SpecuSym: Speculative Symbolic Execution for Cache Timing Leak Detection

Shengjian Guo*
Baidu Security
sjguo@baidu.com

Yueqi Chen*
Penn State University
yxc431@ist.psu.edu

Peng Li, Yueqiang Cheng
Baidu Security
{lipeng28,chengyueqiang}@baidu.com

Huibo Wang
Baidu Security
wanghuibo01@baidu.com

Meng Wu
Ant Financial Services Group
bode.wm@antfin.com

Zhiqiang Zuo
State Key Lab. for Novel Software
Technology, Nanjing University
zqzuo@nju.edu.cn

ABSTRACT

CPU cache is a limited but crucial storage component in modern processors, whereas the cache timing side-channel may inadvertently leak information through the physically measurable timing variance. Speculative execution, an essential processor optimization, and a source of such variances, can cause severe detriment on deliberate branch mispredictions. Despite static analysis could qualitatively verify the timing-leakage-free property under speculative execution, it is incapable of producing endorsements including inputs and speculated flows to diagnose leaks in depth. This work proposes a new symbolic execution based method, SpecuSym, for precisely detecting cache timing leaks introduced by speculative execution. Given a program (leakage-free in non-speculative execution), SpecuSym systematically explores the program state space, models speculative behavior at conditional branches, and accumulates the cache side effects along with subsequent path explorations. During the dynamic execution, SpecuSym constructs leak predicates for memory visits according to the specified cache model and conducts a constraint-solving based cache behavior analysis to inspect the new cache behaviors. We have implemented SpecuSym atop KLEE and evaluated it against 15 open-source benchmarks. Experimental results show that SpecuSym successfully detected from 2 to 61 leaks in 6 programs under 3 different cache settings and identified false positives in 2 programs reported by recent work.

CCS CONCEPTS

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

KEYWORDS

Speculative execution, cache, timing, side-channel leak, symbolic execution

*Both authors contributed equally to this research. Yueqi Chen worked on this project while he interned at Baidu USA.

1 INTRODUCTION

CPU cache is a limited but crucial storage area on modern processor chips. It primarily relieves the speed disparity between the rapid processors and the slow main memory, by buffering recently used data for faster reuse. Cache timing side-channel attacks [26, 47] leverage the distinct cache physical symptoms, i.e., the cache visiting latencies of various program executions, to penetrate the confidentiality of the victims. On exploiting the vulnerable software implementations, adversaries can extract the application secrets [16, 36, 60, 69], infer the neural network structure [30, 40, 41, 79], or even dump the kernel data [42, 46, 52, 75].

A timing side-channel generally serves as the intermediate carrier through which private data could inadvertently disclose to observers who can elaborately measure the timing information of certain operations. One particular instance is the cache timing side-channel, which leaks data by the variance of the cache visiting latency. State-of-the-art program repair method [77] mitigates cache timing leaks by enforcing constant execution time for all secret-relevant operations. However, this strong mitigation may still get compromised by the thread-level concurrency [38] or the instruction-level parallelism like *speculative execution* [44].

Speculative execution [44] is a microarchitectural optimization in modern processors. It primarily increases the CPU instruction pipeline throughput by beforehand scheduling instructions under predicted branches, which prevents control hazards from stalling the pipeline. Despite its essential importance, the cache side effects caused by prediction errors could engender severe detriment through the cache timing side-channel [15, 43, 46, 75].

Program analysis for speculative execution is by no means a new research domain. Previous efforts mainly researched safe and efficient execution [22, 35, 64], worst-case execution time estimation [50, 51], concurrency bug prediction [21, 49], and Spectre vulnerability detection [35, 58, 71]. Wu et al. [78] recently proposed a dedicated static analysis of timing-leakage-free property under speculative execution. However, this abstract interpretation based method [78] qualitatively answers the *yes* or *no* question — it is incapable of generating input and speculative flows to diagnose leaks in depth. Moreover, the over-approximation nature inevitably results in false positives, which desires a more precise method.
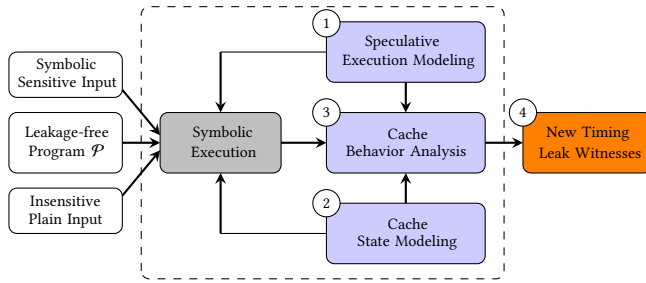
Figure 1: Overall flow of SPECUSYM.

To this end, we propose a new symbolic execution based method, SPECUSYM, for detecting cache timing leaks caused by speculative execution. Figure 1 displays the overall flow of SPECUSYM. Given a program $\mathcal{P}$, which is timing-leakage-free in non-speculative execution, the sensitive input presented in symbol, and the insensitive input, SPECUSYM leverages symbolic execution to explore $\mathcal{P}$'s state space systematically. Meanwhile, it models speculative execution at conditional branches (cf. ①) and accumulates cache side effects along with subsequent executions (cf. ②). Based on a cache model, SPECUSYM constructs leak predicates for memory visits and conducts a constraint-solving based cache behavior analysis (cf. ③) to generate the leak witnesses (cf. ④).

Our new method has three significant challenges. The first challenge comes from the modeling of speculative behaviors. Classic symbolic executors [17, 62] neither support speculative execution nor are cache-aware since they primarily concentrate on the functional correctness rather than reasoning the implicit program properties. The second challenge derives from the cache state maintenance. Due to the symbolic nature, a symbolic memory address may correspond to multiple concrete addresses. Updating the cache status after each memory operation unquestionably leads to an explosive number of different cache states. The last challenge stems from the analysis cost. Processors may trigger multiple branch mispredictions during program execution. Indiscriminately covering all possibilities introduces not only tremendous constraint solving overhead but also many unnecessary cases.

To overcome the first challenge, we design a new modeling algorithm in symbolic execution, which satisfies both feasibility and high-fidelity. In essence, it utilizes the stateful exploration to mimic the speculative behaviors and isolates memory changes in auxiliary states from the normal symbolic states. To tackle the second challenge, we develop a lazy modeling strategy that tracks memory accesses and lazily reasons about cache effects rather than maintaining a complete set of cache states on-the-fly. To address the last challenge, we filter the branches that are unlikely to cause harmful speculative execution. Also, we develop several optimizations to shrink the constraint size for solving cost reduction.

We have implemented SPECUSYM atop KLEE [17] and LLVM [48] and evaluated it on 15 benchmarks, which have 8,791 lines of C code in total. Results demonstrate that SPECUSYM successfully detects from 2 to 61 leaks in 6 programs under 3 set-associative caches and identifies false positives in 2 programs reported by recent work. To summarize, we have made the following contributions:

- A novel technique for modeling microarchitectural speculative execution and analyzing the affected cache behaviors in symbolic execution.

- The implementation of SPECUSYM, which addresses three major challenges and supports cache timing leak detection under speculative execution.
- The evaluation of SPECUSYM on 15 open-source benchmarks to demonstrate its effectiveness through revealing from 2 to 61 leaks under 3 different cache settings.

The remainder of this paper is organized as follows. Section 2 motivates our work, and Section 3 reviews the background knowledge. Section 4 states the core contributions and optimizations. Then we conduct experiments in Section 5 and discuss the related work in Section 6. Finally, we conclude our work in Section 7.

2 MOTIVATION

This section motivates our work with an example. By studying its leakage-free cache behavior under non-speculative execution and the new leaks caused by speculative execution, we position how SPECUSYM should facilitate leak detection.

