Adversarial Symbolic Execution for Detecting Concurrency-Related Cache Timing Leaks

Shengjian Guo
Virginia Tech
Blacksburg, VA, USA

Meng Wu
Virginia Tech
Blacksburg, VA, USA

Chao Wang
University of Southern California
Los Angeles, CA, USA

ABSTRACT

The timing characteristics of cache, a high-speed storage between the fast CPU and the slow memory, may reveal sensitive information of a program, thus allowing an adversary to conduct side-channel attacks. Existing methods for detecting timing leaks either ignore cache all together or focus only on passive leaks generated by the program itself, without considering leaks that are made possible by concurrently running some other threads. In this work, we show that *timing-leak-freedom* is not a compositional property: a program that is not leaky when running alone may become leaky when interleaved with other threads. Thus, we develop a new method, named *adversarial symbolic execution*, to detect such leaks. It systematically explores both the feasible program paths and their interleavings while modeling the cache, and leverages an SMT solver to decide if there are timing leaks. We have implemented our method in LLVM and evaluated it on a set of real-world ciphers with 14,455 lines of C code in total. Our experiments demonstrate both the efficiency of our method and its effectiveness in detecting side-channel leaks.

CCS CONCEPTS

• **Security and privacy → Cryptanalysis and other attacks**; • **Software and its engineering → Software verification and validation**;

KEYWORDS

Side-channel attack, concurrency, cache, timing, symbolic execution

ACM Reference Format:
Shengjian Guo, Meng Wu, and Chao Wang. 2018. Adversarial Symbolic Execution for Detecting Concurrency-Related Cache Timing Leaks. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3236024.3236028

1 INTRODUCTION

Side-channel attacks are security attacks where an adversary exploits the dependency between sensitive data and non-functional properties of a program such as the execution time [28, 43], power consumption [44, 51], heat, sound [37], and electromagnetic radiation [36, 57]. For timing side channels, in particular, there are two main sources of leaks: variances in the number of executed instructions and variances in the cache behavior. Instruction-induced leaks are caused by differences in the number and type of instructions executed along different paths: unless the differences are independent of the sensitive data, they may be exploited by an adversary. Cache-induced leaks are caused by differences in the number of cache hits and misses along different paths.

ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5573-5/18/11...$15.00
https://doi.org/10.1145/3236024.3236028

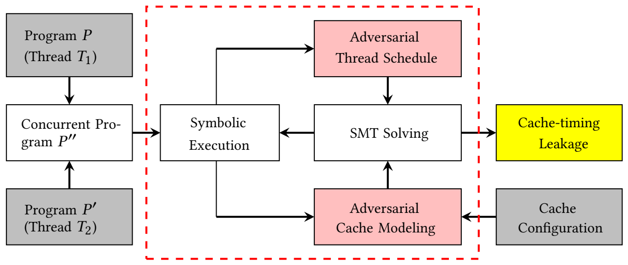
Program P (Thread T_1)
Concurrent Program P''
Program P' (Thread T_2)
Symbolic Execution
Adversarial Thread Schedule
SMT Solving
Adversarial Cache Modeling
Cache-timing Leakage
Cache Configuration

Figure 1: Flow of our cache timing leak detector SymSC.

Existing methods for detecting timing leaks or proving their absence often ignore the cache all together while focusing on instruction-induced leaks. For example, Chen et al. [23] used Cartesian Hoare Logic [58] to prove the timing leak of a program is within a bound; Antonopoulos et al. [8] used a similar technique that partitions the set of program paths in a way that, if individual partitions are proved to be timing attack resilient, the entire program is also timing attack resilient. Unfortunately, these methods ignore the cache-timing characteristics. Even for techniques that consider the cache [12, 21, 25, 30, 46, 61], their focus has been on leaks manifested by the program itself when running alone, without considering the cases when it is executed concurrently with some other (benign or adversarial) threads.

In this work, we show *side-channel leak-freedom*, as a security property, is not compositional. That is, a leak-free program when running alone may still be leaky when it is interleaved with other threads, provided that they share the memory subsystem. This is the case even if all paths in the program have the same number and type of instructions and thus do not have *instruction-induced* timing leaks at all. Unfortunately, no existing method or tool is capable of detecting such timing leaks.

We propose a new method, named *adversarial symbolic execution*, to detect such concurrency-related timing leaks. Specifically, given a program where one thread conducts a security-critical computation, e.g., by calling functions in a cryptographic library, and another thread is (either accidentally or intentionally) adversarial, our method systematically explores both paths in these threads and their interleavings. The exploration is symbolic in that it covers feasible paths under all input values. During the symbolic execution, we aim to analyze the cache behavior related to sensitive data to detect timing leaks caused by the interleaving.

Figure 1 shows the flow of our leak detector named SymSC, which takes the victim thread P, a potentially adversarial thread P', and the cache configuration as input. If P' is not given, SymSC creates it automatically. While symbolically executing the program, SymSC explores all thread paths and searches for an adversarial interleaving of these paths that exposes divergent cache behaviors in P. There are two main technical challenges. The first one is associated with systematic exploration of the interleaved executions of a concurrent program so as not to miss any adversarial interleaving. The second one is associated with modeling the cache accurately while reducing the computational cost.

To address the first challenge, we developed a new algorithm for adversarially exploring the interleaved executions while mitigating the *path and interleaving explosions*. Specifically, cache timing behavior constraints, which are constructed *on the fly* during symbolic execution, are leveraged to prune interleavings redundant for detecting leaks and thus speed up the exploration.

To address the second challenge, we developed a technique for modeling the cache behavior of a program based on the cache's type and configuration, as well as optimizations of the subsequent constraint solving to reduce overhead. For each concurrent execution (an interleaving of the threads) denoted $\pi = (in, sch)$, where in is the sensitive data input and sch is the interleaving schedule, we construct a logical constraint $\tau_t(in, sch)$ for every potentially adversarial memory access t, to indicate when it leads to a cache hit. Then, we seek two distinct values of the data input, in and in', for which the cache behaves differently: $\tau_t(in, sch) \neq \tau_t(in', sch)$, meaning one of them is a hit but the other is a miss, and they are due to differences in the sensitive data input.

We have implemented our method in a software tool based on LLVM and the KLEE symbolic virtual machine [20], and evaluated it on twenty benchmark programs. These security-critical programs are ciphers taken from cryptographic libraries in the public domain; they have 14,455 lines of C code in total. Since these programs are crafted by domain experts, they do not have obvious timing leaks when running alone, such as unbalanced branching statements or variances in lookup-table accesses. However, our experiments of applying SymSC show that they may still have timing leaks when being executed concurrently with other threads.

To summarize, we make the following contributions:

- We propose an *adversarial symbolic execution* method capable of detecting cache timing leaks in a security-critical program when it runs concurrently with other threads.
- We implement and evaluate our method on real-world cipher programs to demonstrate its effectiveness in detecting concurrency-related timing leaks.

In the remainder of this paper, we first motivate our work using several examples in Section 2 and then provide the technical background in Section 3. We present our detailed algorithms in Sections 4 and 5, which are followed by domain-specific optimizations in Section 6 to reduce the computational overhead. We present our experimental results in Section 7 and review the related work in Section 8. Finally, we give our conclusions in Section 9.

2 MOTIVATION

In this section, we use examples to explain the difference between self-leaking and concurrency-induced leaking.

2.1 A Self-leaking Program and the Repair

Figure 2(a) shows a program whose execution time is dependent of the sensitive variable k. It is a revised version of the running example used in [22], for which the authors proposed the leak-free version shown in Figure 2(b). The two programs have the same set of instructions but differ in where the highlighted load instruction is located: line 5 in P and line 9 in P_r.

```
/*  k is sensitive input */
1:  char p[256];
2:  unsigned char k;
3:  char q[256];
4:
5:  load reg1, p[k]
6:  if (k <= 127)
7:    load reg2, q[255-k]
8:  else
9:    load reg2, q[k-128]
10: add reg1, reg2
11: store reg1, p[k]
-----------------------------
k=0 : <miss, miss, miss>
1≤k≤255: <miss, miss, hit>
```

(a) The leaky program P

```
/*  k is sensitive input */
1:  char p[256];
2:  unsigned char k;
3:  char q[256];
4:
5:  if (k <= 127)
6:    load reg2, q[255-k]
7:  else
8:    load reg2, q[k-128]
9:  load reg1, p[k]
10: add reg1, reg2
11: store reg1, p[k]
-----------------------------

0≤k≤255: <miss, miss, hit>
```

(b) The leak-free version P_r

Figure 2: A program with cache-timing leak (cf. [22]).

Consider executing the two programs under a 512-byte direct-mapped cache with one byte per cache line, as shown in Figure 3. The choice of one-byte-per-cache-line — same as in [22] — is meant to simplify analysis without loss of generality. Specifically, the 256-byte array p is associated with the first 256 cache lines, while variable k is associated with the 257-th cache line. Due to the finite cache size, q[255] has to share the cache line with p[0].

There are two program paths in P, each with three memory accesses: load (line 5), load (line 7 or line 9), and store (line 11). However, depending on the value of k, these three memory accesses may exhibit different cache behaviors, thus causing data-dependent timing variance.

