in arrays; more discussion on this later.

[5]Note that our implementation also allows marking a fixed-size array as the secret (e.g., the `key` array in the example of Fig. 1).

channels: in this case, the SMT solver produces two different secret values, which serve as the witness of the identified side channel(s). CaSym then uses an error localization algorithm to diagnose the witness (i.e., two program executions) to pinpoint the problematic program points that cause the side channel(s) in the source code.

On the other hand, when the formula is unsatisfiable, we are assured that there is no cache-based side channel in the given environment: regardless of which secret values are chosen, the cache state does not change. We note that this property is essentially a cache-aware variant of noninterference [38], which states that confidential data cannot influence public outputs (i.e., cache states).

V. Tracking Cache States

In this section, we discuss how CaSym performs symbolic execution to go from the input program to the formula $\sigma(\overline{x}, c, \overline{X}, C)$ that relates the initial program and cache states to the final program and cache states. As in all symbolic-execution systems, CaSym's symbolic execution is based on program paths. For each path, CaSym computes a symbolic state (including cache state) at the end of the path. For a loop-free program, CaSym considers every path in the program and combines the symbolic states of all paths into a single formula. Moreover, for programs with loops, CaSym uses novel language statements and a transformation that soundly converts the loops into loop-free counterparts before performing symbolic execution.

Note that since CaSym considers all paths, its symbolic execution is similar to strongest postcondition calculation in Hoare Logic [39]. However, since CaSym soundly converts loops into loop-free counterparts before symbolic execution, verification in CaSym does not require explicit loop invariants, which is required in traditional postcondition calculation in Hoare logic.

A. Loop-free programs

A loop-free program has a finite number of paths. For a path i, symbolic execution computes a symbolic state $\sigma_i(\overline{x}, c, \overline{X}, C)$ at the end of the path. For a finite number of paths, CaSym could combine the symbolic states of all paths through disjunction; however, this would not be scalable. Therefore, after discussing how CaSym performs symbolic execution on a single path, we present how CaSym shares parts among paths to get compact formulas through path merging.

Symbolic states: A symbolic state contains (1) the symbolic values of program variables, (2) a symbolic cache state, and (3) a path condition, which is a conjunction of boolean tests performed on the path. Specifically, a symbolic state $\sigma(\overline{x}, c, \overline{X}, C)$ is of the following form:

$$X_1 = e_1 \wedge \ldots \wedge X_n = e_n \wedge C = ce \wedge \psi$$

where e_i is a symbolic expression that represents the value in X_i and it may contain occurrences of x_1 to x_n (the initial symbolic values for X_1 to X_n). Similarly, ce is a symbolic cache state that represents the current cache state; ce may contain occurrences of c as well as x_1 to x_n. Path condition ψ may contain occurrences of x_1 to x_n. As an abbreviation, we write $\overline{X} = \overline{e}$ for $X_1 = e_1 \wedge \ldots \wedge X_n = e_n$. The initial symbolic state is

$$\text{initial}(\overline{x}, c, \overline{X}, C) \triangleq \overline{X} = \overline{x} \wedge C = c \wedge \text{True} \qquad (2)$$

We note that even though our goal is to track cache states, for precision it is important to also track program states (path conditions and symbolic values of variables). Since a program has many paths and each path can lead to a different cache state, the path condition in a symbolic state tells under what condition the program produces the associated cache state. Furthermore, tracking symbolic values of variables is important for computing accurate cache states. For instance, when the program accesses $A[X]$, knowing variable X's symbolic value is important to model what element of the array is being accessed.

Interface to cache models: To track how the input program affects the cache state, CaSym needs a cache model that specifies how the cache is affected by memory accesses. A cache implementation makes various choices about cache capacity, cache block size, associativity, and the cache-replacement policy. To accommodate cache diversity, CaSym's symbolic execution is parameterized over a cache model so that different cache models can be plugged into the system. This set up also enables abstract cache models, which abstract away certain cache-implementation details and provide generality (we introduce two abstract cache models in Section VI).

CaSym' symbolic execution interacts with a cache model through a well-defined *interface*. We postpone the discussion of how CaSym's cache models implement the interface to Section VI. For now, it is sufficient to understand a cache model's interface, listed as follows:

(i) An empty cache state, written as empty_C.
(ii) A cache-update function $\text{upd}_\text{C}(l, ce)$, which takes a symbolic memory location l (discussed soon) and a symbolic cache state ce and returns a new symbolic cache state for the result of accessing l under ce.
(iii) An equality test $\text{eq}_\text{C}(ce_1, ce_2)$. It holds if and only if the two symbolic cache states ce_1 and ce_2 are equal according to the cache model.

Two kinds of symbolic memory locations are used. The symbolic memory location for variable X is written as $\text{M}X$. The symbolic location for array A at symbolic expression e is written as $\text{M}A[e]$. As an example, for the program "$X_2 = X_1\&0\text{xFF}; X_3 = A[X_2]$", the symbolic memory location for the array access in the second statement is $\text{M}A[x_1\&0\text{xFF}]$.

Stmt	$S ::= S_1; S_2 \mid \text{SKIP} \mid X := E \mid A[X] := E \mid B \to S$
Expr	$E ::= n \mid X \mid A[X] \mid E \otimes E$
BExpr	$B ::= E \odot E \mid \neg B \mid B_1 \wedge B_2 \mid B_1 \ \vee \ B_2$

$$[\![E]\!]_\sigma = \begin{cases} n, \text{if } E = n \\ e_i, \text{if } E = X_i \text{ and } \sigma \text{ contains } X_i = e_i \\ u, \text{if } E = A[X] \text{ and } u \text{ is a fresh variable} \\ [\![E_1]\!]_\sigma \otimes [\![E_2]\!]_\sigma, \text{if } E = E_1 \otimes E_2 \end{cases}$$

Stmt S	Symbolic execution result $\text{SE}(S, \sigma)$, assuming σ is $\overline{X} = \overline{e} \wedge C = ce \wedge \psi$.
$S_1; S_2$	$\text{SE}(S_2, \text{SE}(S_1, \sigma))$
SKIP	σ
$X_i = E$	$X_1 = e_1 \wedge \ldots \wedge X_i = [\![E]\!]_\sigma \wedge \ldots \wedge X_n = e_n \wedge$ $C = \text{upd}_\text{C}(\text{locs}(E, \sigma) + [\text{M}X_i], ce) \wedge \psi$
$A[X_i] := E$	$\overline{X} = \overline{e} \wedge C = \text{upd}_\text{C}(\text{locs}(E, \sigma) + [\text{M}X_i] + [\text{M}A[e_i]], ce) \wedge \psi$
$B \to S$	$\text{SE}(S, \overline{X} = \overline{e} \wedge C = \text{upd}_\text{C}(\text{locs}(B, \sigma), ce) \wedge (\psi \wedge [\![B]\!]_\sigma))$

Figure 3: Syntax for a path language and symbolic execution over a path.

Symbolic execution over a path: To formally present symbolic execution over a program path, we introduce a small path language in Fig. 3. A statement in the language represents a list of commands in a program path. In the language, we use n for constant numbers, X for program variables, A for fixed-size arrays, $\otimes$ for a binary arithmetic operator, and $\odot$ for a binary comparison operator. We use "$A[X] := E$" for an array assignment. "$B \to S$" is a statement guarded by boolean condition B; it is the single-guard variant of guarded commands in Dijkstra's guarded command language [40]. An if-statement "IF B THEN S_1 ELSE S_2" can be split into two paths: one has "$B \to S_1$" and the other has "$\neg B \to S_2$".