2.1 Program $\mathcal{P}$ and the Cache Mapping

Figure 2(a) shows a program snippet $\mathcal{P}$ whose execution time remains stationary in non-speculative execution but varies in terms of the sensitive input when running under speculative execution.

Listed at line 2, $\mathcal{P}$ has 4 local variables as S, x, v1, and v2. Operating these variables, e.g., the implicit memory read of x (line 6) and the explicit *store* to v1 (line 7), may lead to memory access. The remaining variable i (line 3) is a register variable that incurs no memory access. Also, variable x is the sensitive input, and any form of revealing its value turns to be a leak.

We use a fully associative cache C for the analysis purpose of $\mathcal{P}$, as shown in Figure 2(b). It is an extreme case of the N-way associative cache where the memory address of a variable in $\mathcal{P}$ may map to any cache line of C, subjecting to the line availability and the replacement policy. Here we assume C adopts the *Least Recently Used* (LRU) policy, which always evicts the least used line once C has been entirely occupied.

Cache C consists of 256 cache lines, and each line has exactly 1-byte size. Local variables are mapped to C according to the program execution order and their sizes; e.g., in Figure 2(b) array S maps from cache lines #1 to #254 because of the array traverse in the *while* loop (lines 4-5), and each 1-byte array item successively occupies a full cache line. Then variable x associates with line #255. Next, v1 fills the last available line (line #256) if x>128 satisfies; thus, the execution proceeds into the *if* flow. Otherwise, the execution takes the *else* branch and writes v2, mapping v2 to line #256.

2.2 Leakage-free in Non-speculative Execution

Running $\mathcal{P}$ without speculative execution won't leak any information about x. This section analyzes the cache behavior in detail.

There are two program paths in $\mathcal{P}$ because of the *if-else* branch. Let's name the one containing the *if* branch as p_1 and the other one as p_2. The only difference between p_1 and p_2 is the *store* operation, which writes v1 on p_1 but v2 on p_2. As analyzed in Section 2.1, v1 would map to the last line of cache C. Similarly, without speculative execution, v2 also maps to the same line, as annotated by the solid arrow in Figure 2(b). This overlapping situation is because usually either p_1 or p_2 could be taken; hence either the *store* to v1 or the *store* to v2 may happen. In both cases, line #256 is empty before the *store* operations. Thereby, v1 or v2 uses this available line.
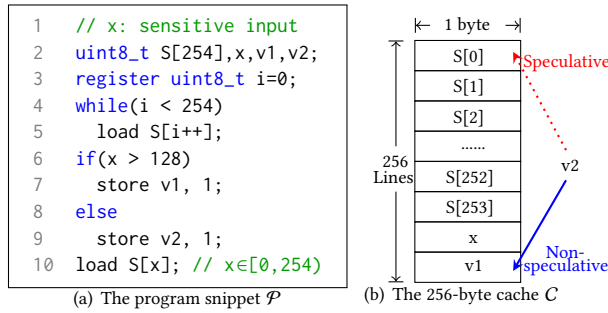
```
1    // x: sensitive input
2    uint8_t S[254],x,v1,v2;
3    register uint8_t i=0;
4    while(i < 254)
5      load S[i++];
6    if(x > 128)
7      store v1, 1;
8    else
9      store v2, 1;
10   load S[x]; // x∈[0,254)
```

(a) The program snippet $\mathcal{P}$ (b) The 256-byte cache C

Figure 2: Program $\mathcal{P}$ and its cache mapping.

In view of this fact, we can observe similar cache behaviors on both p_1 and p_2 under non-speculative execution. The first 254 *load* operations on array S, the next implicit memory read of x, and the following *store* to v1 on p_1 or to v2 on p_2, all cause cold misses since cache C is initially empty. Likewise, the last memory *load* on S[x] (line 10) must be a cache hit on both paths because S and x have already been in cache C. In other words, $\mathcal{P}$'s cache behavior is independent of the sensitive input x. As a result, $\mathcal{P}$ has no cache timing leaks under non-speculative execution.

2.3 New Leak under Speculative Execution

In Section 2.2, only one *store* operation can happen in non-speculative execution. However, the situation changes, and a new timing leak appears when taking speculative execution into account.

Under speculative execution, the instructions guarded by a branch *br* can be scheduled before the execution proceeds into *br* in case CPU predicts that *br* is likely to be taken. For example, suppose we first run $\mathcal{P}$ with x∈[128,255] several times and flush the cache after each run. Afterward, we rerun $\mathcal{P}$ but setting x to 127. Still, the *store* instruction under the *if* branch (line 7) would be executed before $\mathcal{P}$ steers into the *else* branch due to the branch misprediction. More importantly, though the CPU performs a rollback to discard the value update to v1, v1 remains in C even after the remedy.

Going until line 9, array S and variable x map from lines #1 to #255 and v1 still occupies line #256. At this point, there is no empty line available for v2. Following the LRU policy declared in Section 2.1, executing the *store* instruction at line 9 would evict the oldest item S[0] from cache C and map v2 to the vacated line #1, as shown by the dotted arrow in Figure 2(b).

After that, the program execution continues to line 10, reaching the last memory *load* of S[x]. Sensitive input x now determines which array cell would be visited. And we examine the cache behavior of this *load* in terms of two cases x=0 and x≠0.

- x=0: $\mathcal{P}$ reads array cell S[0]. Recall that x is still in cache C, but S[0] is no longer in C due to the replacement by v2. So this memory *load* causes a conflicting cache miss.
- x≠0: Since the whole array S except S[0] is in C, the *load* of S[x] must get a cache hit no matter what value x is.

Only if x=0 there appears one more cache miss on path p_2. It is a unique situation that enables attackers to learn the value of x due to a measurable longer execution time. Note that the cold miss from speculatively writing v1 also causes an internal latency. However, it is un-observable to the external users, and we ignore it safely for analysis purposes.

2.4 What SPECUSYM Should Provide

The motivating example shows that, though a program may have been carefully crafted to avoid cache timing leaks, running it under speculative execution could still exhibit new leaks. Since speculative execution is one of the fundamental optimizations in modern processors, a systematic analysis that detects the subtle leaks would be of vital importance. Specifically, we emphasize two requisite abilities in our proposed new method.

First, SPECUSYM should be able to systematically explore the program state space to identify execution paths and speculative flows that may cause leaks, e.g., p_2 and the speculative execution of the *if* branch. By contrast, speculation of the *else* branch won't cause any leak along with p_1.

Second, SPECUSYM should be able to pinpoint the leak sites by a precise cache analysis and generate concrete inputs that witness new cache behavior at the identified memory visit sites, e.g., the memory *store* instructions to v2 and the value *zero* of input x.

3 PRELIMINARIES

This section reviews the preliminary knowledge of symbolic execution, cache timing leak, and speculative execution.

3.1 Symbolic Execution

Symbolic execution, as a systematic program testing and analysis technique, was first introduced in the 1970s [24, 45]. In this work, we assume that a program $\mathcal{P}$ consists of a finite set of instructions, and $\mathcal{P}$ defines execution semantics in program paths. Let *inst* be an instruction, then interpreting an event $e := (l_b > inst > l_a)$ in symbolic execution stands for the execution of *inst* where l_b and l_a denote the locations before and after *inst*, respectively. A program execution explores a sequence of events along the program path. Close to [37, 38], we abstract a symbolic event e into three categories in terms of the type of *inst* that e contains.

- ψ-event, which presents a branch instruction. It models the *then* branch by assume(c) and the *else* branch by assume($\neg c$), respectively. Term c is the representative of a conditional predicate expressed in symbolic expression.
- χ-event, which corresponds to a memory *read* instruction of the form *var = load addr*, or a memory *write* instruction like *store addr, expr* where *addr* is the memory address, and *expr* is a symbolic expression.
- φ-event, which represents other types of instructions in the form *var := expr*. Here *var* is a variable, and *expr* is a symbolic expression computed from preceding events like arithmetic calculation, bit manipulation, etc.