Assume that k's value is 0, executing P means taking the then branch and accessing p[0], q[255], and p[0]. The first access to p[0] is a cold miss since the cache is empty at the moment. The access to q[255] is a conflict miss because the cache line (shared by q[255] and p[0]) is occupied by p[0]; as a result q[255] evicts p[0]. The next access to p[0] is also a conflict miss since the cache line is occupied by q[255]. All in all, the cache behavior is <miss,miss,miss> for k=0.

This sequence is also unique in that all other values of k would produce <miss,miss,hit> as shown at the bottom of Figure 2(a). This means P, when running alone, leaks information about k. For example, upon observing the delay caused by <miss,miss,miss> via monitoring, an adversary may infer that k's value is 0.

Program P_r is a repaired version [22] where the load is moved from line 5 to line 9 as in Figure 2(b). Thus, the load accessing p[k] at line 9 always generates a cold miss (0<k≤255) or a conflict miss (k=0). Consequently, the store at line 11 is always a hit. Thus, for all values of k, the cache behavior remains <miss,miss,hit> – no information of k is leaked.

2.2 New Leak Induced by Concurrency

Although P_r is a valid repair when the program is executed sequentially, the situation changes when it is executed concurrently with other threads. Specifically, if we use one thread (T_1) to execute P_r while allowing a second thread (T_2) to run concurrently, P_r may exhibit new timing leaks.

Figure 4 shows a two-threaded program comprising T_1 and an adversarial T_2 that accesses a new variable tmp. Assume tmp is

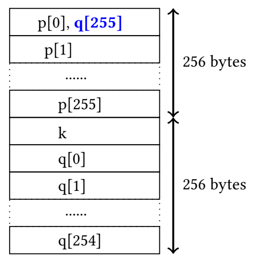

Figure 3: The direct-mapped cache layout (cf. [22]).

```
/*  [Thread T1] */
1:  char p[256];
2:  unsigned char k;
3:  char q[256];
4:
5:  if (k <= 127)
6:    load reg2, q[255-k]
7:  else
8:    load reg2, q[k-128]
9:  load reg1, p[k]
10: add reg1, reg2
11: store reg1, p[k]
```

```
/*  [Thread T2] */
12: unsigned char tmp;
13: load reg3, tmp
14: ...
```

Figure 4: Concurrent program with side-channel leak.

mapped to the same cache line as `p[1]`. Then, it is possible for T_2 to cause T_1 to leak information of its secret data. There are various ways of mapping `tmp` to the same cache line as `p[1]`, e.g., by dynamically allocating the memory used by `tmp` or invoking a recursive (or non-recursive) function within which `tmp` is defined as a stack variable.

Table 1 shows the six interleavings of threads T_1 and T_2. The left half of this table contains three interleavings where T_1 took the `then` branch of the if-statement, while the right half contains three interleavings where T_1 took the `else` branch. In each case, the four columns show the ID, the execution order, the cache sequence of thread T_1, and the value range of `k`. For example, in `6-9-11-13`, the `store` at line 11 is a cache hit because its immediate predecessor (line 9) already loads `p[k]` into the cache. Since the last `load` at line 13 comes from thread T_2, the cache behavior sequence of T_1 is `<miss,miss,hit>`, denoted `<m,m,h>` for brevity.

Table 1: Interleavings and thread T_1's cache sequences.

ID	Interleaving	Cache-seq	k	ID	Interleaving	Cache-seq	k
1	`6-13-9-11`	<m,m,h>	[0,127]	4	`8-13-9-11`	<m,m,h>	(127,255]
2	`6-9-11-13`	<m,m,h>	[0,127]	5	`8-11-9-13`	<m,m,h>	(127,255]
3	`6-9-13-11`	<m,m,h>	[0,1)∪(1,127]	6	`8-9-13-11`	<m,m,h>	(127,255]
		<m,m,m>	1				

Although context switches between the threads T_1 and T_2 may occur at any time in practice, for the purpose of analyzing cache timing leaks, we assume they occur only before the `load` and `store` statements. Furthermore, we only focus on these memory accesses when they are mapped to the same cache line, e.g., between the `load` in T_2 and statements that access `p[k]` in T_1.

We use Figure 5 to show details of `6-9-13-11`. The blue and orange rectangles represent the `load` and `store` accesses, respectively, and the red dashed poly-line shows their execution order. The first three `load` operations all cause cache misses, whereas the last `store` could be a cache hit if (`k!=1`) and a cache miss if

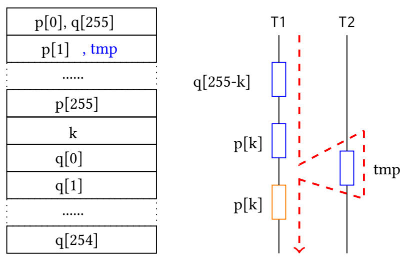

Figure 5: Interleaving `6-9-13-11` and the cache layout.

(`k=1`). When (`k=1`), the four memory accesses would be `q[254]`, `p[1]`, `tmp`, and `p[1]`. The first two trigger cold misses. The third one (`tmp`) triggers a conflict miss as the cache line was occupied by `p[1]`. Evicting this cache line would then lead to another conflict miss for the subsequent `store` to `p[1]`.

The examples presented so far show that, even for a timing-leak-free program (T_1), running it concurrently with another thread (T_2) may cause it to exhibit new timing leaks. This is the case even if the two threads (T_1 and T_2) are logically independent of each other. In other words, they do not need to share variables or communicate through messages; they can affect each other's timing behaviors by sharing the same cache system.

2.3 Adversarial Symbolic Execution

The goal of developing a new symbolic execution method is to detect such timing leaks. More specifically, we are concerned with two application scenarios for SymSC, depending on whether the adversarial thread (T_2) exists in the given program or not.

Case 1. Thread T_2 is given, together with fixed addresses of the memory region accessed by T_2. In this case, T_2 is an integral part of the concurrent system that also contains the security-critical computation in T_1. Since the only source of nondeterminism is thread interleaving, our tool aims to check if the concurrent system itself has timing leaks.

Case 2. Thread T_2 is not given, but created by our tool, and thus the addresses of the memory region accessed by T_2 are assumed to be symbolic. This is when, inside the cache layout of Figure 5, the address of `tmp` would be made symbolic, thus allowing it to be mapped to any cache line (as opposed to be fixed to the 2nd line). There are now two sources of non-determinism: thread interleaving and memory layout. Our tool explores both to check if T_1 may leak information due to interference from T_2.

In the second case, when T_2 executes a memory load instruction t, for example, the symbolic address *addr* may be mapped to any cache line. The purpose of having such aggressive adversarial addressing is to allow SymSC to conduct a (predictive) *what-if* analysis: it searches all potential memory layouts to check if there exists one that allows T_2 to cause a timing leak.

3 THE THREAT MODEL

We now review the technical background and present the threat model, which defines what an adversary can or cannot do.

3.1 Cache and the Timing Side Channels

The execution time of a program depends on the CPU cycles taken to execute the instructions and the time needed to access memory. The first component is easy to compute but also less important in practice, because security-critical applications often execute the same set of instructions regardless of values of their sensitive variables [65]. In contrast, leaks are more likely to occur in the second component: the time taken to access memory. Compared to the time needed to execute an instruction, which may be 1-3 clock cycles, the time taken to access memory, during a cache miss, may be tens or even hundreds of clock cycles.

There are different types of cache based on the size, associativity and replacement policy. For ease of comprehension, we use direct-mapped cache with LRU policy in this paper, but other cache types may be handled similarly. Indeed, during our experiments, both direct-mapped cache and 4-way set-associative cache were evaluated and they led to similar analysis results.

We assume the security-critical program P implements a function $c \leftarrow f(k, x)$, where k is the sensitive input (secret), x is the non-sensitive input (public), and c is the output. In block ciphers, for example, k would be the cryptographic key, x would be the plaintext, c would be the ciphertext, and f would be the encryption or decryption procedure.

Let the execution time of P be $\tau_P(k, x)$. Since there may be multiple paths inside P, when referring to a particular path $p \in P$, we use $\tau_p(k, x)$. But if there is no ambiguity, we may omit the detail and simply use $\tau(k, x)$. We say P is leak-free if $\tau(k, x)$ remains the same for all input values. That is,

$$\forall x, k_1, k_2 \,.\, \tau(k_1, x) = \tau(k_2, x)$$

Here k_1 and k_2 are two arbitrary values of k. Since in practice, decision procedures (e.g., SMT solvers) are designed for checking satisfiability, instead of proving the validity of a formula, we try to falsify it by checking the formula below:

$$\exists x, k_1, k_2 \,.\, \tau(k_1, x) \neq \tau(k_2, x)$$

Here, we search for two values of k that can lead to differences.

If the set of instructions executed by P remains the same, we only need to check whether $\tau(k_1, x)$ and $\tau(k_2, x)$ have the same number of cache hits and misses. Furthermore, in our threat model where the attacker can only observe (passively) the execution time of P, but not control or observe x, we can reduce the computational cost by fixing a value $\overline{x}$ of x arbitrarily and then checking if $\tau(k_1)$ and $\tau(k_2)$ have the same number of cache hits and misses.