Fig. 3 formalizes CaSym's symbolic execution over the path language. Its way of tracking program states (variables' symbolic values and path conditions) is standard in symbolic execution. It uses auxiliary functions $[\![E]\!]_\sigma$ and $[\![B]\!]_\sigma$ to compute the symbolic values of arithmetic expressions and boolean expressions, respectively; we omit the standard definition of $[\![B]\!]_\sigma$.

In addition, cache states are also tracked during symbolic execution. We use the following notation in Fig. 3 for tracking cache states. The cache-update function is lifted to a list of symbolic memory locations: $\text{upd}_\text{C}(L, ce)$ returns the new cache state after accessing the list of locations in L. We use $\text{locs}(E, \sigma)$ for the list of symbolic memory locations in program expression E under symbolic state σ, and similarly $\text{locs}(B, \sigma)$ for the list of symbolic memory locations in boolean expression B. We omit their straightforward definitions. As an example, if in σ we have $X = 2x$, then $\text{locs}(X + A[X], \sigma) = [\text{M}X, \text{M}A[2x]]$. Fig. 3 presents how cache states are tracked for each kind of statements. As an example, showing how cache states are tracked, take $X_i = E$, which accesses the memory locations in E and the memory location of X_i; therefore, it updates the symbolic cache state with those locations. Other cases are similar.

We next present symbolic execution for a toy example "$X_2 = X_1\&0\text{xFF}; X_3 = A[X_2]$", starting from the initial symbolic state.

$$\begin{aligned} &\{X_1 = x_1 \wedge X_2 = x_2 \wedge X_3 = x_3 \wedge C = c \wedge \text{True}\} \\ &X_2 = X_1\&0\text{xFF}; \\ &\{X_1 = x_1 \wedge X_2 = x_1\&0\text{xFF} \wedge X_3 = x_3 \\ &\wedge C = \text{upd}_\text{C}([\text{M}X_1, \text{M}X_2], c) \wedge \text{True}\} \\ &X_3 = A[X_2] \\ &\{X_1 = x_1 \wedge X_2 = x_1\&0\text{xFF} \wedge X_3 = u \\ &\wedge C = \text{upd}_\text{C}([\text{M}A[x_1\&0\text{xFF}], \text{M}X_3], \text{upd}_\text{C}([\text{M}X_1, \text{M}X_2], c)) \\ &\wedge \text{True}\} \end{aligned}$$

The end symbolic cache state σ contains a symbolic memory location $\text{M}A[x_1\&0\text{xFF}]$. Feeding $\text{VC}(\sigma, \{X_1\})$ to an SMT solver would produce two different values for x_1, resulting in two different cache states. This is a side channel caused by key-dependent memory accesses.

As a note, for scalability CaSym's symbolic execution does not track array contents symbolically. This is why $[\![A[X]]\!]_\sigma$ produces a fresh unconstrained variable, which implements an approximation (reading from an array returns arbitrary values). On the other hand, when accessing $A[X]$, CaSym uses X's symbolic value to capture which location of the array is accessed and uses that knowledge to update the symbolic cache state accurately.

CaSym employs a coarse-grained taint tracking for arrays. This means we can use the arbitrary values stored in the arrays and treat them as public values (when not tainted) or sensitive values (when tainted). Therefore, two symbolic execution traces use the same arbitrary value from a public array (i.e. these values cannot contribute to a difference in the cache state). This helps reduce false positives which would occur if we considered all array values to be sensitive.

Path merging: Simply performing symbolic execution over every path and combining the symbolic states of all

$\{X_1 = x_1 \wedge X_2 = x_2 \wedge C = c \wedge \text{True}\}$
if $(X_1 > 0)\{$
$\{X_1 = x_1 \wedge X_2 = x_2 \wedge C = \text{upd}_\text{C}([\text{M}X_1], c) \wedge x_1 > 0\}$
$X_2 = X_2 + 1;$
$\{X_{1a} = x_1 \wedge X_{2a} = x_2 + 1 \wedge C_a = \text{upd}_\text{C}([\text{M}X_1, \text{M}X_2], c)$
$\wedge(\psi_a = x_1 > 0) \wedge \psi_a\}$
$\}$
else $\{$
$\{X_1 = x_1 \wedge X_2 = x_2 \wedge C = \text{upd}_\text{C}([\text{M}X_1], c) \wedge x_1 \leq 0\}$
$X_2 = X_2 + 2;$
$\{X_{1b} = x_1 \wedge X_{2b} = x_2 + 2 \wedge C_b = \text{upd}_\text{C}([\text{M}X_1, \text{M}X_2], c)$
$\wedge(\psi_b = x_1 \leq 0) \wedge \psi_b\}$
$\}$
$\{((\psi_a \wedge X_{1c} = X_{1a} \wedge X_{2c} = X_{2a} \wedge C_c = C_a) \ \vee$
$(\psi_b \wedge X_{1c} = X_{1b} \wedge X_{2c} = X_{2b} \wedge C_c = C_b)) \wedge$
$(\psi_c = \psi_a \ \vee \ \psi_c = \psi_b) \wedge \psi_c\}$

Figure 4: Sample program illustrating how path merging is handled using the infinite cache model.

paths through disjunction at the end is not scalable as it would generate large formulas. As an optimization, CaSym performs path merging to generate formulas that share parts among program paths. In particular, when multiple paths converge at a point, it merges their symbolic states by introducing new logical variables and equations as illustrated by the example in Fig. 4. Consider two paths with path conditions ψ_a and ψ_b, the symbolic values of variables $\overline{X_a}$ and $\overline{X_b}$, and the symbolic cache states C_a and C_b. At the merge point, CaSym introduces new logical variables ψ, $\overline{X}$, and C, and adds the following equalities:

$$\psi = \psi_a \ \vee \ \psi = \psi_b \tag{3}$$

$$(\psi_a \wedge \overline{X} = \overline{X}_a \wedge C = C_a) \vee (\psi_b \wedge \overline{X} = \overline{X}_b \wedge C = C_b) \tag{4}$$

All equations reflect that either one of the paths could be taken, but each case in equation (4) is further guarded by the corresponding path condition for precision. The equation at the end of Fig. 4 follows equations (3) and (4).

The benefit of path merging is that further symbolic execution beyond the merge point can just use the newly introduced logical variables (ψ, $\overline{X}$, and C in the equations above); so all the paths beyond the merge point share the logical variables and equations, resulting in compact logical formulas. We also note that CaSym's implementation takes a control-flow graph as an input and treats every node with more than one adjacent predecessor in the graph as a merge point; as a result, new logical variables are introduced and path merging is performed before every node in the graph with this characteristic.

We note that our path mering is similar to those proposed in [41], [42] (though they do not consider merging cache states). We could further optimize the formulas by applying techniques in [43], [44], which selectively use path merging when it is adventurous to do so; but we leave that as future work.

B. Compositional reasoning

The cache-aware symbolic execution sketched above only handles programs with a finite number of execution paths. However, practical software usually have an unbounded or a large number of execution paths, making symbolic execution infeasible or inefficient. To tackle such a challenge, we introduce two novel statements to enable compositional reasoning

$$\text{Stmt} \quad S ::= \ldots \mid \text{reset} \mid \text{check } K$$

The "reset" statement resets the current symbolic state to an arbitrary initial symbolic state; "check K" directs CaSym to issue a verification condition to a solver based on the current symbolic state, assuming only the secret-variable set K carries confidential data *at the last* reset. The rules for performing symbolic execution over them are shown below:

$$\text{SE}(\text{reset}, \sigma) = \text{initial}(\overline{x'}, c', \overline{X}, C)$$
$$\text{SE}(\text{check } K, \sigma) = \sigma, \text{ and issue condition } \text{VC}(\sigma, K)$$

The definitions of $\text{initial}(\overline{x'}, c', \overline{X}, C)$ and $\text{VC}(\sigma, K)$ were given earlier in equations (1) and (2). Note that we use $\overline{x'}$ and c' to distinguish them from the initial state of symbolic execution.