Although symbolic executors always support a rich set of instructions, we use the above event types to abstract away the internal implementation details like memory allocation, function return, and et al., to center on the high-level flow of symbolic execution.

Algorithm 1 presents the baseline symbolic execution of a program with sensitive input. Unlike prior works [37, 38], both global and local memory accesses are uniformly handled in this algorithm. The data input, $in := \{\lambda, t\}$, determines a program execution path consisting of ordered events $\{e_1, \ldots, e_n\}$ where λ is the sensitive input, e.g., privacy data or cipher keys, and t is the insensitive input.

The global container STACK is used for storing symbolic states during the dynamic exploration. A symbolic state exhibits the frontier of a path execution. We use a tuple $\langle pc, e, brs, \Omega, \rangle$ to define a

---

Algorithm 1: Baseline Symbolic Execution.

Initially: The global state container STACK is empty (STACK⟵∅).
Start **BaseSymExec** (s_{ini}) on an initial state s_{ini} with $in := \{\lambda, t\}$.

1 **BaseSymExec** (SymbolicState s)
2 **begin**
3   STACK.push(s);
4   **if** $s.e$ *is branch event* **then**
5   **for** $c \in s.brs$ **and** $s.pc \wedge c$ *is satisfiable* **do**
6          $s.pc \leftarrow s.pc \wedge c$;
7   **BaseSymExec** (*SubsequentState*(s));   // The ψ event
8   **end**
9   **else if** $s.e$ *is memory access event* **then**
10   **BaseSymExec** (*SubsequentState*(s));   // The χ event
11   **else if** $s.e$ *is other interpretable event* **then**
12   **BaseSymExec** (*SubsequentState*(s));   // The φ event
13   **else**
14   Terminate state s and report error;
15   **end**
16   STACK.pop();
17 **end**

18 *SubsequentState* (SymbolicState s)
19 **begin**
20    $s' \leftarrow$ symbolically execute $e.inst$ in s;
21    $s'.e \leftarrow$ next available event;
22   **return** s';
23 **end**

---

state s. Symbol pc denotes the path condition that leads to s, where e is the event to execute at s. brs contains the set of branch predicates if e is a ψ event. And Ω is the symbolic memory which maintains the symbolic memory values of the program variables at state s.

Initially, the state container STACK is empty, and we start the main procedure BaseSymExec with the initial state s_{ini} on input in. During the recursive execution, BaseSymExec may split a branch (lines 4-8), perform a memory operation (lines 9-10), conduct an internal computation (lines 11-12), or terminate a state (line 14), depending on the type of the event to execute.

Note that at the entry of each recursion, BaseSymExec takes a new symbolic state as the input, which is obtained from invoking a secondary procedure *SubsequentState*. This procedure inputs the current state s and outputs a new state s' by symbolically executing the *inst* in $s.e$. For brevity, we omit the details of the instruction interpretation, which can be found in [17, 62].

3.2 Cache Timing Leak

Memory operations are prone to timing leaks because of the outstanding accessing latency between the cache and the main memory. For example, reading data from the cache may cost 1-3 processor cycles, whereas loading data from memory could spend hundreds of cycles. In this section, we first establish the threat model and then formalize the leak definition based on the threat model.

3.2.1 The Threat Model. As exhibited in Section 2.3, sensitive data involved in memory access may get leaked from the timing traffic of a memory write. To capture such kind of subtle leak in our analysis, we assume the attackers can perform strong external threats.

First, we assume the attackers share the same processor with the victim process. Hence they can learn the shared cache states by probe methods. Second, they are allowed to request the execution of the victim process. Third, they can observe the latency of the interested memory visits in the victim process. Our threat model is close to those used in practical attacks like [27, 60, 80], where the attackers can deduce the cache block states by measuring the timing traffic of either the victim or the attacker process. Moreover, this

model also appears in leak detection techniques like [14, 29, 74, 76]. Therefore, we believe it is a reasonable model for analysis purposes.

3.2.2 The Leak Definition. Formally, we abstract a sensitive data related program $\mathcal{P}$ to be a function $F_{\mathcal{P}}(in) \Rightarrow out$. $F_{\mathcal{P}}$ processes the data input $in := \{\lambda, t\}$ (cf. Section 3.1), and returns the output out. For example, assuming $\mathcal{P}$ is an encryption process, λ is the private key, and t is the content to encrypt; then, out is the ciphertext. Let $T(F_{\mathcal{P}}(in))$ denotes the execution time of $\mathcal{P}$ with input in under non-speculative execution. Different inputs may explore various program paths. However, since we focus on timing leaks introduced by speculative execution, here we assume the time of those non-speculative program executions remain similar or the same, no matter what the sensitive inputs are, which is,

$$\forall t, \lambda, \lambda' \, . \, T(F_{\mathcal{P}}(\lambda, t)) \simeq T(F_{\mathcal{P}}(\lambda', t)) \tag{1}$$

Symbols λ and λ' denote any two sensitive inputs, and t is still the public input. Nevertheless, since in practice, attackers may gain information by observing several memory visits or cache lines, we restrict the leak granularity at the memory operation level regarding our threat model. Specifically, let e be a memory event on executing $\mathcal{P}$ and the time of interpreting e with and without speculative execution be $T_s(\mathcal{P}_e(in))$ and $T(\mathcal{P}_e(in))$, we assume:

$$\forall t, \lambda, \lambda' \, . \, T(\mathcal{P}_e(\lambda, t)) \simeq T(\mathcal{P}_e(\lambda', t)) \tag{2}$$

Then the existence of a new cache timing leak under speculative execution can be checked by the following formula:

$$\exists t, \lambda, \lambda' \, . \, \big(\lambda \neq \lambda' \wedge T_s(\mathcal{P}_e(\lambda, t)) \neq T_s(\mathcal{P}_e(\lambda', t))\big) \vee$$
$$T(\mathcal{P}_e(\lambda, t)) \neq T_s(\mathcal{P}_e(\lambda', t)) \tag{3}$$

where a leak appears if (1) two different sensitive inputs can cause significant timing differences in executing e under speculative execution, or (2) two sensitive inputs can cause significant timing difference in executing e with and without speculative execution. In other words, $\mathcal{P}$ leaks due to speculative execution if any pair of λ and λ' exists. Furthermore, we transform formula (3) into a dedicated leak constraint and simplify the constraint to a more concise form in Section 4. Also, as public input plays a minor role in modeling cache timing leaks [73], we set insensitive input t to a fixed value to reduce the reasoning cost of formula (3).

3.3 Speculative Execution

The CPU instruction pipeline [65] allows overlapped executions of proper instructions where each instruction execution consists of a series of micro-stages. This instruction-level parallelism benefits hardware utilization since instruction can start execution before the time its prior instructions have completed all their stages. However, a pipelined processor may get stalled once the program control flow needs to divert, but the destination remains unknown (e.g., at a conditional branch). Accordingly, the pipeline has to wait until the flow decision gets computed.

To alleviate the cost of such control hazard, processors leverage *speculative execution* [44] and *branch prediction* [54] to reduce the delay that could incur on conditional branch instructions. Generally, they predict the execution flow based on the history of recently executed branches and schedule instructions under predicted branches ahead of jumping into these branches. Specifically, on approaching a control hazard, the processors first predict a branch to take. Then, they execute the instructions under the selected branch and

maintain the temporary path state in a dedicated buffer. Finally, they commit the buffered state to continue the program flow if they achieved a correct prediction. Otherwise, upon an incorrect prediction, they have to discard the temporary state to revert the effects of the executed instructions hence avoiding functional errors.