3.2 Example of an Attack

Now, we show a concrete example of exploiting cache timing leaks in concurrent systems. The goal is to illustrate what an adversary may be able to achieve in practice.

Figure 6 shows a two-threaded program, its cache mapping, and the thread-local control flows. Initially, T2 allocates a memory area (buf) whose size matches the input. Although the input size may be arbitrary, here, we assume it is an integral multiple of 64, e.g., 1024 bytes (INPUT_SIZE=1024). In the *while*-loop (line 14) T2 reads 64 bytes from input every time to fill buf. Thread T1 tracks the progress (idx) of T2 (line 4) and repeatedly retrieves 64-byte data from buf to the array out (line 5). The encryption on out involves the S-Box array S and a given key (lines 6-7). Once the data is encrypted, T1 sends it out (line 8). When T1 finds that buf runs out of data, it sleeps for 50ms (line 10).

First, we explain why the program has a timing leak. We use a 32KB direct-mapped cache here and set each cache line to 64 bytes. The S-Box array S hence maps to 4 cache lines and the buf array maps to 16 cache lines. For brevity, we only focus on the important

```
uint8_t *buf = 0; uint32_t size = INPUT_SIZE; uint32_t idx = 0;
```

```
/*  [Thread T1] */
1:  uint8_t S[256] = {0x4b,...};
2:  uint8_t out[64] = {0};
3:  for(int i=0; i<size; )
4:    if (i < idx)
5:      memcpy(out,buf+i,64);
6:      for (int j=0;j<64;j++,i++)
7:        out[j] &= S[key[j]];
8:      write(out, ...);
9:    else
10:     sleep(50);
```

```
/*  [Thread T2] */
11: ......
12: buf=(uint8_t *)malloc(size);
13: while(idx<size)
14:   memcpy(buf+idx,read(...),64);
15:   idx+=64;
16: ......
```

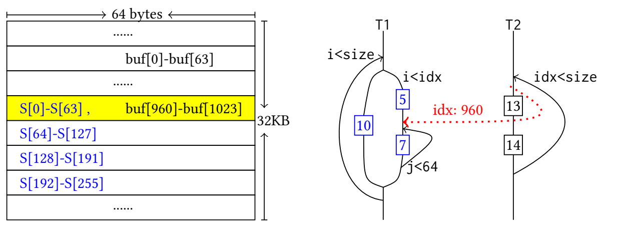

Figure 6: A two-threaded encryption program.

arrays (S and buf) while assuming other variables do not affect the cache mapping. Furthermore, we assume S and buf share one cache line as highlighted in Figure 6.

The graph in Figure 6 shows an interleaving of T1 and T2, where the dotted red arrow represents a context switch after T2 executes the memcpy statement (line 14) while T1 just reaches the *for*-loop at line 6. The text above the arrow means idx's value is 960 at the moment, indicating thread T2 has just accessed the last 64 bytes of buf at line 14.

After the context switch, T1 enters the *for*-loop (line 6) and reads S[key[j]] at line 7. Note that the offset to S's base address depends on key[j], thus different keys may make thread T1 access different items of S. We pick two 64-byte keys **k1** and **k2** which differ in the first eight bits: 10000000 for **k1** and 00000000 for **k2**. Using **k1**, thread T1 first reads key[0] and S[128]. The access to S[128] would lead to a cache hit if i is greater than 63. This is because after the *for*-loop (lines 6-7) finishes once (i=64), S[128] is already mapped to cache and no further accesses evict it.

In contrast, with **k2**, thread T1 loads S[0] which maps to the cache line shared with buf[960-1023]. Recall that, before the context switch, T2 just accessed the area starting from buf+idx (buf[960]). Consequently T1's access to S[0] causes a conflict miss because the shared cache line was occupied by buf. Thus, we find a leak: two keys (**k1** and **k2**) leading to divergent cache behaviors at a program location due to thread interleaving.

Next, we discuss how this leak may be exploited. The leak is due to the sharing of cache between S and buf, which is crucial to our threat model. In this program, S has a fixed size while buf is dynamically allocated at run time based on the input data. Furthermore, INPUT_SIZE is a variable affected by the external input. Although the actual input size cannot be arbitrarily large in practice, for this exploit to work, it only needs to be larger than the total cache size, which is 32KB.

Thus, the attacker could mutate the input to alter the buffer size, hence affecting the memory layout. Furthermore, real applications sometimes use relatively large fixed buffers. For example, in OpenSSH [5], the scp program has a 16KB buffer for COPY_BUFLEN and the sftp program has a 32KB buffer for DEFAULT_COPY_BUFFER.

```
/*  cipher-ctr-mt.c */
   ...
504:for(i=0;i<CIPHER_THREADS;i++){
    ......
507:  pthread_create(...,
               thread_loop,...);
    ......
509:}
```

```
/*  cipher-ctr-mt.c */
238:static void*
239:thread_loop(void *x) {
      ......
326:  for(i=0;i<KQLEN;i++) {
327:    AES_encrypt(q->ctr,
            q->keys[i],&key);
      ......
```

Figure 7: Concurrency-related code in HPN-SSH [2].

Moreover, OpenSSH's `SSHBUF_SIZE_MAX` buffer for a socket channel is as large as 256MB. These large buffers allow room for attackers to construct the desired cache layout.

We have found a similar scenario in the open-source implementation of HPN-SSH [2], which is an enhancement of OpenSSH [5] by leveraging multi-threading to accelerate the data encryption. Figure 7 shows the code snippet directly taken from the HPN-SSH [2] repository: On the left-hand side are threads created to run the `thread_loop` function, shown on the right-hand side, which repeatedly calls `AES_encrypt` to encrypt data given by the user (line 327). By controlling the size and content of the data, as well as the number of threads, a malicious user is able to affect both the memory layout and the thread interleaving.

In our experimental evaluation (Section 7), we will show that the AES subroutine from OpenSSL indeed has cache timing leaks, which may subject HPN-SSH to attack scenarios similar to the one illustrated in Figure 6.

4 ADVERSARIAL SYMBOLIC EXECUTION

We first present the baseline algorithm for concurrent programs. Then, we enhance it to search for cache timing leaks.

4.1 The Baseline Algorithm

Following Guo et al. [40], we assume the entire program consists of a finite set $\{T_1, \ldots, T_n\}$ of threads where each thread T_i $(1 \leq i \leq n)$ is a sequential program. Without loss of generality, we assume T_1 is critical and any of $T_2, \ldots, T_n$ may be adversarial. Let st be an instruction in a thread. Let *event* $e = \langle tid, l, st, l' \rangle$ be an instance of st, where l and l' are thread-local locations before and after executing st. A *global location* is a tuple $s = \langle l_1, \ldots, l_n \rangle$ where each l_i is a location in T_i. Depending on the type of st, an event may have one of the following types:

- α-event, which is an assignment $v_l := exp_l$ where v_l is a local variable and exp_l is an expression in local variables.
- β-event, which is a local branch denoted `assume`$(cond_l)$ where the condition $cond_l$ is expressed in local variables.
- γ-event, which is a `load` from global memory of the form $v_l := v_g$, a `store` to global memory of the form $v_g := exp_l$, or a thread synchronization operation.

For an `if(c)-else` statement, we use `assume`(c) to denote the then-branch, and `assume`$(\neg c)$ to denote the else-branch. Since c is expressed in local variables or local copies of global variables, β-events are local branching points whereas γ-events are thread interleaving points. Both β- and γ-events contribute to the state-space explosion problem. In contrast, α-events are local to their own threads. Details on handling of language features such as pointers and function calls are omitted, since they are orthogonal issues addressed by existing symbolic execution tools [20, 26].

Algorithm 1 shows the baseline symbolic execution procedure that follows the prior work [14, 26, 40] except that, for the purpose of detecting timing leaks, it considers two events as *dependent* also when they are mapped to the same cache line. Here, an execution is characterized by $\pi = (in, sch)$ where $in = \{k, x\}$ is the data input and sch is the thread schedule, corresponding to a total order of events $e_1 \ldots e_n$, and *Stack* is a container for symbolic states. Each $s \in Stack$ is a tuple $\langle \mathcal{M}, pcon, branch, enabled, crt \rangle$, where $\mathcal{M}$ is the symbolic memory, *pcon* is the path condition, *branch* is the set of branching (β) events, *enabled* is the set of thread interleaving (γ) events, and *crt* is the event chosen to execute at s.

Algorithm 1: Baseline Symbolic Execution Procedure.

```
   Initially: State stack Stack = ∅;
   Start SymSC(s_0) with the initial symbolic state s_0.
1  SymSC(State s)
2  begin
3      Stack.push(s);
4      if s is a thread-local branching point then
5          for t ∈ s.branch and s.pcon ∧ t is satisfiable do
6              SymSC(NextSymbolicState(s, t));   // β event
7          end
8      else if s is a thread interleaving point then
9          for t ∈ s.enabled do
10             SymSC(NextSymbolicState(s, t));   // γ event (enhanced)
11         end
12     else if s is other sequential computation then
13         SymSC(NextSymbolicState(s, s.crt));       // α event
14     else
15         terminate at s;
16     end
17     Stack.pop();
18 end
19 NextSymbolicState(State s, Event t)
20 begin
21     s.crt ← t;
22     s′ ← Execute the event t in the state s;
23     return s′;
24 end
```

At the beginning, the stack is empty and the entry is the initial state s_0. Then, depending on the type of the state s, we may execute a local branch (line 4), perform a context switch (line 8), or execute a sequential computation (line 12). In all cases, SymSC is invoked again on the new state.