The introduced "reset" and "check K" statements has several benefits:

Flexibility: The check and reset statements allow CaSym to flexibly decide where to reset to the initial state and where to check for cache-based side channels. For example, by turning S into "$\text{reset}; S; \text{check } K_0$", we tell CaSym to perform symbolic execution from the initial symbolic state and perform the side-channel check at the end, assuming that the initial secret variables are in K_0 and an attacker observes the cache state only at the end of the program. As another example, when "$S = S_1; S_2$" and the attacker can observe the cache state in the middle and at the end, we can perform symbolic execution over "$\text{reset}; S_1; \text{check } K_0; S_2; \text{check } K_0$", which triggers two side-channel checks. In the most extreme case, a check can be inserted at every control-flow point in the program, corresponding to what a trace-based attacker can observe (discussed in Section II-B).

Scalability: The new statements also enable compositional, scalable reasoning. Suppose performing symbolic execution over the entire program S produces a large formula at the end. Feeding the formula to an SMT solver may not be feasible given the amount of time that is needed to solve the formula. One way of reducing the time pressure is to break S into two parts and check them individually. Suppose $S = S_1; S_2$, we can then turn it into

$$\text{reset}; S_1; \text{check } K_0; \text{reset}; S_2; \text{check } K_1$$

The first check verifies that S_1 is free of cache-based side channels; that is, running S_1 twice with two different secrets and the same initial cache states results in the same cache states. After this check, we can reset the symbolic state and perform symbolic execution on S_2 and the check after S_2 verifies that running S_2 twice with different secrets and the same initial cache states results in the same cache states. This is a rely-guarantee reasoning since, when checking S_2, it relies on the assumption that the initial cache states are the same and the assumption is discharged by the verification on S_1. The compositional reasoning is more scalable than checking $S_1; S_2$ as a whole, since the reset in the middle throws away the symbolic state. However, it may cause false positives when checking S_2 due to the loss of information.

One complication in the above process is that the two check statements are with respect to two separate secret-variable sets: the first check assumes K_0 is the secret-variable set at the beginning, while the second check assumes K_1 is the secret-variable set at the point between S_1 and S_2. The two sets might be different; for instance, K_0 might be $\{X_1\}$, and, if S_1 copies X_1 to X_2, then the set of secret variables after S_1 becomes $\{X_1, X_2\}$. In general, the set of secret variables may change due to secret propagation in a program. To soundly estimate the set of secret variables, CaSym has a static taint tracking component, which takes initial secret variables and outputs the set of secret variables at each program location. This was implemented by a standard flow-sensitive dataflow analysis in LLVM. With the result of this analysis, CaSym knows the secret-variable set at each location, including K_1.

C. Transforming loops

Symbolic-execution systems for bug finding only explore a limited number of paths. Hence, they do not guarantee a coverage of all paths for programs with loops. With the help of the new statements introduced in Section V-B, CaSym transforms programs with loops into loop-free programs. Specifically, the transformation works as follows:

$$
\begin{array}{l}
S_1; (\text{WHILE } B \text{ DO } S); S_2 \\
\quad \Rightarrow \\
S_1; \text{check } K_0; \text{reset}; \\
(\text{IF } B \text{ THEN } (S; \text{check } K_1) \text{ ELSE SKIP}); \\
\text{reset}; \\
\neg B \rightarrow S_2
\end{array}
$$

This transformation is sound (i.e., the original program is side-channel free whenever the transformed one is) since the loop-free program enforces the following invariant in the original program: any two executions of S starting from the same initial cache state results in the same final cache state. From the Hoare logic point of view, this is the cache-state loop invariant checked by CaSym. Note that this is performed with respect to K_1, the set of secret variables right before the loop body S (this is determined by tracking how values of secret variables propagate in the original program, as discussed before).

To see why the invariant is enforced, the first check makes sure that the initial cache states are identical before the loop (i.e., S_1 is side-channel free). After that, the symbolic state is reset. Hence, statement $(\text{check } K_1)$ ensures that there is no side channel for the loop body starting from any memory and cache state. After checking the loop body, the symbolic state is reset again so that the verification of S_2 assumes nothing after the if statement, which semantically represents the memory and cache state after zero or one loop iteration. After that, $\neg B$ can be assumed when checking S_2.

In theory, the transformation may cause some false positives. For example, the transformation assumes nothing on the initial memory and cache state before each iteration, which may cause false positives. But in practice, we have found only one false positive due to the transformation in database systems (Section IX). Moreover, when a loop has a constant number of iterations, we can also unroll it for better precision.

VI. Cache Models

CaSym takes a cache model and identifies potential side channels based on it. In principal, it can take any cache model with sufficient abstractions in place: the empty cache state, the cache-update function, and the equality-testing function. In this section, we introduce two novel *abstract* cache models: the infinite cache model and the age model. We also discuss how to support more concrete models, such as the LRU model, used in previous work.

A. Abstract vs. concrete cache models

Concrete cache models (e.g., LRU, FIFO, PLRU models used in [12]) accurately model details such as the replacement policy of the expected architecture that a program will be executed on. While such detailed models allow accurate reasoning about the cache state (i.e., existence or absence of data in the cache), one downside is that the verified programs are secure *only* on those expected architectures. For example, a crypto implementation that is side-channel-free on cache with LRU might have side channel on cache with FIFO. Moreover, reasoning over a concrete model likely will cause scalability issues for static program analysis.

Another approach is to use a higher-level abstraction, such as the entire trace of memory accesses [13], [11]. Doing so is architecture-independent and sufficient since in all realistic architectures, cache state is determined by the trace of memory accesses. However, this approach may be too conservative, since the footprint of secret dependent memory accesses might be erased by later accesses before an attacker probes the cache.

We propose two novel cache models that offer good balance between precision and generality. The infinite cache model represents an optimistic view of the cache: if there

is a side channel under this model, then the side channel likely will exist in realistic cache models (i.e., they are high-priority side channels that may show up on most architectures); the age model represents a pessimistic view of the cache: if there is a side channel under the this model, then the side channel likely will exist in some realistic cache model (i.e., they are low-priority side channels that may show up on some architectures). Empirical results suggest that the infinite cache model and the age model achieves a good balance between analysis scalability and precision (Section IX).

B. Infinite cache model

This is an idealized cache model with an infinite size and associativity, so that it never evicts any data that is already being cached. This is clearly idealized, but it is also interesting since it represents an *optimistic view* of cache: if there is a side channel under the infinite cache model, the side channel likely will exist in other more realistic cache models. Moreover, it is the (conceptual) model that cryptography software writers have in mind when they apply software countermeasures to cache-based side channels. One example is *preloading* in cryptography software, which we detail in Section VII-B1. Furthermore, empirical results suggest that the infinite cache model offers a significant speedup with few false negatives on both crypto systems and database systems, compared with more conservative and realistic cache models.

In the infinite cache model, a cache state is represented as a set of symbolic memory addresses for program variables and array elements.

- The empty cache is the empty set: $\mathrm{empty_C} = \{\}$.
- The cache-update function is implemented as set union: $\mathrm{upd_C}(l, ce) = \{l\} \cup ce$.
- The cache-equality testing becomes set equality:

$$\mathrm{eq_C}(ce_1, ce_2) = \forall l, l \in ce_1 \leftrightarrow l \in ce_2$$

To see why this model is more optimistic than other more realistic models, we note that ce_1 and ce_2 are different only if there is some address l that is accessed in one execution but not in the other, starting with different confidential data. Except for a *fully-associative* cache, that implies if the compiler maps l to some cache set and maps all other addresses to other cache sets, most cache replacement policies will result in a difference in the cache set that l gets mapped to. Hence, this model gives a "lower bound" on side channels among various cache models.