This rollback mechanism, unfortunately, withdraws no affected cache state, which raises security risks. As shown in Section 2.3, variable v1 maps to cache line #256 due to the misprediction of the *if* branch; and, this cache effect remains unchanged, even control flow directs to the *else* branch. As a result, sensitive data x leaks due to the interfered cache state. Abstracting away the hardware details, we can model the speculative behavior in symbolic execution as a three-phase analysis and evaluate its cache side effects with a constraint-solving based approach.

3.3.1 Misprediction Modeling. As aforementioned, on reaching a control hazard, the processors make a prediction to select a branch for execution. Despite the experience-based hardware realization in CPUs, we can model this behavior by an auxiliary symbolic state. To be specific, before diverging the control flow into the *else* branch, our symbolic executor would decide to duplicate a new state from the current symbolic state and schedule it immediately into the *if* flow, which models the branch prediction of the *if* branch. Similarly, we can model the misprediction of the *else* branch on demand.

3.3.2 Speculative State Execution. Each duplicated state in *misprediction modeling* has the same snapshot of its parent state. For clarity, we use the *speculative* state to alias this newly forked state. A *speculative* state would be prioritized to the front of the state container in the symbolic executor and uninterruptedly executes until it meets a predefined threshold, like the size of the Reorder Buffer (ROB), the pipeline stage number, the branch depth, etc. Also, a *speculative* state runs independently from its parent state; thus, any memory updates in the *speculative* state won't taint its parent state who is waiting for the *speculative* state returns. For the sake of cache analysis, we also maintain the cache data in each symbolic state. A *speculative* state inherits such data from the parent state and keeps updating the cache during its execution.

3.3.3 Rollback and Cache Merging. Once the *speculative* state reaches the threshold, it has to stop and exit the state stack. Right before the termination, it notifies the awaiting parent state the finish of the speculation and transfers the latest cache data back to its parent. Upon accepting the notification, the parent symbolic state merges the received cache information into its cache data, to form the latest cache state. After that, the parent state aborts waiting and resumes the regular execution. This step models the processor rollback in high fidelity and retains the cache status changes from speculation. Since the cache changes have already merged into the parent state, terminating the *speculative* state causes no further effects.

It is worth noting that in this work, we constrain the scope of *speculative execution* to customary *branch prediction* and subsequent out-of-order instruction execution. In practice, modern processors may perform other forms of speculation, e.g., memory dependence speculation and disambiguation [55, 56, 59, 66]. However, they are not the main focus of this work.

4 ALGORITHM

In this section, we explain the core technical contributions portrayed in Figure 1. Algorithm 2 shows the procedure of SpecuSym

---

Algorithm 2: Symbolic Execution in SpecuSym .

Initially: The global state container Stack is empty (Stack⇐∅).;
Start **SpecuSym** (s_{ini}) on an initial symbolic state s_{ini} with $in := \{\lambda, t\}$.

1 **SpecuSym** (SymbolicState s)
2 **begin**
3   **if** *s is a speculative state* **and** *s reaches the threshold* **then**
4   **return;**
5   **end**
6   Stack.push(s);
7   **if** *s.e is branch event* **then**
8   **for** $c \in s.brs$ and $s.pc \wedge c$ is satisfiable **do**
9   *SpeculativeExplore*(s, c);   // Enter speculative modeling
10   ......   // Resume normal execution
11   **end**
12   **else if** *s.e is memory access event* **then**
13   *AnalyzeCache*(s);
14       $s.\pi \leftarrow$ update cache state by interpreting $s.e$;
15   **SpecuSym** (*SubsequentState*(s));
16   ......
17 **end**

18 *SpeculativeExplore*(SymbolicState s, Predicate c)
19 **begin**
20   **if** *c relies on a memory access* **then**
21       $s' \leftarrow$ duplicate state s;   // Fork speculative state
22       $s' \leftarrow$ redirect s' to the $\neg c$ control flow; // Negate branch direction
23   **SpecuSym** (*SubsequentState*(s'));
24       $s.\pi \leftarrow s'.\pi$;
25   Terminate s';
26   **end**
27 **end**

28 *AnalyzeCache*(SymbolicState s)
29 **begin**
30   **if** *s is a regular symbolic state* **and** *s.e relates to sensitive input* **then**
31       $\theta \leftarrow$ build the leak constraint for $s.e$;
32   **if** $s.pc \wedge \theta$ is satisfiable **then**
33   Generate witness;
34   **end**
35   **end**
36 **end**

---

built upon the baseline algorithm, where the main changes are highlighted at the state checkpoint (lines 3-5), the branch point (line 9), and the memory access point (lines 13-14).

4.1 Speculative Modeling

This section explains how SpecuSym models branch misprediction and speculative execution by introducing an auxiliary speculative state and orchestrating it with the normal symbolic state properly.

4.1.1 Modeling Overview. On entering the SpecuSym procedure in Algorithm 2, we first check if current state s is a *speculative state* and whether s has reached the predefined threshold (line 3). If both conditions meet, the recursive symbolic execution on s stops and returns immediately (line 4). Note that a state becomes a *speculative* state if it is duplicated from a normal symbolic state (line 21) by invoking *SpeculativeExplore* at a branch event (line 9).

Lines 18-27 in Algorithm 2 present the modeling procedure *SpeculativeExplore*. Generally, at a conditional branch (line 8), we call *SpeculativeExplore* (line 9) to start the speculative probe. That is, if the branch predicate closely relates to memory visit (line 20), e.g., using a variable for the first time, we duplicate a new state s' from current state s (line 21) with a negated branch direction (line 22). This assumption bases on the observation that the memory visiting latency at a branch may form a time window for speculative execution to load data into the cache. Otherwise, despite a branch misprediction, CPU may spend only several cycles on speculative execution before its rollback, which is too transient to let speculation affect the cache.
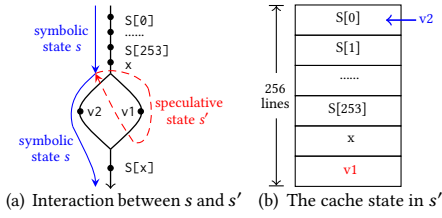
(a) Interaction between s and s' (b) The cache state in s'

Figure 3: The speculative execution modeling of $\mathcal{P}$.

After state duplication, new state s' becomes a *speculative* state, since it is about to mimic the speculation of the mispredicted branch. We let SpecuSym explore s' (line 23) and assess the effects of the memory accesses in s' on the cache (line 15). Details of the assessment present in Section 4.2. Once the execution of s' finishes, we use the accumulated cache data in s' to update that of s (line 24) and terminate s' (line 25) to end its lifecycle. Such design ensures that the speculation of s' only contributes to the cache state changes but never affects the memory Ω of its parent symbolic state.

Next, after *SpeculativeExplore* returns, we resume the normal symbolic execution of s (line 10), who now has an updated cache. In this way, we have constructed a speculative scenario and retained the latest cache changes. Our modeling method leverages the stateful mechanism of symbolic execution. It not only models the speculative behavior but also precisely transfers the cache information between symbolic states with controllable flexibility.

4.1.2 The Motivating Example Revisit. Figure 3 shows the speculative modeling of motivating program $\mathcal{P}$. Figure 3(a) displays the simplified control flow graph on which we annotate the memory access related variables. That is, S[0] indicates the first array read in the *while* loop, and v1 means the memory *write* in the *if* branch. The solid arrow line denotes the execution flow of state s, which takes the *else* branch in regular symbolic execution.

In Figure 3(a), on reaching the branch point, SpecuSym duplicates a *speculative* state s' from s. Then, it enforces a bounded symbolic execution of s', e.g., one memory access inside the *if* branch, as shown by the dashed arrow curve. Once meeting the speculation threshold, SpecuSym stops the exploration and turns back to find state s who is awaiting the finish of s'. Also, before resuming s, SpecuSym incorporates the cache state of s' (cf. Figure 3(b)) into s and terminates s'. Right now, the entire cache has been filled since the memory write in s' mapped v1 into line #256.