Sub-procedure *NextSymbolicState* takes the current state s and to-be-executed event t as input, and returns the new state s' as output: s' is the result of executing t at s. We omit details since they are consistent with existing symbolic execution methods [39–41, 66, 67].

Also note that, in the prior work, symbolic execution would allow interleavings between global (γ) events only if they have *data conflicts*, i.e., they are from different threads, accessing the same memory location, and at least one of them is a write. This is because only such accesses may lead to different states if they are executed in different orders. However, in our case, whether these events are mapped to the same cache line also matters.

4.2 Enhanced Algorithm

We enhance the baseline algorithm to arrive at Algorithm 2, where the main difference is in the interleaving points. Upon entering the *for*-loop at line 5, we first check if an enabled event t may lead to a timing leak by invoking *DivergentCacheBehavior(s,t)*. Details of the subroutine will be presented in Section 5, but at the high level, it constructs a cache behavior constraint τ_t and then searches for two values, $\overline{k_1}$ and $\overline{k_2}$, such that $\tau_t(\overline{k_1}) \neq \tau_t(\overline{k_2})$.

Since detecting such divergent behaviors is computationally expensive, prior to invoking the subroutine, we make sure that event t indeed may be involved in an adversarial interleaving. This is determined by *AdversarialAccess(s,t)* which checks if (1) t comes from the critical thread T_1 and (2) there exists a previously executed event $t' = s'.crt$ where $s' \in Stack$ and the two events (t and t') are mapped to the same cache line.

Algorithm 2: Symbolic Execution in SymSC

```
Initially: State stack $Stack = \emptyset$;
Start SymSC($s_0$) with the initial symbolic state $s_0$.
SymSC(State $s$)
begin
    ......
    else if $s$ is a global-memory access point then
        for $t \in s.enabled$ do
            if DivergentCacheBehavior($s$, $t$) then
                generate test case;
                terminate at $s$;
            else
                SymSC(NextSymbolicState($s$, $t$));
            end
        end
    ......
end

DivergentCacheBehavior(State $s$, Event $t$)
begin
    if AdversarialAccess($s$, $t$) then
        $\tau_t \leftarrow$ compute $t$'s cache hit constraint;
        if $\exists k, k'$ such that $\tau_t(k) \neq \tau_t(k')$ then
            return true;
    end
    return false;
end

AdversarialAccess(State $s$, Event $t$)
begin
    if $t$ is from the critical thread then
        let $s' \in Stack$ and $t' = s'.crt$;
        if $\exists t'$ . $t$ and $t'$ may map to same cache line then
            return true;
    return false;
end
```

For our running example in Figure 4, in particular, Algorithm 2 would explore the first three interleavings in Table 1 before detecting the leak. The process is partially illustrated by Figure 8, where events t_1:`load q[255-k]`, t_2:`load p[k]` and t_3:`store p[k]` belong to thread T_1 whereas t_4:`load tmp` belongs to thread T_2.

Assume T_1 executes t_1 to reach t_2 and T_2 is about to execute t_4: this corresponds to the figure on the left. At this moment, *s.enabled* = { t_2, t_4 }. If t_4 is executed before t_2, *AdversarialAccess(s,*t_2*)* would evaluate to true because t_2 comes from the critical thread and `p[k]` may be mapped to the same cache line as `tmp` accessed by t_4. However, there is no timing leak at t_2, because `p[k]` differs from t_1's access `q[255-k]`, meaning the cache behavior at t_2 remains the same for all values of k.

If t_2 were executed before t_4, we would have the second scenario in Figure 8. At this moment, *s.enabled* = { t_3, t_4 }. If t_4 is executed after t_3, the interleaving would be `6-9-11-13`, which does not have timing leaks either. But if t_4 were executed before t_3, we would have the third scenario in Figure 8, where *AdversarialAccess(s,* t_3*)* evaluates to true, $\tau_{t_3}(k)$ evaluates to *false* for (k=1) but to *true* for (k≠1)∧(k≤127), as shown in Table 2, leading to divergent cache behaviors in `6-9-13-11`.

5 ADVERSARIAL CACHE ANALYSIS

Our method for detecting divergent cache behaviors is as follows. First, it constructs the behavioral constraint for each memory access. Then, it solves the constraint to compute a pair of sensitive values that allow the constraint to return divergent results.

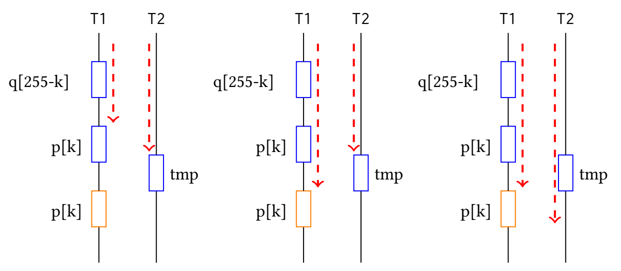

Figure 8: The three interleavings generated by SymSC.

5.1 Cache Modeling

Recall that the entire program contains T_1 and T_2, among other threads, where T_1 invokes the critical computation and T_2 is potentially adversarial. During symbolic execution, SymSC conducts context switches when `load` or `store` instructions may be mapped to the same cache line. Here, each interleaving p corresponds to a data input *in* and a thread schedule *sch*. The data input is divided further into $in = \{k, x\}$, where k is sensitive (secret) and x is non-sensitive (public). Whenever the value of x is immaterial, we assume $in = \{k\}$.

- An interleaving p is a sequence of memory accesses denoted $p(sch, in) = \{A_1, ..., A_n\}$ where *sch* represents the order of these accesses and *in* represents the data input.
- Each A_i, where $i \in [0, n]$, denotes a memory access.
- $pcon_i(k)$ is the path condition under which A_i is reached.

Thus, when $pcon_i(k)$ is true, meaning A_i is reachable, we check if A_i can lead to a cache hit:

- $\tau_i(k)$ denotes the condition under which A_i triggers a cache hit.
- $addr_i$ denotes the memory address accessed by A_i.
- $tag(addr)$ is a function that returns the unique *tag* of *addr*.
- $line(addr)$ is a function that returns the cache line of *addr*.

Thus, we define the *cache-hit condition* as follows:

$$\tau_i(k) \equiv \bigvee_{0 \le j < i} \Big(tag(addr_j) = tag(addr_i) \wedge \forall l \in [j+1, i-1] \,\big|\, line(addr_l) \neq line(addr_i) \Big) \quad (1)$$

For each memory access A_i, SymSC traverses the preceding memory accesses in the interleaving p to see if any such A_j may result in A_i being a cache hit. This is done by comparing the tag of $addr_i$ to that of $addr_j$—a hit is possible only when two tags are the same. Furthermore, any other memory access (A_l) between A_i and A_j must not evict the cache line occupied by A_j (and hence A_i). This means, for all $j < l < i$, we have $line(addr_l) \neq line(addr_i)$.

If A_i always causes a cache hit, or a miss, it cannot leak sensitive information because it implies $\forall k_1, k_2 \,.\, \tau_i(k_1) = \tau_i(k_2)$. In contrast, if $\tau_i(k)$ evaluates to *true* for some value of k but to *false* for a different value of k, then it is a leak.

5.2 Leakage Detection

After constructing $\tau_i(k)$, which is the *cache-hit condition* for a potentially adversarial memory access A_i, we instantiate the symbolic expression twice, first with a fresh variable k_1 and then with another fresh variable k_2. We use an off-the-shelf SMT solver to search for values of k_1 and k_2 that can lead to divergent behaviors.

Table 2: Cache-related information of interleaving p.

# line	i	$pcon_i$	$addr_i$	τ_i	cache
6	0	$k \le 127$	q[255-k]	$false$	miss
9	1	$k \le 127$	p[k]	$tag(p[k]) = tag(q[255-k])$	miss
13	2	$k \le 127$	tmp	$tag(tmp) = tag(p[k]) \lor (tag(tmp) = tag(q[255-k]) \land line(tmp) \neq line(p[k]))$	miss
11	3	$k \le 127$	p[k]	$tag(p[k]) = tag(tmp) \lor (tag(p[k]) = tag(p[k]) \land line(p[k]) \neq line(tmp))$	miss *or* hit

Precise Solution. The precise formulation is as follows:

$$\exists k_1, k_2 \, . \, (k_1 \neq k_2) \land \tau_i(k_1) \neq \tau_i(k_2) \tag{2}$$

We need to conduct this check at every memory access A_i, where $i \in [0, n]$, along the symbolic execution path p. If the above formula is satisfiable, the SMT solver will return values $\overline{k_1}$ and $\overline{k_2}$ of variables k_1 and k_2, respectively.

Two-Step Approximation. Since computing both values at the same time is expensive, in practice, we can take two steps:

- First, solve subformula $\exists k_1 \, . \, \tau_i(k_1)$ to compute a concrete value for k_1, denoted $\overline{k_1}$.
- Second, solve subformula $\exists k_2 \, . \, (\overline{k_1} \neq k_2) \land \tau_i(\overline{k_1}) \neq \tau_i(k_2)$ to compute a concrete value $\overline{k_2}$ for k_2.