C. Age model

Unlike the optimistic infinite model, the age model is on the *pessimistic* end: for each symbolic memory location, it tracks the distance to its most recent access, called the *age*. The recently accessed location has age zero, while the second recently accessed location has age one, and so on. In this model, a cache state is a map from symbolic memory locations to their ages:

- The empty cache maps all memory locations to infinity: $\mathrm{empty_C} = \lambda l.\ \infty$.
- The cache-update function marks the current location's age to be zero and increments other locations' ages by one: $\mathrm{upd_C}(l, ce) = \lambda l',$ if $l' = l$ then 0 else $ce(l') + 1$
- The cache-equality tests equality of ages: $\mathrm{eq_C}(ce_1, ce_2) = \forall l, ce_1(l) = ce_2(l)$.

The age model is the opposite of the infinite cache model: while the infinite cache model may miss (less crucial) side channels that only manifest themselves for some particular caches, the age model captures all potential side channels for most caches.

Property 1. *If there is no cache-based side channel on the age model, then there is no cache-based side channel for any cache replacement policy that replaces cache lines based on the most recent accesses, such as LRU.*

To see why, we note that the final cache expression ce tracks the sequence of the *last* access to each memory address. For any cache replacement policy that depends only on the latest usage of memory addresses, such as LRU, it implies that the final cache state can be expressed as a function of ages. Hence, $\mathrm{eq_C}(ce_1, ce_2)$ implies the same cache state under those policies.

For a trace-based attacker, a stronger result holds:

Property 2. *For a trace-based attacker, no cache-based side channel on the age model implies no cache-based side channel for any* deterministic *cache replacement policy (i.e., a replacement policy that can be expressed as a function of memory address traces), such as FIFO and LRU.*

The reason is that there is a one-to-one mapping between a sequence of ages (for all symbolic locations), and a sequence of memory locations being accessed. Consider a sequence of ages $A = \{ce_1, ce_2, \ldots, ce_n\}$ as well as a sequence of memory locations being accessed, say $T = \{t_1, t_2, \ldots, t_n\}$. Then, we can construct A from T as follows:

$$ce_i = \lambda l.\ \text{if } t_i = l \text{ then } 0 \text{ else } ce_{i-1}(l) + 1$$

Also, we construct T from A as follows: $t_i = l$ iff $ce_i(l) = 0$ (note that exactly one l in ce is 0 at any program point).

D. More concrete models

While our infinite cache and age models are capable of detecting side channels, we show how to enrich a cache model in CaSym if more cache details (such as cache line size, associativity and replacement policy) are needed for precision reasons.

Cache line size: To model cache lines, we simply take the index into the array and compute the cache-line-granularity location being accessed. More specifically, suppose that an integer array A is aligned and location

$A[X]$ is being accessed and x is the symbolic value of X. We simply use $(x/LINESIZE)$ as the location l in the $\mathrm{upd_C}(l, ce)$ interface to the cache models above, where $LINESIZE = 64/4 = 16$ assuming 4-bytes integer and 64-bytes cache line.[6] Note that cache line size only affects array accesses, since the memory layout for other variables are unknown at the IR level.

Cache associativity: To model cache associativity, we model the cache state ce as a collection of non-overlapping cache sets (i.e., $ce = [c_1, c_2, \ldots, c_W]$). The empty cache and equality test on ce is simply the lifted definition of those on each cache state. For the update function, let way be a function that maps an array index to the corresponding cache set (the definition of way depends on cache configuration), then the following formula illustrates how the new cache state would be computed when an array is accessed.

$$\mathrm{upd_C}(MA[X], ce) = \mathrm{upd_C}(MA[X], ce[way(X)])$$

LRU replacement: In the LRU model, the cache state is still modeled as a map from symbolic memory locations to their ages. The empty cache and the cache-update function remain the same as the age model. The cache-equality test, however, is changed to

$$\mathrm{eq_C}(ce_1, ce_2) = \forall l, ce_1(l) < n \leftrightarrow ce_2(l) < n.$$

to reflect the fact that if l is in the final cache state or not. Here we assume n is the cache size.

VII. Localizing and Mitigating Side Channels

As discussed so far, satisfiable constraints at a certain program point suggest potential side channels. Although such a binary decision helps to some extent, one novel feature of CaSym is the ability to help programmers localize the cause of the identified side channels as well as to mitigate them.

A. Localizing side channels

To localize and explain the causes of the identified side channels, we leverage the key information generated by the SMT solver: a model of constraints. A model consists of the concrete values for each constraint variable in the inequality test $\neg(\mathrm{eq_C}(ce_1, ce_2))$. According to the way that the formulas are built, a model consists of two sets of values (one used by ce_1 and one used by ce_2) that will lead to different cache states. We refer to those two value sets as M_1 and M_2 respectively. Our localization algorithm proceeds in two steps.

[6]The computation assumes row-major layout for arrays. Column-major layout can be handled in similar way.

Compute shared path: We first use M_1 and M_2 to reconstruct two execution paths taken according to the model reported by the solver. This is possible since CaSym keeps track of the path condition for each basic block. Given M_1 and M_2, we can tell which basic block is in a path by checking the validity of its path condition. Based on that, we compute the *shared* blocks between them. Finally, we traverse the control flow graph in topological order to recover a path of those *shared* blocks. We call the path the *shared path*, denoted by SP.

Discover divergence: In the second step, we first find all symbolic addresses which are different in the final cache states. That is, we use M_1 and M_2 to find addresses such that they make a difference between ce_1 and ce_2. Intuitively, these are the problematic memory addresses that we need to localize the error cause for each of them.

For each problematic memory address l, the localization algorithm reports the first point in SP, say an instruction c, so that the abstract cache states of l are different for the next point in SP, but are identical for the previous point in SP. Such an instruction is reported as the root cause of the side channel at l. Note that when multiple addresses may cause side channels, our algorithm reports multiple instructions in a program, following the same procedure for each l that causes a difference between ce_1 and ce_2.

Example: Fig. 1 shows a simple example with two side channels. For the final cache state, an SMT solver reports a model where $RK[3] = 256, key[0] = 255$ in value set M_1 and $RK[3] = 0, key[0] = 0$ in value set M_2. Based on the model and tracked path conditions in the control flow graph, the localization algorithm constructs a shared path consisting of the blocks in grey, shown in Fig. 5.

In this example, the final cache state differs for several symbolic addresses, including `Sbox[0]`, `base` and so on. For the address `Sbox[0]`, the first point in the shared path that ce differs between the two execution paths is after the assignment `RK[4]=RK[0]^Sbox[(RK[3] >> 8) & 0xFF`, which is the correct location to blame for the cache difference of address `Sbox[0]`. For the address `base`, the first point in the shared path that ce differs between the two execution paths is after the branch condition `bit_set_at_i(key[0],i)`. This is the correct location to blame as well, since the secret dependent branch caused cache difference. For other problematic addresses, the localization algorithm points to those two problematic instructions as well in this example.

B. Fixing side channels

The localized causes of side channels enable a programmer or a compiler to fix the identified side channels. We explore two commonly used techniques for cache-based side channel mitigation in this section and show how the localized error causes facilitate error fixing.

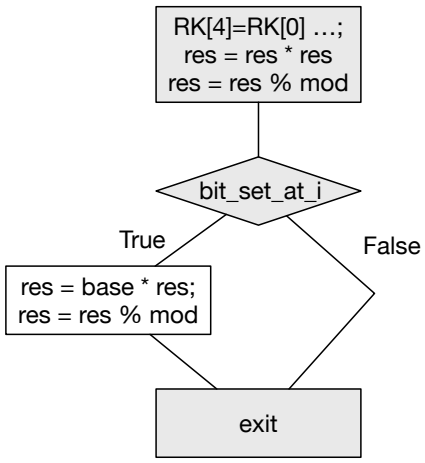

Figure 5: Shared path computed from the model generated by the SMT solver for the code in Fig. 1.