Subsequently, executing the memory *store* to v2 in state s has to evict S[0] from cache line #1 following the LRU policy, as shown in Figure 3(b). Then, the last memory *load* of S[x] might result in a cache miss or a hit, counting on the value of x, which is consistent with the situation explained in Section 2.3. Hence, SpecuSym succeeds in modeling speculative execution and conforms to the three-phase analysis designed in Section 3.3. Note that Figure 3(a) only shows the modeling of the *if* branch while SpecuSym considers both branches and misses no potential cases.

In this example, updating the cache state is straightforward. By checking the in-cache addresses to find a proper cache line upon a cache miss, or update the most recently used line in case of a hit. For the non-deterministic address, i.e., S[x], we can, at worst, try 254 possibilities to test the differences. However, eagerly enumerating all potentials is impractical for real-world programs due to the unbearable overhead. Furthermore, a great many cache mappings

are redundant in terms of leak exposure. For instance, only one specific cache layout can reveal the leak in the motivating example.

Instead, SpecuSym models branch mispredictions along with a path exploration and maintains a trace of memory events as the alternative of the cache state. Then it analyzes the event trace to build the leak constraint and lazily searches for feasible solutions. We detail this approach in Section 4.2 and Section 4.3.

As discussed before, the processor stops speculative execution once it has computed the branch destination. Since speculation may end at an arbitrary point, we model the speculative window with a configurable three-dimensional threshold. That is, if the number of interpreted events in a *speculative* state or the speculatively executed branch events meet the predefined bound, we stop speculation; and, if a sensitive input related memory event in speculative execution must cause a cache miss, we also stop the speculation after its execution. Otherwise, we continue the execution.

4.2 Cache State Modeling

This section addresses how SpecuSym models the cache state during dynamic symbolic execution. We extend the definition of a symbolic state s (cf. Section 3.1) to form a new tuple $\langle pc, e, brs, \Omega, \pi \rangle$. The newly introduced symbol π denotes the memory event trace in state s. Then we establish the following notions:

- A program state in SpecuSym is either a normal *symbolic* state or a *speculative* state that from the former can the latter be duplicated, but not vice versa.
- A memory event trace denoted as $\pi := \{m_0, ..., m_n\}$, consists of happened memory events in the execution order.
- Each memory event m_i in π, where index $i \in [0, n]$, comes from either a *symbolic* state or a *speculative* state.
- π and the correlated computations alternatively represent the cache state in a program state s.

To be specific, before interpreting a memory event m_i in a *symbolic* state s, SpecuSym analyzes if m_i can cause timing leak under two scenarios, namely the new *divergent* behavior and the new *opposite* behavior. Note that SpecuSym won't perform the analyses in any *speculative* state. To explain these scenarios, we define four necessary notions as follows.

- a_i denotes the used memory address in event m_i.
- $cs(a_i)$ denotes the cache set that address a_i maps to.
- $tag(a_i)$ denotes the unique *tag* of address a_i.
- N denotes the cache associativity where N$\in$[1, the total number of cache lines] (e.g., N-way set-associative cache).

4.2.1 New Opposite Behavior. It means that executing m_i under non-speculative execution causes an always-miss, while the result under speculative execution is an always-hit, and vice versa. Recall that we only target new leaks introduced by speculative execution, and this scene depicts the exact opposite situation. To build the formal definition, we define three new notions:

- $\eta_{in}(m_i)$ denotes the condition that m_i triggers an always-hit under speculative execution with input *in*.
- $\eta'_{in}(m_i)$ denotes the condition that m_i triggers an always-hit under non-speculative execution with input *in*.
- Boolean variable $org(m_i)$ denotes the origin of m_i, where 0 means m_i is from a normal *symbolic* state, and 1 means m_i comes from a *speculative* state.

Then we can formalize $\eta_{in}(m_i)$ and $\eta'_{in}(m_i)$ as follows.

$$\eta_{in}(m_i) := \underbrace{\exists j \in [0, i)| \wedge a_j = a_i \wedge \forall x \in (j, i)|a_x \neq a_i}_{\text{①} \ Find \ the \ nearest \ identical \ address \ a_j} \wedge$$

$$\underbrace{\sum_{y=j+1}^{i-1} cs(a_j) = cs(a_y) \wedge \nexists z \in (j, y)|tag(a_z) = tag(a_y)}_{\text{②} \ Count \ the \ unique \ a_y \ who \ and \ a_i \ map \ to \ the \ same \ set} < N \quad (4)$$

$$\eta'_{in}(m_i) := \eta_{in}(m_i) \wedge \underbrace{org(a_j) \neq 1 \wedge org(a_x) \neq 1 \wedge org(a_y) \neq 1}_{\text{③} \ Filter \ events \ from \ speculative \ execution} \quad (5)$$

Given a m_i, we first search π for an identical address a_j, which was visited in event m_j happened before m_i. Also, a_j has to be the nearest candidate that no other addresses between a_j and a_i are qualified (cf. ①). Second, we count the unique addresses who and m_i map to the same cache set and assure the number of such addresses is less than the set associativity N (cf. ②).

Based on $\eta_{in}(m_i)$ we define $\eta'_{in}(m_i)$ who has an extra constraint (cf. ③) upon $\eta_{in}(m_i)$. The new constraint ensures that all involved memory events must be from normal *symbolic* states hence filtering the influences of memory events from *speculative* states.

4.2.2 New Divergent Behavior. It means executing m_i under speculative execution with two different inputs *in* and *in'* can cause a miss and a hit, respectively. For instance, reading S[x] exposes this symptom (cf. Section 2.3). To analyze the behavior, we build $\mu_{in}(m_i)$ which returns the cache hit condition of m_i in trace π:

$$\mu_{in}(m_i) := \bigvee_{0 \leq j < i} \Bigg(\underbrace{tag(a_j) = tag(a_i) \wedge cs(a_j) = cs(a_i)}_{\text{④} \ Identify \ the \ potential \ a_j} \wedge$$

$$\underbrace{\forall x \in (j, i)|tag(a_x) \neq tag(a_i) \vee cs(a_x) \neq cs(a_i)}_{\text{⑤} \ Find \ the \ nearest \ a_j} \wedge$$

$$\underbrace{\sum_{y=j+1}^{i-1} cs(a_j) = cs(a_y) \wedge \nexists z \in (j, y)|tag(a_z) = tag(a_y)}_{\text{⑥} \ Count \ the \ unique \ a_y \ who \ and \ a_i \ map \ to \ the \ same \ set} < N \Bigg) \quad (6)$$

Still, given a m_i, we first seek a prior m_j from π who might let m_i cause a cache hit. That is, the addresses a_j and a_i have the same *tag* and *set* values (cf. ④). Moreover, we need to find the nearest m_j since m_i's cache behavior directly relies on it — this demand is achieved by ensuring the non-existence of a m_x between m_j and m_i. And, m_x and m_i own the same *tag* and *set* values (cf. ⑤).

Next, we consider the cache replacement policy. SPECUSYM models the N-way set-associative cache with the LRU policy, whereas other policies can be modeled and embedded in equation (6) as well. Intuitively, on finding a candidate m_j for m_i, the executed events between m_j and m_i should not evict m_j from the cache, to promise a cache hit for m_i. Under the LRU policy, we observe that if satisfying property 1, can this non-evicting requirement be met.

PROPERTY 1. *To avoid evicting the most recently used cache line from a cache set Θ, the total number of subsequent new cache mappings to Θ must be less than the set associativity.*

Sub-formula ⑥ checks whether an event m_y between m_j and m_i can form a new mapping to the cache set that address a_j maps to.

We perform the check by first comparing the *set* values of a_y and a_j. If the comparison satisfies, we further analyze if address a_y has never been accessed by an event m_z before — by confirming that the *tag* values of a_z and a_y always differ. If this check also satisfies, then a_y must form a new cache mapping. We count the amount of new mappings and model PROPERTY 1 in equation (6).