Since the formula solved in each step is (almost twice) smaller, the solving time can be reduced significantly. Furthermore, a valid solution ($\overline{k_1}$ and $\overline{k_2}$) is guaranteed to be a valid solution for the original formula as well. However, in general, the two-step approach is an under-approximation: when it fails to find any solution, it is not a proof that no such solution exists.

To make the two-step approach precise, one would have to apply it repeatedly, each time with a different $\overline{k_1}$ computed in the first step, until all solutions of $\overline{k_1}$ is covered. Nevertheless, we shall show through experiments that, in practice, applying it once is often accurate enough to detect the actual leak.

5.3 The Running Example

We revisit the example in Figure 4 to show how our approach detects the leak. Recall that SymSC would generate the six interleavings shown in Table 1. For each interleaving, Table 2 shows the line number (#line) of every access A_i, path condition $pcon_i$, memory address $addr_i$, and the cache-hit constraint τ_i.

Inside the interleaving `6-9-13-11`, for instance, upon reaching the `load` of `q[255-k]` at line 6, the path condition would be ($k \le 127$). Since it is the first memory access, τ_0 must be *false* (cache miss). We will record this memory address for further analysis.

Next is the `load` of `p[k]` at line 9. SymSC builds τ_1 and checks its satisfiability. Since `p[k]` and the preceding `q[255-k]` correspond to different memory addresses, the *tag* comparison in τ_1 returns *false*, indicating a cache miss. The `load` at line 13 accesses `tmp`. Since `tmp` is different from any of the elements in arrays `p` and `q`, the *tag* comparisons in τ_2 return *false*, making A_2 a cache miss.

Similarly, τ_3 for the `store` at line 11 is shown in the last row of Table 2. It is worth mentioning that τ_3 only compares `p[k]` ($addr_3$) with `tmp` ($addr_2$) and `p[k]` ($addr_1$) but not `q[255-k]` ($addr_0$) because SymSC finds that, if the access to `tmp` does not evict the cache line used by its preceding access to `p[k]` ($addr_1$), the last `store` to `p[k]` ($addr_3$) must be a cache hit; SymSC stops here to avoid further (and unnecessary) analysis.

Differing from τ_0, τ_1 and τ_2, the constraint τ_3 depends on k due to the constraint $line(p[k]) \neq line(tmp)$. Specifically, $\tau_3(k)$ is *true* when ($k! = 1 \land k \le 127$) and is *false* when ($k = 1$).

In SymSC, two symbolic variables k_1, k_2 will be used to substitute k in the symbolic expression of $\tau_3(k)$, to form $\tau_3(k_1)$ and $\tau_3(k_2)$. Solving the satisfiability problem described by $\tau_3(k_1)$ *XOR* $\tau_3(k_2)$ would produce the assignment $\{k_1$=`0` and k_2=`1`$\}$, which makes $\tau_3(0)$ evaluate to *true* and $\tau_3(1)$ evaluate to *false*.

```
1  uint8_t SBOX1[64]={0x6f,0x3c,0x77,0xb7,0x2f,0x7b,0x5f,0xc6, ...};
2  uint8_t SBOX2[64]={0x3d,0x4c,0x5f,0xb6,0xd1,0xff,0x3e,0xed, ...};
3  void encrypt(uint8_t *block){
4     for (uint8_t i = 0; i < 64; i++){
5        block[i] |= SBOX1[block[i]];
6        block[i] ^= SBOX2[block[i]];
7     }
8  }
```

Figure 9: Example code for accessing S-Box lookup tables.

6 OPTIMIZATIONS

Symbolic execution, when applied directly to cipher programs, may have a high computational overhead because of the heavy use of arithmetic computations and look-up tables in these programs. In this section, we present techniques for reducing the overhead.

Toward this end, we have two insights. First, when conducting cache analysis, we are not concerned with the actual numerical computations inside the cipher unless they affect the addresses of memory accesses that may depend on sensitive data, e.g., indices of lookup tables such as S-Boxes. Second, for the purpose of detecting leaks, as opposed to proving their absence, we are free to under-approximate as long as it does not diminish the leak-detection capability of our analysis.

6.1 Domain-specific Reduction

By studying real-world cipher programs, we have found the computational overhead is often associated with symbolic indices of lookup tables such as the one shown in Figure 9.

Here, `block` points to a 8-byte storage area whose content depends on the cryptographic key; thus, the eight bytes are initialized with symbolic values. Accordingly, indices to the S-Box tables – `block[i]` at line 4 – are symbolic. However, not all memory accesses should be treated as symbolic. For example, the address of `block[i]` itself, and the address of local variables such as `i` should be treated as concrete values to reduce the cost of symbolic execution. Therefore, we conduct a static analysis of the interleaved execution trace p to identify the sequence of memory accesses that need to be kept symbolic while avoiding the symbolic expressions of other unnecessary memory addresses.

Also, a program may have multiple S-Box arrays, like `SBOX1` and `SBOX2` in Figure 9. Two successive accesses to `SBOX1` and `SBOX2` (at lines 5 and 6) cannot form a cache hit no matter what the lookup indices are. Therefore, we do not need to invoke the SMT solver to check the equivalence of these symbolic addresses. This can significantly cut down the constraint-solving time.

6.2 Layout-directed Reduction

Another reduction is guided by the memory layout. In LLVM, memory layout may be extracted from the compiler back-end after the code generation step. Recall that when analyzing a pair of potentially adversarial addresses, we need to compare them with all other addresses accessed between them to build the cache behavior constraint. More specifically, to check if A_2 is a cache hit because of A_1 along the execution $A_1 - B_1 -, ..., -B_n - A_2$, we need to check if any B_i $(1 \leq i \leq n)$ could evict the cache line used by A_1. Due to the large value of n and often complex symbolic expression of B_i, the constraint-solving time could be large.

Our approach in this case is to directly compare A_1 and A_2 while postponing the comparisons to B_i. This is based on the observation that, in practice, the cache line of A_1 can possibly be evicted by B_i only if the differences between their addresses is the multiple of the cache size (e.g., 64KB), which may not be possible in compact cipher programs. For example, in a 64KB direct-mapped cache, for B_1 to evict the 64-byte cache line of A_1, their address difference has to be 2^{16} = 64KB. In a 4-way set-associative cache, their address difference has to be 2^{14} = 16KB. Furthermore, in the event that A_2 has a cache hit due to A_1, we can add back the initially-omitted comparisons to $B_1, \ldots, B_n$ to undo the approximation.

7 EXPERIMENTS

We have implemented SymSC using the LLVM compiler [48] and *Cloud9* [18], which is a symbolic execution engine for multithreaded programs built upon KLEE [20]. We enhanced *Cloud9* in three aspects. First, we extended its support for multi-threading by allowing context switches prior to accessing global memory; the original *Cloud9* only allows context switches prior to executing a synchronization primitive (e.g., lock/unlock). Second, we made *Cloud9* fork new states to flip the execution order of two simultaneously enabled events when they may be mapped to the same cache line; the original *Cloud9* does not care about cache lines. Third, we made *Cloud9* record the address of each memory access along the execution, so it can incrementally build the cache-hit constraint. Based on these enhancements, we implemented our cache timing leak detector and optimized it for efficient constraint solving.

After compiling the C code of a program to LLVM bit-code, our SymSC tool executes it symbolically to generate interleavings according to Algorithm 2. The cache constraint at each memory access is expressed in standard KQuery expressions defined in KLEE [20]. By solving these constraints, we can obtain a concrete execution that showcases the leak, including a thread schedule, two input values $\overline{k_1}, \overline{k_2}$ and the adversarial memory address.

7.1 Benchmarks

We evaluated SymSC on a diverse set of open-source cipher programs. Specifically, the first group has five programs from a lightweight cryptographic system named *FELICS* [29], which was designed for resource-constrained devices. The second group has four programs from *Chronos* [27], a real-time Linux kernel. The third group has four programs from the GNU cryptographic library *Libgcrypt* [3], while the remaining programs are from the *LibTomCrypt* [4], the *OpenSSL* [6], and a recent publication [21]. They include multiple versions of several well-known algorithms such as `AES` [6, 27] and `DES` [3, 27], which are useful in evaluating the impact of cipher implementations on the performance of SymSC.

Table 3: Benchmark statistics: lines of C code (LOC) and LLVM code (LL), sensitive key-size (KS), and the memory accesses (MA).

Name	LOC	LL	KS	MA	Name	LOC	LL	KS	MA
`AES`[6]	1,429	4,384	24	771	`FCrypt`[27]	437	1,623	12	428
`AES`[27]	1,368	4,144	24	788	`KV_name`[21]	1,350	1,402	4	19
`Camellia`[4]	776	5,319	16	1,301	`LBlock`[29]	930	4,010	10	1,618
`CAST5`[4]	735	2,790	16	909	`Misty1`[1]	391	1,199	16	270
`CAST5`[27]	883	4,190	16	1,180	`Piccolo`[29]	301	1,034	12	350
`Chaskey`[29]	248	638	16	242	`PRESENT`[29]	194	272	10	94
`DES`[3]	596	2,166	8	963	`rfc2268` [3]	388	870	16	149
`DES`[27]	1,010	3,926	8	1,029	`Seed`[3]	607	3,535	16	979
`Kasumi`[1]	350	1224	16	259	`TWINE`[29]	256	562	10	229
`Khazad`[27]	838	463	16	123	`Twofish`[3]	1,048	4,510	16	1,180

Table 3 shows the statistics of these benchmark programs. The **LOC** and **LL** columns denote the lines of C code and the corresponding LLVM bit-code. The **KS** column shows the size of the sensitive input in bytes. The maximum number of memory accesses on program paths of each benchmark is shown in the **MA** column, which indicates the computational cost of the program.