1) Preloading: Preloading eliminates cache-based side channels by loading certain memory addresses before the vulnerable instructions [2], [45]. It is typically used in AES implementations, where all SBox tables fit in cache. In AES, those tables only contain public data, but indexes used to access the tables are key-dependent, which enables an attacker to infer the key based on the footprints of the AES implementation on cache [14], [2], [15]. This is similar to the code in Example 1. To mitigate such attacks, AES implementations preload the entire lookup tables into the data cache before the actual encryption/decryption. That is, they insert code that accesses every table entry to ensure all table data are in the cache before encryption/decryption starts. Hence, even if there are key-dependent table lookups, they will not affect the cache state as long as all table entries are already in the cache initially and they are not evicted during encryption/decryption.

2) Pinning: Pinning prevents cache misses on the data that is explicitly "pinned" in a program. For instance, this feature can be implemented in a customized cache, where a cache entry with the "pin" bit set is never evicted [32]; it can also be implemented on some commodity hardware with Hardware Transactional Memory (HTM) [46]. Compared with preloading, pinning provides extra assurance that pinned data will not be evicted until it is explicitly "unpinned" in the program. Previous work has shown that pinning can be use to defend against cache-based side channels [32], [46].

3) Fixing side channels: To support preloading and pinning, CaSym introduces special instructions in the form of `PRELOAD` l and `PIN` l, which semantically preload/pin the corresponding symbolic addresses into cache (when l is an array, the instruction preloads/pins all elements in the array).

The localized root causes of side channels makes it straightforward to insert needed preloading/pinning instructions to remove side channels: if the vulnerable point for memory address l is an instruction c, then preload/pin l before c will remove the counterexample found by the SMT solver. For example, preloading/pinning the entire `SBox` table right before its vulnerable point at line 3 in Fig. 1 as well as preloading `base, res, mod` right before their shared vulnerable point at line 6 in Fig. 1 (found by the localization algorithm in Section VII-A) will remove the side channels in this program.

Although fixing side channels seems easy with the help of CaSym, we emphasize that finding *where and what* to preload/pin is nontrivial without CaSym, since identifying what data to preload can be difficult. For example, the crucial data in AES is the lookup table, which only stores public information. Moreover, for preloading, fetching the data too early may cause the data to be evicted before the vulnerable instructions, which undermines the effect of preloading.

VIII. Implementation

CaSym is implemented inside LLVM [47] as a compiler pass that performs cache analysis and error localization. It analyzes LLVM IR code and performs symbolic execution to build a cache formula. We use the Z3 SMT solver [48] to check the satisfiability of the cache formula, but any SMT solver with theories for bit-vectors and arrays could suffice.

The compiler pass of CaSym sends to the Z3 solver (in the same process as the compiler pass) the cache formula as an in-memory object. If the formula is satisfiable (meaning there is a side channel), the solver generates a model containing the assignments for the formula's variables. Using this model, CaSym localizes the vulnerable LLVM IR instructions. CaSym then uses the debugging information to report the corresponding line numbers in the source program to the user.

IX. Evaluation

We evaluated CaSym on a set of crypto and database benchmarks. All experiments were run on Ubuntu 14.04 in a virtual machine with 16 GB of RAM and an Intel i7-5820K CPU at 3.30 GHZ. During evaluation, we were mostly interested in answering the following questions:

1) how effective is CaSym in identifying cache-based side channels and how accurate are the results?
2) how efficient is CaSym and whether it can generate useful results within a reasonable amount of time?
3) how do different cache models compare when identifying side channels?
4) how well does CaSym's error localization perform?
5) whether CaSym can validate the result after applying prefetching or pinning to fix a side channel?

For benchmarks, we collected realistic crypto implementations from popular libraries, including Libgcrypt 1.8.1, mbed TLS 2.6.0, and glibc 2.26. These benchmarks can be

Benchmarks	LOC	Access: Infinite	Access: Age	Trace: Infinite TP	Trace: Infinite FP	Trace: Infinite t(s)	Trace: Age TP	Trace: Age FP	Trace: Age t(s)
AES gcry	182	☹ 0.30	☹ 0.60	64	0	8.9	64	0	16.7
AES mbed	220	☹ 0.14	☹ 0.34	17	0	5.9	17	0	17.0
triple-DES gcry	127	☹ 0.06	☹ 28.70	128	0	62.5	128	0	189
triple-DES mbed	111	☹ 0.12	☹ 22.9	48	0	27.0	48	0	73.2
DES glibc	114	☹ 0.08	☹ 122.9	2	0	0.92	2	0	2.65
UFC glibc	41	✓ 0.01	✓ 0.02	0	0	0.24	0	0	1.27
sqr-alwys-mul gcry	131	☹ 0.20	☹ 0.66	3	0	18.9	4	0	184
sqr-mul gcry	130	☹ 0.25	☹ 0.44	4	0	8.2	4	0	125
LR-mod-expo gcry	208	☹ 0.61	☹ 6.39	3	0	84.8	3	0	2618

Table I: Evaluation results for access and trace-based attackers for crypto benchmarks. For access-based results, ☹ means a side channel identified and ✓means no side channel identified; it also reports the amount of time in seconds for CaSym to perform side-channel checking. For trace-based results, the TP column identifies the number of true positives found. Similarly FP is the number of false positives. The third column for each models depicts the amount of time (in seconds) for each test.

Benchmarks	LOC	Access: Infinite	Access: Age	Trace: Infinite TP	Trace: Infinite FP	Trace: Infinite t(s)	Trace: Age TP	Trace: Age FP	Trace: Age t(s)
advance array keys	99	☹ 0.26	☹ 253	5	0	1.33	5	0	9.89
binsrch	126	✓ 0.06	☹ 115	0	0	0.46	1	0	1.90
compare	174	☹ 1.24	☹ 5.84	7	0	0.31	8	0	19.9
find xtreme element	117	✓ 0.38	☹ 2.52	0	0	0.62	1	0	2.33
heap key test	118	☹ 0.07	☹ 3.47	3	0	0.33	3	0	0.53
is equal	89	☹ 0.12	☹ 3.80	1	0	1.68	1	0	1.32
mark array keys	34	✓ 0.01	✓ 0.13	0	0	0.08	0	0	0.27
sort array elements	144	☹ 0.81	☹ 5.44	2	1	4.17	4	1	6.74
start array keys	42	☹ 0.01	☹ 2.43	2	0	0.43	2	0	3.27

Table II: Evaluation results for access and trace based attackers for the PostgreSQL database.

roughly divided into two categories: encryption using symmetric ciphers and modular exponentiation using asymmetric ciphers. In order to evaluate CaSym on other less scrutinized codebases, we also analyzed functions from the PostgreSQL 10.2, which is a popular database back end.

Symmetric cipher benchmarks: We include six benchmarks: AES gcry, AES mbed (the 128-bit-key AES encryption in Libgcrypt and mbed TLS respectively), triple-DES gcry, triple-DES mbed (the triple-DES encryption in Libgcrypt and mbed TLS respectively), DES glibc (the DES encryption in the glibc library) and UFC glibc (the ultra fast encryption algorithm in glibc).

Asymmetric cipher benchmarks: Given a base b, an exponent e, and a modulus m, modular exponentiation computes "b^e mod m". The majority of asymmetric encryption such as RSA and ElGamal performs modular exponentiation.