Take a 9-event trace $\pi := \{m_1, m_2, \underline{m_1, m_3, m_3, m_4, m_5, m_4, m_1}\}$ as the example. For brevity, we assume all the memory addresses associate with the same cache set, and each address corresponds to one cache line. And, the cache set associativity is *four*. The question is, whether the last event, m_1, can lead to a cache hit. Here we use #n to index these events where $n \in [1,9]$.

Backtracking from the last event m_1, we first locate two prior m_1 events, #1 and #3, as the candidates that satisfy ④. Then, according to ⑤, we select the nearest event #3, to form a sub-trace ρ as underlined in π. Note that after ρ's head event m_1, the cache line used by this m_1 becomes the most recently used line. Let's name the line as l. Then, following ⑥, along with ρ we can identify *3* new cache mappings incurred by events #4 (m_3), #6 (m_4), and #7 (m_5), respectively. Other events, i.e., #5 (m_3) and #8 (m_4), cannot form new mappings since they do not satisfy ⑥.

Therefore, we conclude that the target event, m_1, must lead to a cache hit because the identified *three* new cache mappings at most evict the other three lines rather than the line l. In other words, the number of new mappings is less than the set associativity of the given cache, which conforms to PROPERTY 1.

4.3 Cache Behavior Analysis

This section leverages the formal terms $\mu_{in}(m_i)$, $\eta_{in}(m_i)$ and $\eta'_{in}(m_i)$ (cf. Section 4.2) to analyze the cache behavior of m_i. In general, if m_i has no cache behavior variance in both speculative execution and non-speculative execution for any inputs, it has no leaks. Otherwise, at the program location of m_i, there is a leak.

In Algorithm 2, SPECUSYM invokes *AnalyzeCache* right before interpreting a memory event (line 13). In this procedure, SPECUSYM first examines whether the current state s is a *symbolic* state, and the event $s.e$ relates to sensitive input (line 30). It is because the timing effect inside speculative execution is externally invisible, and we are interested in the sensitive data related memory accesses. Next, SPECUSYM builds the leak constraint θ for event $s.e$ and solves for a solution (lines 31-33). If the solution does not exist, SPECUSYM claims leakage-free at $s.e$ in current state s.

To form θ, we need to build the constraint exposing new cache behaviors at a given memory event m_i as follows.

$$opp_i := \forall in, in'.(\eta_{in}(m_i) \neq \eta'_{in'}(m_i)) \quad (7)$$

$$div_i := \exists in, in'.(in \neq in' \wedge \mu_{in}(m_i) \neq \mu_{in'}(m_i)) \quad (8)$$

Constraint opp_i checks the existence of *New Opposite Behavior* and div_i reasons *New Divergent Behavior*, respectively. We define $\theta := opp_i \vee div_i$ as the leak constraint at m_i. SPECUSYM first computes $\eta'_{in'}(m_i)$ because of the fewer cost of a *must-be* solving. Then, it uses the concretized value to substitute $\eta'_{in'}(m_i)$ in opp_i for the further solving of *in*. If this targeted *in* exists under the path condition, then opp_i must be satisfiable, and SPECUSYM can safely skip the rest analysis of div_i. Otherwise, it has to continue reasoning div_i. Finally, if the solver successfully returns a concrete solution of θ, SPECUSYM generates a test case including two inputs, the trace π, and event m_i as the leak witness.

For instance, in Figure 3(a), before interpreting the `load` event of `S[x]`, SpecuSym analyzes the cache since the sensitive data related address `S[x]` is from a *symbolic* state s. At this point, trace π in s is {`S[0]`,...,`S[254]`, x,v2,v1,`S[x]`}. Also, in non-speculative execution reading `S[x]` must get a cache hit (cf. Section 2.2). Substituting $\eta'_{in'}(m_i)$ with value 1 in opp_i, SpecuSym tries to solve $\forall in.\eta_{in}(m_i) \neq 1$. Unfortunately, this simplified opp_i is still unsatisfiable since $\eta_{in}(m_i)$ evaluates to 1 if $in \neq 0$. Therefore, SpecuSym moves forward to div_i. This time, it successfully returns two values 0 and 1, which expose a cache miss and a hit, respectively. Finally, SpecuSym outputs the solved values, the trace $s.\pi$, and the *read* event of `S[x]`.

It is worthwhile to note that processors may mispredict branches several times along a program path. As a result, we need to count the cache side effects from multiple mispredictions. However, attackers always expect the minimum effort to trigger mispredictions for leak purposes. Meanwhile, manipulating multiple speculation windows is very difficult for external attackers. In SpecuSym, we follow the attacker's perspective to study whether one misprediction could be enough to cause timing leaks and leave the multiple speculation cases as our future work.

4.4 Optimizations

4.4.1 Constraint Transformation. In Section 4.3, constraints opp_i and div_i capture the complete set of leak-related cache behaviors. However, based on the timing-leakage-free assumption of a target program under non-speculative execution, we observe a property which helps SpecuSym accelerate the cache analysis:

PROPERTY 2. *If under the same input $in := \{\lambda, t\}$, the cache behaviors of a λ-related memory event m_i in speculative execution and non-speculative execution are opposite, there is a cache timing leak.*

Property 2 reflects the intersection of opp_i and div_i. As assumed, the cache behavior of m_i in non-speculative execution is deterministic. In case an opposite behavior under speculative execution, m_i would cause either new opposite behavior or new divergent behavior. In any case, it is a leak. Then the solving of θ turns to be reasoning the existence of such in, which has cheaper computation cost. We model the property into a new constraint θ':

$$\theta' := \exists in.\mu_{in}(m_i) \neq \eta'_{in}(m_i) \tag{9}$$

Due to the assumption mentioned above, $\eta'_{in}(m_i)$ is always 1 (hit) or 0 (miss). Accordingly, if $\mu_{in}(m_i)$ (may-hit condition under speculation) evaluates to 0 or 1 on the same in, either opp_i or div_i may evaluate to 1, which already deduces a leak without solving opp_i or div_i. Therefore, if θ' is satisfiable, θ also satisfies.

4.4.2 Formula Reduction. However, the size of θ' still intensely increases along with the increment of i, which pressures the constraint solver. By inspecting the constraint structure, we separate lengthy formulas into smaller pieces by following strategies. First, if a subformula of a Conjunctive Normal Form (CNF) formula evaluates as *false*, we directly return *false*. For example, in equation (6), if ④ is *false*, we skip querying solver for ⑤ and ⑥. Second, if a subformula in a Disjunction Normal Form (DNF) formula is determined as *true*, we discard all other subformulas. As in equation (6), once we find the nearest a_j by ⑤, we discard subformulas for indices before j. Third, if two subformulas are in negated forms, we avoid an extra solver query. For example, in equation 6, constraint ⑤ of m_j is the negation of constraint ④ of m_{j+1}, which implies we can save one constraint solving time. Fourth, since we repeatedly

traverse memory event trace in constraint construction, caching the intermediate results avoids repeated computation. We bank the computed formulas into a hash map so that SpecuSym can quickly retrieve them with tolerable storage overhead.

4.4.3 Speculative Assumption Checking. Recall that in Algorithm 2, SpecuSym duplicates a new state s' from symbolic state s (line 21) if the branch predicate closely relates to a memory value rather than a cache value. As mentioned earlier in Section 4.2, this strategy filters transient speculations that are unlikely to affect the cache. Wu et al. [78] over-approximately treated all memory events as actual memory visits, though predicate-related values may have been loaded to the cache. SpecuSym proposes an alternative for symbolic execution to deal with this issue. Ideally, to address the problem, we have to perform a burdensome backward cache analysis from the current branch to the first memory access, examining whether the value involved in the branch predicate is in cache or not. However, we observe that the memory accesses from the current branch to the last branch typically have a critical influence on the cache state before the current branch. Therefore, to balance the precision and efficiency, SpecuSym backwardly examines the memory accesses in this range. If the predicate-related values have been fetched to cache beforehand and still exist in the cache, we regard the speculative modeling at this branch as unnecessary. Otherwise, we still conduct all analyses as usual, as usual, to avoid missing leaks. Although this optimization introduces extra computation overhead, it still benefits overall performance since the targeted cases commonly exist in programs because of the time and spatial locality purposes.