Each program in the benchmark suite has from 194 to 1,429 lines of C code. In total, there are 14,455 lines of C code, which compile to 49,048 lines of LLVM bit-code. These numbers are considered substantial because ciphers are typically compact programs with highly computation-intensive operations, e.g., due to their use of loops and lookup-table based transformations. For example, the program named `PRESENT` has only 194 lines of C code but 8,233 memory accesses at run time.

We analyzed these benchmark programs using two types of caches: direct-mapped cache and four-way set-associative cache. The cache size is 64KB with each cache line consisting of 64 bytes; thus, there are 64KB/64B = 1024 cache lines, which are typical in mainstream computers today.

Our experiments were designed to answer two questions:

- Can SymSC detect cache-timing leaks exposed by concurrently running a program with other threads?
- Are the optimizations in Section 6 effective in reducing the cost of symbolic execution and constraint solving?

We conducted all experiments with Ubuntu 12.04 Linux running on a computer with a 3.40GHz CPU and 8GB RAM. For all evaluations we set the timeout threshold to 1,600 minutes.

7.2 Results Obtained with Fixed Addresses

Table 4 shows our results obtained using fixed addresses in the cache layout (Case 1 in Section 2.3). The first column shows the benchmark name. The next three columns show the result of computing the precise solution for our cache analysis problem. The last three columns show the result of running the simplified, two-step version, where the solution for $\exists k_1, k_2 \,.\, \tau(k_1) \neq \tau(k_2)$ is computed in two steps, by first computing a value of k_1 and then computing a value of k_2. In each method, we show the number of interleavings explored (#.Inter), the number of leaky memory accesses detected (#.Test), and the execution time in minutes (m). For the two-step approach, we also show the number of leakage points detected after the first step and after the second step.

Among these twenty programs, we detected leakage points in four: `ASE` from *OpenSSL* [6], `DES` from *Libgcrypt* [3], `FCrypt` from *Chronos* [27], and `Khazad` from *Chronos* [27]. We manually inspected these four programs in a way similar to what is described in Section 3.2, and confirmed that all these leakage points are realistic.

Table 4: Results of leak detection with *fixed* addresses: Is the program leaky w.r.t. the *given* thread?

Name	Precise			Two-Step		
	#.Inter	#.Test	Time (m)	#.Inter	#.Test step1 / step2	Time (m)
AES[6]	57	55	430.2	57	55 / 55	140.3
AES[27]	1	0	288.9	1	1 / 0	41.4
Camellia[4]	1	0	0.1	1	1 / 0	0.1
CAST5[4]	1	0	0.1	1	1 / 0	0.1
CAST5[27]	1	0	0.1	1	1 / 0	0.1
Chaskey[29]	1	0	0.1	1	1 / 0	0.1
DES[3]	16	15	7.8	16	16 / 15	3.5
DES[27]	1	0	0.1	1	1 / 0	0.1
FCrypt[27]	16	15	4.1	16	15 / 15	8.1
Kasumi[1]	1	0	0.1	1	1 / 0	0.2
Khazad[27]	25	23	206.5	25	23 / 23	83.0
KV_Name[21]	1406	0	0.5	1406	1406 / 0	0.4
LBlock[29]	1	0	0.1	1	1 / 0	0.1
Misty1[1]	1	0	0.1	1	1 / 0	0.1
Piccolo[29]	1	0	0.1	1	1 / 0	0.1
PRESENT[29]	1	0	0.1	1	1 / 0	0.1
rfc2268[3]	1	0	0.1	1	1 / 0	0.1
Seed[3]	1	0	0.1	1	1 / 0	0.1
TWINE[29]	1	0	0.1	1	1 / 0	0.1
Twofish[3]	1	0	0.1	1	1 / 0	0.2

Table 5: Results of leak detection with *symbolic* addresses: Is the given program leaky w.r.t. *any* adversarial thread?

Name	#.Acc	Precise			Two-Step		
		#.Inter	#.Test	Time(m)	#.Inter	#.Test step1 / step2	Time(m)
AES [6]	1,026	224	220	1016.4	224	220 / 220	237.5
AES[27]	2,568	141	139	>1600	256	302 / 254	548.3
Camellia[4]	2,590	176	172	830.8	176	172 / 172	303.5
CAST5[4]	1,815	167	164	>1600	384	381 / 381	1337.4
CAST5[27]	1,392	183	180	>1600	384	381 / 381	1392.5
Chaskey[29]	1,380	1	0	0.1	1	1 / 0	0.1
DES[3]	2,135	144	127	38.6	144	164 / 127	27.2
DES[27]	2,539	119	114	>1600	194	187 / 183	1191.5
FCrypt[27]	428	64	60	15.1	64	60 / 60	20.1
Kasumi[1]	1,785	83	82	>1600	96	94 / 94	151.9
Khazad[27]	684	114	103	>1600	248	254 / 240	165.3
KV_Name[21]	140	1406	0	0.5	1406	1406 / 0	0.5
LBlock[29]	4,068	1	0	0.1	1	1 / 0	0.1
Misty1[1]	2,966	76	75	>1600	96	94 / 94	265.1
Piccolo[29]	5,103	1	0	0.1	1	1 / 0	0.1
PRESENT[29]	8,233	1	0	0.2	1	1 / 0	0.2
rfc2268[3]	3,190	113	112	303.4	113	112 / 112	42.9
Seed[3]	1,632	201	197	>1600	320	316 / 316	1505.1
TWINE[29]	10,492	1	0	0.1	1	1 / 0	0.1
Twofish[3]	12,400	2514	84	>1600	900	84,063 / 76	>1600

Furthermore, our two-step approach returned exactly the same results as the precise analysis for all benchmark programs, but in significantly less time.

We also conducted our experiments using *4-way set-associative* cache instead of *direct-mapped* cache. The results of these experiments are similar to the ones reported in Table 4. Therefore, we omit them for brevity.

Nevertheless, the similarity is expected. For example, a 1024-byte S-Box would be mapped to 16 consecutive cache lines in *directed-mapped* cache as well as *4-way set-associative* cache, provided that the cache size is 64KB and the line size is 64-byte. The only minor difference is that, in the *4-way set-associative* cache, we need four adversarial memory accesses from thread T_2 to fully evict a cache set. But if we have already detected the first adversarial address (say *addr*), the remaining three could simply be *addr+cache_size*, *addr+2*cache_size*, and *addr+3*cache_size*. Thus, there is no significant difference from analyzing *direct-mapped* cache.

7.3 Results Obtained with Symbolic Addresses

The results shown in Table 4 are useful, but also somewhat *conservative*. A more aggressive analysis is to assume the adversarial thread T_2 may access memory regions whose cache layout is symbolic (refer to Case 2 in Section 2.3).

Table 5 shows the experimental results obtained using direct-mapped cache and symbolic addresses in thread T_2 (Case 2 in Section 2.3). The first two columns show the benchmark name and the maximum number of memory addresses accessed by an interleaving at run time. The *Precise* column shows the result of computing the precise solution for our cache analysis problem. The *Two-Step* column shows the result of running the simplified version. In both cases, we report the total number of interleavings explored by symbolic execution (#.Inter), the number of leaky memory accesses detected (#.Test), and the total execution time in minutes (m). For *Two-Step*, the number of leaky accesses is further divided into two subcolumns: the leaky accesses detected after the first step and the leaky accesses detected after the second step.

The results show that, for most of the benchmark programs, the overhead of precisely solving our cache analysis is too high: on nine of the twenty programs, it could not complete within the time limit. In contrast, our two-step analysis was able to complete nineteen out of the twenty programs. In terms of accuracy, our two-step approach is almost as good as precise analysis: in all completed programs, they detected the same number of leakage points, which indicate a possible combination of adversarial threads and memory layout that *can* trigger timing leaks.

Our results also show that, for the same type of cryptographic algorithms (such as AES), different implementations may lead to drastically different overhead. For example, we detected 34 more leakage points in the `AES` implementation of *Chronos* [27] than that of *OpenSSL* [6]. However, the `AES` of *Chronos* took almost twice as long for our tool to analyze. For `DES` implementations from *Libgcrypt* [3] and *Chronos* [27], we detected a slightly different number of leakage points, but the time taken is significantly different (27.1 minutes versus 1191.5 minutes). In contrast, for the two versions of `CAST5`, we detected the same number of leakage points in roughly the same amount of time.

For the benchmark where *Two-Step* took a long time, we found it is due to the increasing size of symbolic constraints which consist of the addresses in S-Box accesses. Typically the later a S-Box access in a loop, the larger its symbolic address expression would be. In `Twofish`, SymSC timed out because it encountered a large number of "may-be-related" event pairs (i.e., accessing the same S-Box but not the same cache line), which made SMT solving difficult.