Computing modular exponentiation directly would be rather costly in both time and space. Libgcrypt implemented three versions of efficient modular exponentiation, which we call sqr-alwys-mul gcry, sqr-mul gcry, and LR-mod-expo gcry. The first two versions implement the square-and-multiply method; the main difference between the two is that the square and multiply algorithm only performs the multiplication when the bit being processed is set, while the square-and-always-multiply algorithm performs the multiplication regardless of whether the current bit is set. The final version implements the left-to-right k-ary method [49]. Following previous analysis on crypto implementations [13], we did not analyze the code of library implementing multi-precision integers (MPI).

Database benchmarks: The crypto benchmarks are typically well scrutinized for side channels. To see how CaSym performs on other codebases, we consider PostgreSQL, a popular database used by many applications. For this database system, we treat the primary keys (identifiers for records in a database) as sensitive, since they are commonly account numbers, social security numbers, etc.

The entire PostgreSQL contains over 1 million lines of code. We narrowed down the scope to a set of functions that process likely sensitive data. In particular, we investigated the binary tree implementation under the `/src/backend/nbtree` directory. Under this directory, there are 16 functions that use the primary key of a record. Of those 16, 7 pass the key onto other functions without processing it. Therefore, we evaluate on the remaining 9 functions which actually process the primary keys. The functions are: advance arrays keys, binsrch, compare, find xtreme element, heap key test, is equal, mark array keys, sort array keys, and start array keys.

A. Evaluation for access-based attackers

Recall that an access-based attacker observes only the final cache state of the victim program. Therefore, CaSym performs the verification-condition check only at the end of each benchmark. We evaluated benchmarks based on the two abstract cache models: the infinite model and the age model (we will compare the abstract cache models with concrete cache models in Section IX-D).

Cryptography Benchmarks: Table I presents the results on cryptography benchmarks. We note that we have removed preloading that is present in the original AES and triple-DES code; we will report separately the results when it is present. UFC and DES in glibc do not use preloading.

From the table, we observe that CaSym can finish side-channel checking rather quickly for most of the cases: all checks under the infinite cache model finish under 1 second; those under the age mode are slower, but all finish in about 2 minutes. Moreover, we note that the optimistic infinite cache model gives exactly the same result as the more conservative age model.

The side channels reported for the two AES, two triple-DES and one DES implementations are due to key-

dependent array accesses (we discuss examples in the next section). Side channels are also reported for the three modular-exponentiation algorithms, since they contain secret-dependent branches (examples in the next subsection). These are previously known side channels and CaSym confirms their presence.

We note that DES-glibc contains a side channel that was *newly found* by CaSym; it results from key-dependent memory accesses to a lookup table that was not preloaded (`key` is the private key):

```
t=r^key[i + 1]; ...
l^= des_SPtrans[1][(t)&0x3f]| ...;
```

UFC-glibc is the only benchmark that is side channel free with respect to the cache without extra security mechanisms. This is because it does not use any precomputed table and is virtually straight line code. It illustrates how different techniques can avoid side channels.

Database Benchmarks: Table II presents the results on the 9 functions we analyzed for an access based attacker. Consistent with the results on cryptography benchmarks, analysis based on the infinite cache model is more efficient: on average, the age model takes over 100 times longer. Also, most of the functions that we test are potentially vulnerable to side channel attacks (examples in the next subsection).

Interestingly, there are two functions (binsrch & find xtreme element) in which the infinite cache model missed two positives reported by the age model. The cause is a key dependent branch which accesses different locations but they were previously used, and thus already in the cache. This example demonstrates the difference of those two abstract models: the infinite cache model is optimistic (that is, it optimistically assumes that the loaded memory locations were not evicted before the sensitive branch), while the age model is pessimistic (that is, it pessimistically assumes that the loaded memory locations were evicted before the sensitive branch).

B. Evaluation for trace-based attackers

Recall that a trace-based attacker can observe the intermediate cache states of the victim program. For each benchmark, we ran CaSym to perform a check on the symbolic state after every statement (following a topological order of the program's control-flow graph); CaSym then stopped at the first point where a side-channel was found. Since CaSym's error reporting includes the source line number where the side channel is, we then went to that line and fixed the problematic statement, as described in Section VII-B. Then we applied CaSym on the fixed program to find the next side channel. Through this iterative process, we were able to find a set of independent side channels in each benchmark.

Cryptography Benchmarks: Table I reports the number of side channels identified on the cryptography benchmarks. Similar to the results for access-based attacks, we found that the infinite cache model is more efficient than the age model, while they provide very close results (we discuss the only difference in sqr-alwys-mul shortly).

We inspected the error-reporting results and manually confirmed that for all cases, CaSym localized the side channels to the right lines and also confirmed that all reported side channels are true side channels. It was a surprising result considering that CaSym's symbolic execution approximates a program's behavior, for example, when the program reads from arrays; also our loop transformation could also introduce false positives.

Next, we take a closer look at the positives. The side channels CaSym found in symmetric ciphers are due to the sbox tables being indexed by key-dependent variables in all AES/DES implementations. Below is a representative example from the AES Libgcrypt benchmark where `encT` is the encryption table and the array index `sa[0]` is derived from the key.

```
sa[0] = sb[0] ^ key[0][0];
... encT[sa[0] >> (0 * 8)] ...
```

The reason why the number of side channels for symmetric ciphers are high is because they contain multiple lines of code following the same sbox table access pattern as above.

For the three modular exponentiation algorithms implemented in Libgcrypt, CaSym found multiple side channels. The side channels are due to either array accesses indexed by the exponent bits or branches whose outcome depends on a key. A similar situation happens for the left-to-right algorithm (LR-mod-expo gcry). A simplified code snippet that depicts the issue can be found in Listing 1.

```
if (c >= W) // c is tainted by the key
  c0 = 0;
else {
  e0 = (e >> (BITS_PER_MPI_LIMB - c));
  j += c - W; }
```

Listing 1: LR-mod-expo Example

Interestingly, we notice that CacheD [11] reports no side channels in Libgcrypt 1.7.3. The reason is two fold. First, two algorithms (sqr-alwys-mul gcry and sqr-mul gcry) are not used in the default configuration of Libgcrypt. Since CacheD [11] only explores side channels exhibited in an execution trace, side channels in those algorithms are missed. Second, CacheD [11] does not detect side channels due to secret-dependent branches (e.g., the side channels in the code snippet above), since it detects cache difference only for executions that follow *the same control flow*. The example in Listing 1 confirms that CaSym avoids the coverage issue of CacheD.

To assess false negatives, we treat the trace-based age model result as the ground truth, since under this model, no positive implies no cache-based side channel on most realistic caches (Property 2). Based on the results, the only case where the infinite cache model had a false negative

was for the square and always multiply implementation. The relevant code snippet can be found in Listing 2.

```
res = res * res; res = res % mod;
temp = res * base; temp = temp % mod;
if (((1 << 31 ) & expo) != 0) {
   res = temp;}
```

Listing 2: A False Negative of the Infinite Cache Model

Database Benchmarks: The results on database benchmarks are summarized in Table II on the right. Both of our models perform well in the trace based attacker analysis. Again, analysis based on the infinite cache model is more efficient than the age model, while they provide very close results. In our evaluation, CaSym identified potential vulnerabilities in most benchmarks.