5 EVALUATIONS

We have implemented SpecuSym based on the KLEE [17] symbolic executor and the LLVM compiler [48]. We refit KLEE in three main aspects. First, we made KLEE support the bounded execution of auxiliary *speculative* state and schedule the parent *symbolic* state afterward, to mimic the branch misprediction. Second, we made KLEE stitch a sequence of memory events from both normal *symbolic* state and *speculative* states, to represent the incorporation of cache data from speculation. Third, we made KLEE analyze the cache behaviors on memory events to generate concrete test cases for timing leaks. Based on these new components, we built SpecuSym for the cache timing leak analysis under speculative execution.

Specifically, after loading the LLVM bit-code of the target program, SpecuSym executes it symbolically, explores speculative states, records the memory event trace, and conducts cache analysis following Algorithm 2. The leak constraint is encoded in Z3-compatible form. On solving the constructed constraints, SpecuSym outputs leak witness, including the event trace, the leaky memory operation location, and the solved inputs.

We design the following research questions for the experiments:

- Can SpecuSym successfully identify cache timing leaks introduced by speculative execution?
- Can SpecuSym complement the abstract interpretation based method by providing more accurate results?
- Can the proposed optimizations boost the overall performance of SpecuSym?

5.1 Benchmark Programs

Table 1 shows the statistics of the benchmarks for evaluation. The first three columns, **Name**, **LoC**, and **Source**, indicate the names,

Table 1: Benchmark statistics: Name, Lines of C code (LoC), Source, Sensitive Input Size in Bytes (S-In), and Number of Branches (Brs).

Name	LoC	Source	S-In	Brs
hash[2]	320	The hpn-ssh hash implementation	64	5
AES[4]	1,838	The LibTomcrypt AES cipher	16	17
blowfish[4]	467	The LibTomcrypt blowfish cipher	8	11
chacha20[4]	776	The LibTomcrypt chacha20 cipher	36	133
encoder[4]	134	The LibTomcrypt hex encoder	100	2
ocb[4]	377	The LibTomcrypt OCB implementation	28	23
DES[5]	1,100	The OpenSSL DES cipher	64	79
str2key[5]	371	The OpenSSL key prepare for DES	16	8
DES[1]	547	The glibc DES implementation	16	7
Camellia[7]	1,324	The NVIDIA Tegra Camellia cipher	16	16
Salsa[7]	279	The NVIDIA Tegra Salsa20 stream cipher	28	12
Seed[7]	487	The NVIDIA Tegra Seed cipher	16	2
DES[3]	337	The Libgcrypt DES cipher	8	3
Salsa[3]	344	The Libgcrypt Salsa20 stream cipher	40	22
Spectre[6]	90	The Spectre V1 application	4	1

the lines of code, and the sources of these benchmarks. Column **S-In** denotes the sensitive input size in bytes. The last column **Brs** shows the number of conditional branches in each program.

Our benchmark suite consists of a diverse set of open-source C programs. The first program comes from hpn-ssh [2]. The second group has five programs from LibTomCrypt 1.18.1 [4]. The third group uses two programs from glibc 2.29 [1]. The fourth group includes three programs from the Tegra library [7]. The rest benchmarks are from the Libgcrypt 1.8.4 [3] and the Spectre vulnerability application [6]. Most benchmarks are computation-intensive despite the compact program volumes. The sensitive input of each benchmark is initialized to a symbol whose size shows in column **S-In**. Also, each program has 1 to 133 conditional branches.

We evaluate the benchmarks on the N-way set-associative cache and LRU policy with different cache settings. To be specific, we design four caches as (1) 32KB cache size with N=4; (2) 32KB cache size with N=8; (3) 64KB cache size with N=8; and (4) 32KB cache size with N=512. Each cache line has 64 bytes for all caches. The first three settings are close to the L1 data cache classes in modern processors. The last setting creates a fully-associative cache on which we compare SPECUSYM with [78]. Besides, we evaluate the effectiveness of optimizations using setting (3). All the evaluations are conducted on a machine running Ubuntu 16.04 64-bit Server Linux with Intel(R) Xeon(R) 2.20GHz CPUs 24 cores and 256GB RAM. Each benchmark is allowed to run at most 12 hours.

5.2 Experimental Results

5.2.1 Timing Leak Detection. In this section, we conduct experiments for the first research question. Table 2 shows the leak detection results under the 3 set-associative cache settings. Let's name these caches as C_1, C_2, and C_3. C_1 has a 32KB size, and each cache set consists of 4 lines. As each line has 64 bytes, C_1 has 128 sets in total. C_2 also owns 32KB size, but its associativity increases to 8. Hence it has 64 sets. C_3 has 64KB size while its associativity remains 8, thereby it has 128 sets and 1024 cache lines. For each benchmark, we collect execution statistics under all 3 cache settings, including the total execution time in minute (Time (m)) and the amount of divergent/opposite leaks, named #.D/O.

Overall, SPECUSYM identified that 6 out of 15 benchmarks suffer from timing leaks. Specifically, blowfish is leaky only under cache C_2 while the other 5 programs have leaks under all 3 caches. Among

Table 2: Detection results on set-associative caches

Name	C_1: 32KB, 4-Way		C_2: 32KB, 8-Way		C_3: 64KB, 8-Way	
	Time (m)	#.D/O	Time (m)	#.D/O	Time (m)	#.D/O
hash[2]	0.57	0/0	0.56	0/0	0.56	0/0
AES[4]	32.53	20/3	99.43	25/0	65.27	26/0
blowfish[4]	0.04	0/0	0.03	0/2	0.03	0/0
chacha20[4]	0.31	0/5	0.31	0/61	0.28	0/61
encoder[4]	< 0.01	0/5	< 0.01	0/5	< 0.01	0/5
ocb[4]	< 0.01	0/0	< 0.01	0/0	< 0.01	0/0
DES[5]	10.72	0/5	10.65	0/10	10.77	0/10
str2key[5]	0.01	0/0	< 0.01	0/0	< 0.01	0/0
DES[1]	51.98	16/6	384.57	26/1	393.48	24/3
Camellia[7]	0.13	0/0	0.12	0/0	0.13	0/0
Salsa[7]	0.03	0/0	0.03	0/0	0.04	0/0
Seed[7]	0.21	0/0	0.21	0/0	0.21	0/0
DES[3]	0.37	0/0	0.37	0/0	0.37	0/0
Salsa[3]	< 0.01	0/0	< 0.01	0/0	< 0.01	0/0
Spectre[6]	<0.01	0/0	< 0.01	0/0	< 0.01	0/0

these 5 programs, DES [1] has both D/O leaks in all caches; AES has both D/O leaks in C_1, but only opposite leaks in C_2 and C_3; and the rest 3 programs (chacha20, encoder, and DES [5]) have only opposite leaks under all the 3 caches.

Next, we compare the results under C_1 and C_2 to research how the cache associativity variation affects leak occurrence. Intuitively, the more lines per cache set, the less potential cache conflicts hence fewer leaks. However, experiments affirm that the increased associativity indeed triggers more leaks. First, SPECUSYM detected 60 and 130 leaks under C_1 and C_2, respectively; second, 5 out of the 6 leaky programs have more leaks under C_2. The only exception, encoder, has 5 leaks in both C_1 and C_2. Third, the non-leaky program under C_1, blowfish, now turns to expose 2 new leaks under C_2. This phenomenon is mostly because the speculative memory visits raise more cache hits — the misses under non-speculative execution now have higher possibilities to be cache hits since speculative execution has fetched memory data earlier into the cache.