7.4 Discussion

Based on the results, we answer the two research questions as follows. First, SymSC is able to identify cache timing leaks in concurrent programs automatically. Specifically, using *symbolic* addresses in the adversarial thread allows us to demonstrate the possibility of triggering leaks in a concurrent system, whereas using *fixed* addresses in the analysis allows us to show that such leaks are more practical. Second, SymSC's performance optimization techniques are effective in reducing the computational overhead, which is demonstrated on a diverse set of real-world cipher programs.

SymSC searches for sensitive inputs as well as an interleaving schedule that, together, trigger divergent cache behaviors. If an individual program path has a constant cache behavior, e.g., all the memory accesses refer to fixed memory addresses regardless of the value of the sensitive input, then timing leaks are impossible. By checking for and leveraging such conditions, SymSC can reduce the computation cost even further. For instance, with naive exploration, SymSC would have generated 1,406 interleavings for the benchmark program named `KV_name`. However, with the above analysis, it does not have to generate any interleaving.

In this example, `KV_name`'s 4-byte symbolic input only affects the branch conditions but does not taint any memory access address. Thus, many paths are explored by symbolic execution. However, no leak is detected on these paths.

Another example is `Chaskey`, which has a single program path, together with 1,380 memory accesses on this path. These memory addresses are all independent of the 16-Byte symbolic input, which means no leakage point can be found by SymSC.

8 RELATED WORK

Side-channel leaks have been exploited in a wide range of systems [28, 36–38, 43–45, 51, 54, 57]. For timing side channels, in particular, many analysis and verification techniques have been developed. For example, Chen et al. [23] proposed a technique named Cartesian Hoare Logic [58] for proving that the timing leaks of a program are bounded. Antonopoulos et al. [8] proposed a similar method for proving the absence of timing channels: it partitions the program paths in a way that, if individual partitions are proved to be timing attack resilient, the entire program is also timing attack resilient. However, these methods only consider *instruction-induced* timing while ignoring the cache.

In the context of analyzing real-time systems, there is a large body of work on cache analysis [49, 50, 52], with the goal of estimating the worst-case execution time (WCET). Various techniques including abstract interpretation [61], symbolic execution [12, 21], and interpolation [25] have been used to compute the upper bound of execution time along all program paths. Chattopadhyay et al. [22] also developed *CHALICE* to quantify information leaked through the cache side channel, but the focus was on dependencies between sensitive data and misses/hits on the CPU's data cache.

Doychev et al. [30] developed *CacheAudit*, a tool relying on abstract interpretation based static analysis to analyze cache timing leaks. Wang et al. [64] developed *CacheD*, an offline trace analysis tool for detecting key-dependent program points in a cipher program that may be vulnerable to side channel attacks. Sung et al. [60] developed *CANAL*, an LLVM transformation that models cache timing behaviors for standard verification tools. However, these techniques handle sequential programs or traces only.

Pasareanu et al. [55] developed a symbolic execution tool for reasoning about the degree of leaked information, assuming the attacker can take multiple measurements. The test input that causes the maximum amount of leakage is computed using Max-SMT solving. Bultan et al. [10, 17, 19] developed techniques for quantifying information leaked by string operations. Their method can handle both single and multiple runs [10]: it applies probabilistic symbolic execution to collect path constraints of a single run and then uses these constraints to compute the leakage of multiple runs. Phan et al. [56] also developed a symbolic attack model and formulated the problem of test generation to obtain the maximum leakage as an optimization problem.

However, in all these existing methods, the program is assumed to be sequential. In contrast, SymSC focuses on concurrency-induced leaks. Although Barthe et al. [11] proposed an abstract interpretation technique based on *CacheAudit* [30] to track the cache state of a program with concurrent adversary, the adversary is a separate process (that tries to probe and set the cache states), not a thread. Furthermore, users have to provide data inputs and interleaving schedules, whereas SymSC generates them automatically.

Stefan et al. [59] proposed an instruction-based scheduling mechanism in information flow control systems running on a single CPU, to avoid cache timing attacks introduced by classic time-based schedulers. Therefore, it is a system-level mitigation technique. In contrast, SymSC focuses on detecting whether a security-critical program may leak sensitive information through the timing side channel due to interference from other threads.

Our state-space reduction in SymSC is related to partial order reduction (POR) [35] in model checking, but with an important difference. In classic POR [9, 24, 42, 47, 63, 69], one would typically select representative interleavings from equivalence classes, which are defined based on standard data-conflict and data-dependence relations. However, in SymSC, they must be broadened to also include functionally-independent events that may access the same cache line.

So far, SymSC focuses on cases where the adversarial thread flushes a single cache line. In the terminology of side-channel analysis, this corresponds to *first-order* attacks. If, on the other hand, the adversarial thread is capable of flushing multiple cache lines, it may be more likely to trigger timing leaks. Such cases would be called *high-order* attacks. We leave the analysis of *high-order* attacks for future work.

Besides leak detection, there are side-channel leak mitigation techniques that can generate countermeasures automatically, e.g., using compiler-like program transformations [7, 13, 53, 65] or SMT solver based formal verification [15, 32, 33, 68] and program synthesis [16, 31, 34, 62] techniques. However, none of these emerging techniques was designed for, or applicable to, cache timing side channels due to concurrency.

9 CONCLUSIONS

We have presented a symbolic execution method for detecting cache timing leaks in a computation that runs concurrently with an adversarial thread. Our method systematically explores both thread paths and their interleavings, and relies on an SMT solver to detect divergent cache behaviors. Our experiments show that real cipher programs do have concurrency related cache timing leaks, and although it remains unclear *to what extent* such leaks are exploited in practice, our method computes concrete data inputs and interleaving schedules to demonstrate these leaks are realistic. To the best of our knowledge, this is the first symbolic execution method for detecting cache timing side-channel leaks due to concurrency.

ACKNOWLEDGMENTS

This work was supported in part by the U.S. National Science Foundation (NSF) under grants CNS-1617203 and CNS-1702824 and the Office of Naval Research (ONR) under grant N00014-17-1-2896.

REFERENCES

[1] *Botan.* https://botan.randombit.net/.

[2] *High Performance SSH/SCP - HPN-SSH.* https://www.psc.edu/hpn-ssh.

[3] *Libgcrypt.* https://gnupg.org/software/libgcrypt/index.html.

[4] *LibTomCrypt.* http://www.libtom.net/LibTomCrypt/.

[5] *OpenSSH.* http://www.openssh.com/.

[6] *OpenSSL.* https://github.com/openssl/openssl/tree/OpenSSL_0_9_7-stable.

[7] Giovanni Agosta, Alessandro Barenghi, and Gerardo Pelosi. A code morphing methodology to automate power analysis countermeasures. In *ACM/IEEE Design Automation Conference*, pages 77–82, 2012.

[8] Timos Antonopoulos, Paul Gazzillo, Michael Hicks, Eric Koskinen, Tachio Terauchi, and Shiyi Wei. Decomposition instead of self-composition for proving the absence of timing channels. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 362–375, 2017.

[9] Stavros Aronis, Bengt Jonsson, Magnus Lång, and Konstantinos Sagonas. Optimal dynamic partial order reduction with observers. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 229–248, 2018.

[10] Lucas Bang, Abdulbaki Aydin, Quoc-Sang Phan, Corina S. Pasareanu, and Tevfik Bultan. String analysis for side channels with segmented oracles. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 193–204, 2016.

[11] Gilles Barthe, Boris Köpf, Laurent Mauborgne, and Martín Ochoa. Leakage resilience against concurrent cache attacks. In *International Conference on Principles of Security and Trust*, pages 140–158, 2014.

[12] Tiyash Basu and Sudipta Chattopadhyay. Testing cache side-channel leakage. In *IEEE International Conference on Software Testing, Verification and Validation*, pages 51–60, 2017.

[13] Ali Galip Bayrak, Francesco Regazzoni, Philip Brisk, François-Xavier Standaert, and Paolo Ienne. A first step towards automatic application of power analysis countermeasures. In *ACM/IEEE Design Automation Conference*, pages 230–235, 2011.

[14] Tom Bergan, Dan Grossman, and Luis Ceze. Symbolic execution of multithreaded programs from arbitrary program contexts. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 491–506, 2014.

[15] Roderick Bloem, Hannes Groß, Rinat Iusupov, Bettina Könighofer, Stefan Mangard, and Johannes Winter. Formal verification of masked hardware implementations in the presence of glitches. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 321–353, 2018.

[16] Arthur Blot, Masaki Yamamoto, and Tachio Terauchi. Compositional synthesis of leakage resilient programs. In *International Conference on Principles of Security and Trust*, pages 277–297, 2017.

[17] Tegan Brennan, Seemanta Saha, and Tevfik Bultan. Symbolic path cost analysis for side-channel detection. In *International Conference on Software Engineering*, pages 424–425, 2018.

[18] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. Parallel symbolic execution for automated real-world software testing. In *European Conference on Computer Systems*, pages 183–198, 2011.