Among all positives, we do find one false positive in a function that sorts array elements. The false positive presented in Table II is due to an if-statement where one branch has a reset (due to our loop transformation) and the other one does not. Due to the information lost from issuing a reset, CaSym reports a side channel but in fact there is not one at this point. This causes two different scopes to be compared at the merge point, which causes a side channel reported. The false positve reported by CaSym is presented in Listing 3.

```
if (nelems <= 1) {
  return nelems;}
elemtype = skey->sk_subtype;
if (elemtype == InvalidOid) {
  elemtype = rd_opcintype[skey->sk_attno - 1];
}
...
last_non_dup = 0;
for (i = 1; i < nelems; i++) {
...
} // reset after loop
return last_non_dup + 1;
```

Listing 3: A False Positive in PostgreSQL

```
for (i = 1; i <= keysz; i++) {
  datum = index_getattr(itup, attno, itupdesc,
                        &isNull);
  result = DatumGetInt32(FunctionCall2Coll(
             &scankey->sk_func,
             scankey->sk_collation, datum,
             scankey->sk_argument));
  if (result != 0) // result is tainted
    return false; // early exit
  scankey++;}
```

Listing 4: A True Postive in PostgreSQL

Based on the infinite cache model, our analysis was able to detect 20 unique *newly found* locations in the source code which could leak information to an attacker about a database key. These 20 reported locations contain both secret-dependent array accesses and secret-dependent branches. The former usually happens when a variable is tainted by the primary key and then used as an index in an array. The code snippet shown in Listing 4 shows a common pattern of a key-dependent branch in PostgreSQL.

Benchmarks	**Preloading**						**Pinning**					
	Infinite			Age			Infinite			Age		
	TP	FP	t(s)	TP	FP	t(s)	TP	FP	t(s)	TP	FP	t(s)
AES gcry	0	0	2.95	64	0	17.4	0	0	4.02	0	0	13.6
AES mbed	0	0	1.68	17	0	17.4	0	0	2.00	0	0	9.60
triple-DES gcry	0	0	84.0	128	0	170	0	0	0.61	0	0	1.53
triple-DES mbed	0	0	1.53	48	0	65.5	0	0	0.03	0	0	1.70
DES glibc	0	0	0.56	2	0	3.15	0	0	0.51	0	0	1.79

Table III: Evaluation results for symmetric ciphers with table preloading and pinning for trace-based attackers.

The age model was able to detect even more positives in the benchmarks, *finding 5 new* potential vulnerabilities. This model found more locations since it considers temporal differences of when memory locations can be placed into the cache. Even though this model is more conservative, we see that it does not introduce any new false positives in our experiments.

C. Fixing side channels

We next discuss our experience of fixing the side channels that were discovered in the previous experiments. In symmetric ciphers, one common strategy to avoid side channels resulting from key-dependent table lookups is to preload their sbox tables. To support that, our LLVM implementation supports a special `PRELOAD` attribute, which can be used by programmers to annotate their source code to specify what variables or arrays should be preloaded at the start of execution. A preloaded variable/array means that it is initially in the cache. We used this attribute to annotate our symmetric-cipher benchmarks and reevaluated them using CaSym. Table III presents the results of performing preloading in symmetric ciphers for trace-based attackers. We did not evaluate preloading/pinning for asymmetric cipher benchmarks since other techniques (e.g. scatter/gather) are used to secure them [13]. Also, we did not include the UFC implementation since it is already side channel free.

As expected, Table III shows that preloading is sufficient to eliminate the side channels in the infinite cache model: after the sbox tables are preloaded, they always stay in the cache and the following key-dependent table accesses will not change the cache state. For the age model, side channels still exist since the age model tracks the ordering of memory accesses; the preloading at the beginning will not change the ordering of memory accesses in the following execution.

As we have discussed, another strategy for fixing side channels is to pin some data in the cache. We also implemented a special attribute for programmers to specify what variables/arrays should be pinned into the cache. Table III presents the results of performing pinning of tables in symmetric ciphers for trace-based attackers. It shows that all side channels disappear with this fix, across all cache models. When cache entries are pinned to the cache, they are never evicted; therefore, CaSym does not update the ages

	Access	Trace		
Benchmarks w/o Preloading	LRU	LRU		
		TP	FP	t(s)
AES gcry	timeout	64	0	635
AES mbed	timeout	17	0	757
triple-DES gcry	☹ 1654	128	0	54.3
triple-DES mbed	☹ 8531	48	0	803
DES glibc	☹ 1044	2	0	9.20
UFC glibc	✓ 0.09	0	0	5.35
sqr-alwys-mul gcry	☹ 2.64	3	0	180
sqr-mul gcry	☹ 2.19	4	0	163
LR-mod-expo gcry	☹ 23.45	3	0	6275

Table IV: Evaluation results using the LRU model; the table shows results for an access based attacker on the left and a trace based attacker on the right.

of pinned cache entries. The ages at the beginning are the ages at the end for the pinned entries, preventing CaSym from reporting them as causes of side channels.

D. Abstract vs. Concrete Cache Models

In order to show the effectiveness of the infinite cache model and the age model, we implemented the following features as described in Section VI-D: the LRU replacement policy, cache lines, and cache associativity.

LRU replacement policy: Table IV shows the evaluation results on our cryptography benchmarks, based on the LRU cache model with 2k cache slots. We note that the LRU model takes a significant amount of time to finish (the SMT solver even timed out with a limit of 3 hours for three tests). Despite the significantly longer execution time, for all tests that finish, we note that it reports exactly the same results as the age model, and very similar results as the infinite cache model. This result demonstrates that our abstract cache models offer better balance between precision and efficiency, when compared to more concrete cache models.

Cache line size and cache associativity: We also implemented cache models with various cache line sizes and cache associativity, as discussed in Section VI-D. We tested our benchmarks using common cache associativity values and cache line sizes. The results reported were the same as without specifying associativity or cache line size[7].

X. Limitations and Future Work

As mentioned before, the symbolic execution of CaSym does not track array contents. That is, reading from an array returns an arbitrary value. This in theory can cause false positives, but has not caused problems during our experiments. The implementation also does not support dynamic allocation and deallocation of memory. Array sizes must be statically declared. Our support of pointers is also limited. We require that a pointer variable must be initialized to the base address of some array. Pointer arithmetic on the pointer is allowed, but the pointer can only reference locations inside the array it was initialized to for the entire lifetime of the pointer. This reflects how pointers are used in crypto applications, but it is not the case for general C/C++ applications. Finally, CaSym inlines all functions before performing symbolic evaluation. We plan to gradually lift these restrictions so that CaSym can support more general applications beyond crypto applications.

[7]This does not mean different associativity or cache line sizes will never impact the result; it just means that typical values for associativity and cache line sizes have no impact on the benchmarks we evaluated.

XI. Conclusions

In this paper, we present CaSym for identifying and mitigating cache-based side channels. We show that CaSym's symbolic execution and cache models are effective at identifying cache-based side channels in realistic benchmarks, including cryptography implementation and database systems. CaSym was able to detect new side channels as well as known ones in multiple functions used to handle confidential data frequently. The novel abstract cache models are shown to provide good balance between precision and efficiency. Furthermore, CaSym produces accurate and helpful error reports when side channels are identified. The reports are used to strategically place mitigation mechanisms to eliminate the side channels in programs.

Acknowledgment

The authors would like to thank anonymous reviewers for their constructive feedback. This work was partially supported by NSF grants CCF-1822923, CCF-1439021, CCF-1629915, CCF-1626251, CCF-1629129, CNS-1702760, CNS-1816282, CCF-1723571, CNS-1408826, CNS-1801534, ONR grant N00014-17-1-2539, as well as a gift from Intel Corporation.

References

[1] D. J. Bernstein, "Cache-timing attacks on AES," 2005.

[2] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of AES," *Topics in Cryptology–CT-RSA 2006*, pp. 1–20, Jan. 2006.

[3] C. Percival, "Cache missing for fun and profit," in *BSDCan*, 2005.

[4] Y. Xu, M. Bailey, F. Jahanian, K. Joshi, M. Hiltunen, and R. Schlichting, "An exploration of l2 cache covert channels in virtualized environments," in *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop*, 2011, pp. 29–40.

[5] Z. Wu, Z. Xu, and H. Wang, "Whispers in the hyper-space: High-speed covert channel attacks in the cloud." in *Proceedings of USENIX Security symposium*, 2012, pp. 159–173.

[6] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, 2009, pp. 199–212.

[7] Y. Xiao, M. Li, S. Chen, and Y. Zhang, "Stacco: Differentially analyzing side-channel traces for detecting ssl/tls vulnerabilities in secure enclaves," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017, pp. 859–874.

[8] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-vm side channels and their use to extract private keys," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, 2012, pp. 305–316.

[9] Y. Yarom and K. Falkner, "Flush+reload: A high resolution, low noise, l3 cache side-channel attack," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, 2014, pp. 719–732.

[10] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. Lee, "Last-level cache side-channel attacks are practical," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, May 2015, pp. 605–622.

[11] S. Wang, P. Wang, X. Liu, D. Zhang, and D. Wu, "CacheD: Identifying cache-based timing channels in production software," in *Proceedings of the the 26th USENIX Security Symposium (USENIX Security)*, 2017, pp. 235–252.

[12] G. Doychev, D. Feld, B. Kopf, L. Mauborgne, and J. Reineke, "Cacheaudit: A tool for the static analysis of cache side channels," in *Proceedings of the the 22nd USENIX Security Symposium (USENIX Security)*, 2013, pp. 431–446.

[13] G. Doychev and B. Köpf, "Rigorous analysis of software countermeasures against cache attacks," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017, pp. 406–421.

[14] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games—bringing access-based cache attacks on AES to practice," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2011, pp. 490–505.

[15] E. Tromer, D. Osvik, and A. Shamir, "Efficient cache attacks on aes, and countermeasures," *Journal of Cryptology*, vol. 23, no. 1, pp. 37–71, 2010.

[16] C. S. Pasareanu, Q.-S. Phan, and P. Malacaria, "Multi-run side-channel analysis using symbolic execution and max-smt," in *Proceedings of the IEEE Computer Security Foundations (CSF)*, 2016, pp. 387–400.

[17] Q.-S. Phan, L. Bang, C. S. Pasareanu, P. Malacaria, and T. Bultan, "Synthesis of adaptive side-channel attacks," in *Computer Security Foundations Symposium (CSF), 2017 IEEE 30th*, 2017, pp. 328–342.

[18] T. Antonopoulos, P. Gazzillo, M. Hicks, E. Koskinen, T. Terauchi, and S. Wei, "Decomposition instead of self-composition for proving the absence of timing channels," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017, pp. 362–375.

[19] J. Chen, Y. Feng, and I. Dillig, "Precise detection of side-channel vulnerabilities using quantitative cartesian hoare logic," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 875–890.

[20] D. Page, "Partitioned cache architecture as a side-channel defense mechanism," in *Cryptology ePrint Archive, Report 2005/280*, 2005.

[21] J. Bonneau and I. Mironov, "Cache-collision timing attacks against AES," in *Cryptographic Hardware and Embedded Systems - CHES 2006*, ser. Lecture Notes in Computer Science, L. Goubin and M. Matsui, Eds. Springer Berlin Heidelberg, 2006, vol. 4249, pp. 201–215.

[22] A. Bogdanov, T. Eisenbarth, C. Paar, and M. Wienecke, "Differential cache-collision timing attacks on AES with applications to embedded cpus," in *Topics in Cryptology–CT-RSA 2010*, ser. Lecture Notes in Computer Science, J. Pieprzyk, Ed., 2010, vol. 5985, pp. 235–251.

[23] J. Agat, "Transforming out timing leaks," in *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, Jan. 2000, pp. 40–53.

[24] D. Hedin and D. Sands, "Timing aware information flow security for a JavaCard-like bytecode," *Electronic Notes in Theoretical Computer Science*, vol. 141, no. 1, pp. 163–182, 2005.

[25] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner, "The program counter security model: automatic detection and removal of control-flow side channel attacks," in *Proceedings of the 8^{th} International Conference on Information Security and Cryptology*, 2006, pp. 156–168.

[26] B. Coppens, I. Verbauwhede, K. D. Bosschere, and B. D. Sutter, "Practical mitigations for timing-based side-channel attacks on modern x86 processors," in *Proceedings of the 30^{th} IEEE Symposium on Security and Privacy (S&P)*, 2009, pp. 45–60.

[27] D. Zhang, A. Askarov, and A. C. Myers, "Language-based control and mitigation of timing channels," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012, pp. 99–110.

[28] H. Raj, R. Nathuji, A. Singh, and P. England, "Resource management for isolation enhanced cloud services," in *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*, 2009, pp. 77–84.

[29] J. Shi, X. Song, H. Chen, and B. Zang, "Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring," in *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops*, 2011, pp. 194–199.

[30] Ú. Erlingsson and M. Abadi, "Operating system protection against side-channel attacks that exploit memory latency," Microsoft Research, Tech. Rep. MSR-TR-2007-117, August 2007.

[31] T. Kim, M. Peinado, and G. Mainar-Ruiz, "Stealthmem: System-level protection against cache-based side channel attacks in the cloud," in *Proceedings of the 21st USENIX Conference on Security Symposium*, 2012, pp. 189–204.

[32] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, 2007, pp. 494–505.

[33] X. Li, V. Kashyap, J. K. Oberg, M. Tiwari, V. R. Rajarathinam, R. Kastner, T. Sherwood, B. Hardekopf, and F. T. Chong, "Sapper: A language for hardware-level security policy enforcement," in *Proceedings of the 19th Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014, pp. 97–112.

[34] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A hardware design language for timing-sensitive information-flow security," in *Proceedings of the 20th Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015, pp. 503–516.

[35] Y. Zhang and M. K. Reiter, "Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2013, pp. 827–838.

[36] Z. Wang and R. B. Lee, "A novel cache architecture with enhanced performance and security," in *Proceedings of the 41st Annual IEEE/ACM Int'l Symposium on Microarchitecture (MICRO)*, 2008, pp. 83–93.

[37] F. Liu and R. B. Lee, "Random fill cache architecture," in *Proceedings of the 47th Annual IEEE/ACM Int'l Symposium on Microarchitecture (MICRO)*, 2014, pp. 203–215.

[38] J. A. Goguen and J. Meseguer, "Security policies and security models," in *IEEE Symposium on Security and Privacy (S&P)*, Apr. 1982, pp. 11–20.

[39] M. Gordon and H. Collavizza, "Forward with Hoare," *Reflections on the Work of C.A.R. Hoare*, pp. 101–121, 2010.

[40] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Commun. ACM*, vol. 18, no. 8, pp. 453–457, Aug. 1975.

[41] D. Babic and A. J. Hu, "Calysto: scalable and precise extended static checking," in *Proceedings of the 30th international conference on Software engineering*, 2008, pp. 211–220.

[42] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, 2018.

[43] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," *SIGPLAN Not.*, vol. 47, no. 6, pp. 193–204, Jun. 2012.

[44] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, "Enhancing symbolic execution with veritesting," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 1083–1094.

[45] E. Brickell, G. Graunke, M. Neve, and J.-P. Seifert, "Software mitigations to hedge AES against cache-based software side channel vulnerabilities." *IACR Cryptology ePrint Archive*, vol. 2006, p. 52, 2006.

[46] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa, "Strong and efficient cache side-channel protection using hardware transactional memory," in *Proceedings of the 26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 217–233.

[47] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization*, ser. CGO '04, 2004.

[48] L. D. Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Proceedings of the Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[49] A. J. Menezes, S. A. Vanstone, and P. C. V. Oorschot, *Handbook of Applied Cryptography*, 1st ed. Boca Raton, FL, USA: CRC Press, Inc., 1996.