[19] Tevfik Bultan, Fang Yu, Muath Alkhalaf, and Abdulbaki Aydin. *String Analysis for Software Verification and Security.* Springer, 2017.

[20] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 209–224, 2008.

[21] Sudipta Chattopadhyay. Directed automated memory performance testing. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 38–55, 2017.

[22] Sudipta Chattopadhyay, Moritz Beck, Ahmed Rezine, and Andreas Zeller. Quantifying the information leak in cache attacks via symbolic execution. In *ACM-IEEE International Conference on Formal Methods and Models for System Design*, pages 25–35, 2017.

[23] Jia Chen, Yu Feng, and Isil Dillig. Precise detection of side-channel vulnerabilities using quantitative cartesian hoare logic. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 875–890, 2017.

[24] Lin Cheng, Zijiang Yang, and Chao Wang. Systematic reduction of GUI test sequences. In *IEEE/ACM International Conference On Automated Software Engineering*, pages 849–860, 2017.

[25] Duc-Hiep Chu, Joxan Jaffar, and Rasool Maghareh. Precise cache timing analysis via symbolic execution. In *IEEE Symposium on Real-Time and Embedded Technology and Applications*, pages 293–304, 2016.

[26] Liviu Ciortea, Cristian Zamfir, Stefan Bucur, Vitaly Chipounov, and George Candea. Cloud9: a software testing service. *Operating Systems Review*, 43(4):5–10, 2009.

[27] Matthew Dellinger, Piyush Garyali, and Binoy Ravindran. Chronos linux: a best-effort real-time multiprocessor linux kernel. In *ACM/IEEE Design Automation Conference*, pages 474–479, 2011.

[28] Jean-François Dhem, François Koeune, Philippe-Alexandre Leroux, Patrick Mestré, Jean-Jacques Quisquater, and Jean-Louis Willems. A practical implementation of the timing attack. In *International Conference on Smart Card Research and Applications*, pages 167–182, 1998.

[29] Daniel Dinu, Yann Le Corre, Dmitry Khovratovich, Léo Perrin, Johann Großschädl, and Alex Biryukov. Triathlon of lightweight block ciphers for the internet of things. Cryptology ePrint Archive, Report 2015/209, 2015.

[30] Goran Doychev, Dominik Feld, Boris Köpf, Laurent Mauborgne, and Jan Reineke. Cacheaudit: A tool for the static analysis of cache side channels. In *USENIX Security Symposium*, pages 431–446, 2013.

[31] Hassan Eldib and Chao Wang. Synthesis of masking countermeasures against side channel attacks. In *International Conference on Computer Aided Verification*, pages 114–130, 2014.

[32] Hassan Eldib, Chao Wang, and Patrick Schaumont. SMT-based verification of software countermeasures against side-channel attacks. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 62–77, 2014.

[33] Hassan Eldib, Chao Wang, Mostafa Taha, and Patrick Schaumont. QMS: Evaluating the side-channel resistance of masked software from source code. In *ACM/IEEE Design Automation Conference*, pages 209:1–6, 2014.

[34] Hassan Eldib, Meng Wu, and Chao Wang. Synthesis of fault-attack countermeasures for cryptographic circuits. In *International Conference on Computer Aided Verification*, pages 343–363, 2016.

[35] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 110–121, 2005.

[36] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic analysis: Concrete results. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 251–261, 2001.

[37] Daniel Genkin, Adi Shamir, and Eran Tromer. RSA key extraction via low-bandwidth acoustic cryptanalysis. In *Annual International Cryptology Conference (CRYPTO)*, pages 444–461, 2014.

[38] Daniel Gruss, Julian Lettner, Felix Schuster, Olga Ohrimenko, István Haller, and Manuel Costa. Strong and efficient cache side-channel protection using hardware transactional memory. In *USENIX Security Symposium*, pages 217–233, 2017.

[39] Shengjian Guo, Markus Kusano, and Chao Wang. Conc-iSE: incremental symbolic execution of concurrent software. In *IEEE/ACM International Conference On Automated Software Engineering*, pages 531–542, 2016.

[40] Shengjian Guo, Markus Kusano, Chao Wang, Zijiang Yang, and Aarti Gupta. Assertion guided symbolic execution of multithreaded programs. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 854–865, 2015.

[41] Shengjian Guo, Meng Wu, and Chao Wang. Symbolic execution of programmable logic controller code. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2017.

[42] Vineet Kahlon, Chao Wang, and Aarti Gupta. Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In *International Conference on Computer Aided Verification*, pages 398–413, 2009.

[43] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Annual International Cryptology Conference (CRYPTO)*, pages 104–113, 1996.

[44] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Annual International Cryptology Conference (CRYPTO)*, pages 388–397, 1999.

[45] Jingfei Kong, Onur Aciiçmez, Jean-Pierre Seifert, and Huiyang Zhou. Architecting against software cache-based side-channel attacks. *IEEE Trans. Computers*, 62(7):1276–1288, 2013.

[46] Boris Köpf, Laurent Mauborgne, and Martín Ochoa. Automatic quantification of cache side-channels. In *International Conference on Computer Aided Verification*, pages 564–580, 2012.

[47] Markus Kusano and Chao Wang. Assertion guided abstraction: a cooperative optimization for dynamic partial order reduction. In *IEEE/ACM International Conference On Automated Software Engineering*, pages 175–186, 2014.

[48] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *IEEE/ACM International Symposium on Code Generation and Optimization*, pages 75–88, 2004.

[49] Xianfeng Li, Tulika Mitra, and Abhik Roychoudhury. Accurate timing analysis by modeling caches, speculation and their interaction. In *ACM/IEEE Design Automation Conference*, pages 466–471, 2003.

[50] Yan Li, Vivy Suhendra, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. Timing analysis of concurrent programs running on shared cache multi-cores. In *IEEE Real-Time Systems Symposium*, pages 57–67, 2009.

[51] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks - revealing the secrets of smart cards.* 2007.

[52] Tulika Mitra, Jürgen Teich, and Lothar Thiele. Time-critical systems design: A survey. *IEEE Design & Test*, 35(2):8–26, 2018.

[53] Andrew Moss, Elisabeth Oswald, Dan Page, and Michael Tunstall. Compiler assisted masking. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 58–75, 2012.

[54] Elke De Mulder, Thomas Eisenbarth, and Patrick Schaumont. Identifying and eliminating side-channel leaks in programmable systems. *IEEE Design & Test*, 35(1):74–89, 2018.

[55] Corina S. Pasareanu, Quoc-Sang Phan, and Pasquale Malacaria. Multi-run side-channel analysis using symbolic execution and max-smt. In *IEEE Computer Security Foundations Symposium*, pages 387–400, 2016.

[56] Quoc-Sang Phan, Lucas Bang, Corina S. Pasareanu, Pasquale Malacaria, and Tevfik Bultan. Synthesis of adaptive side-channel attacks. In *IEEE Computer Security Foundations Symposium*, pages 328–342, 2017.

[57] Jean-Jacques Quisquater and David Samyde. *ElectroMagnetic Analysis (EMA): Measures and Counter-measures for Smart Cards*, pages 200–210. 2001.

[58] Marcelo Sousa and Isil Dillig. Cartesian hoare logic for verifying k-safety properties. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 57–69, 2016.

[59] Deian Stefan, Pablo Buiras, Edward Z. Yang, Amit Levy, David Terei, Alejandro Russo, and David Mazières. Eliminating cache-based timing attacks with instruction-based scheduling. In *European Symposium on Research in Computer Security*, pages 718–735, 2013.

[60] Chungha Sung, Brandon Paulsen, and Chao Wang. CANAL: A cache timing analysis framework via llvm transformation. In *IEEE/ACM International Conference On Automated Software Engineering*, 2018.

[61] Valentin Touzeau, Claire Maïza, David Monniaux, and Jan Reineke. Ascertaining uncertainty for efficient exact cache analysis. In *International Conference on Computer Aided Verification*, pages 22–40, 2017.

[62] Chao Wang and Patrick Schaumont. Security by compilation: an automated approach to comprehensive side-channel resistance. *ACM SIGLOG News*, 4(2):76–89, 2017.

[63] Chao Wang, Zijiang Yang, Vineet Kahlon, and Aarti Gupta. Peephole partial order reduction. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 382–396, 2008.

[64] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. CacheD: Identifying cache-based timing channels in production software. In *USENIX Security Symposium*, pages 235–252, 2017.

[65] Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. Eliminating timing side-channel leaks using program repair. In *International Symposium on Software Testing and Analysis*, 2018.

[66] Qiuping Yi, Zijiang Yang, Shengjian Guo, Chao Wang, Jian Liu, and Chen Zhao. Eliminating path redundancy via postconditioned symbolic execution. *IEEE Trans. Software Eng.*, 44(1):25–43, 2018.

[67] Tingting Yu, Tarannum S. Zaman, and Chao Wang. DESCRY: reproducing system-level concurrency failures. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 694–704, 2017.

[68] Jun Zhang, Pengfei Gao, Fu Song, and Chao Wang. SCInfer: Refinement-based verification of software countermeasures against side-channel attacks. In *International Conference on Computer Aided Verification*, 2018.

[69] Naling Zhang, Markus Kusano, and Chao Wang. Dynamic partial order reduction for relaxed memory models. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 250–259, 2015.