Further, we compare between C_2 and C_3, to study how the cache size could affect leak detection. We find that despite the sizeable 32KB capacity difference, the detection results show no drastic variances. First, SPECUSYM detected the same types and amounts of leaks in 3 programs (chacha20, encoder, and DES [5]) in both C_2 and C_3. The analysis time is also close. Second, another two leaky programs, AES and DES [1], also exhibit similar statistics. Third, the main difference comes from blowfish, which has two leaks on C_2 but no leaks on C_3. Though this unique case indicates that smaller cache might incur more leaks, it is not the majority situation.

Besides, we observe drastically varying results from different versions of the same algorithm. For example, in three DES implementations from OpenSSL [5], glibc [1], and Libgcrpt [3], SPECUSYM detected only opposite leaks in OpenSSL DES, both D/O leaks in glibc DES, and no leaks in Libgcrypt DES. Meanwhile, OpenSSL DES needs about 11 minutes of analysis time upon all caches. The glibc DES uses 52 minutes for C_1 but around 6.5 hours in both C_2 and C_3. By contrast, Libgcrypt DES uses merely 1 minute upon all 3 caches.

Therefore, we can answer the first research question: SPECUSYM is capable of detecting timing leaks introduced by speculative execution. It also supports various caches for comparative analysis.

5.2.2 Existing Method Comparison. We compare SPECUSYM with [78] in Table 3, to answer the second research question. Column 1

Table 3: Comparison betwee Wu et al. [78] and SPECUSYM

| Name	Wu et al. [78]		SpecuSym		
	Leak Detected	Time (s)	Leak Detected	#.Leaks	Time (s)
hash[2]	✔	1.15	✗	0	33.61
AES[4]	✗	2.13	✗	0	5.01
chacha20[4]	✔	9.24	✔	9	17.84
encoder[4]	✔	0.10	✔	5	0.13
ocb[4]	✔	0.68	✗	0	0.05
DES[5]	✔	14.20	✔	8	725.66
str2key[5]	✗	0.01	✗	0	0.69
Camellia[7]	✗	6.35	✗	0	7.59
Salsa[7]	✗	0.06	✗	0	0.99
Seed[7]	✗	0.07	✗	0	12.15

of Table 3 lists all the used benchmarks in [78] for timing leak evaluation. Columns 2-3 show the detection result and the computation time of this abstract interpretation based method [78]. Columns 4-6 depict the result of SPECUSYM, while column 5 lists the total amount of detected leaks. To ensure a fair comparison, we configure a 512-line fully-associative cache with LRU policy, which equals the cache setting used in [78]. Let's name it as C_4.

Overall, [78] reported that 5 of the 10 benchmarks are leaky, as marked with ✔ in column 2. Consequently, the rest 5 programs are free of timing leaks. For these robust programs, SPECUSYM also detects no leaks, as shown in column 4. The result means SPECUSYM introduces no false positives on analyzing these programs. In contrast, for those ✔ results from [78], SPECUSYM provides different results on two benchmarks hash and ocb, as underlined in Table 3. They are deemed to be leaky in [78], while SPECUSYM found no leaks in them. We manually inspect the two programs and confirm that SPECUSYM gives the right answer — no leaks exist in these programs. The inaccurate results of [78] should root from its over-approximation solution, and they are indeed false positives.

For the remaining programs, chacha20, encoder, and DES, both SPECUSYM and [78] have detected leaks. However, [78] only reports property violations, whereas SPECUSYM generates rich information. For example, SPECUSYM pinpoints 9, 5, and 8 leaky sites in these benchmarks, respectively. Moreover, SPECUSYM can provide inputs and memory event trace for leak diagnosis. Note that SPECUSYM generally performs faster on C_4 than on other three set-associative cachess since the extreme cache setting of C_4 advances both cache utilization and cache hit rate for these benchmarks.

Compared to [78], SPECUSYM may have false negatives because of two reasons. First, the bounded speculative modeling might miss certain speculation cases. Second, SPECUSYM stops speculation once it finds that an event on the speculative path must cause a cache miss; thus, it skips executing following events on that path. Nevertheless, SPECUSYM provides more precise results because of the high-fidelity modeling and fine-grained per-event reasoning.

Another drawback of SPECUSYM is the computation overhead. Table 3 shows that [78] finishes all benchmarks in 35 seconds, whereas SPECUSYM needs 725.7 seconds to analyze an OpenSSL DES. Worse, the analysis time under C_3 for the glibc DES [1] even rises to about 6.6 hours (cf. Table 2). The cost mainly comes from constraint solver since SPECUSYM reasons individual memory accesses on each program path. However, sacrificing the performance for the precision achieves reasonable paybacks. First, the increased overhead of most benchmarks is still tolerable. Next, SPECUSYM finds false positives from [78] and generates precise inputs. Moreover, if [78]

(a) The benchmark analysis time

(b) The SMT constraint size
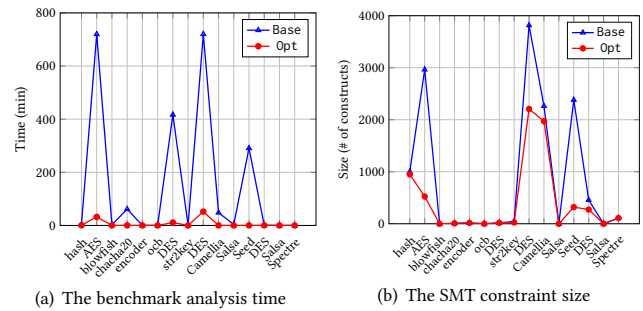
Figure 4: Performance comparison for the individual program. Benchmarks are listed by their orders in Table 1.

Table 4: The performance boost due to optimizations.

Version	Time	Constraints Size	#.Div	#.Opp
Base	100%	100%	12	76
Opt	30.7%	78.9%	50	79

provides assisting information, e.g., the suspicious locations on the control flow graph, SPECUSYM can purposely guide its analysis to the problematic area at an earlier stage.

Now, we answer the second research question: SPECUSYM can complement the latest abstract interpretation based analysis [78]. It introduces no false negatives and identifies two false positives reported in [78]. Further, it can provide inputs, the speculation flows, and the leak locations for detailed diagnosis.

5.2.3 Optimization Performance. This section studies the performance of the optimizing strategies proposed in Section 4.4. Table 4 compares the overall performance between the baseline SPECUSYM without any optimizations (Base), and the optimized SPECUSYM with all optimizations enabled (Opt), as listed in column 1. Column 2 and column 3 compare the analysis time and the size of involved constraints between Base and Opt. Column 4 and column 5 disclose the number of identified new divergent leaks (#.**Div**) and new opposite leaks (#.**Opp**), respectively. The cache setting for this experiment is C_3, under which Base and Opt have the minimum performance gap, forming a conservative comparison result.

As shown in column 2, after optimizations, the analysis time of the Opt SPECUSYM significantly decreases, falling to only 30.7% of the analysis time in Base, which confirms the effectiveness of the optimizations in accelerating the analyses.

Apart from the time drop, the Opt SPECUSYM reduces the size of the involved constraint by more than 20%, compared to the Base version, as shown in column 3. Though the reduction to constraint sizes looks not as impressive as the time reduction, the proposed strategies greatly ease the solver burden by constraint transformation and formula reduction. Undoubtedly, constraint size reduction is the primary reason for the time decrease.

Moreover, column 4 and column 5 disclose that the Opt SPECUSYM detects 31.8% more leaks in total than the Base version, within a 12-hour threshold. In particular, Opt identifies 3 times more divergent leaks than the Base version, which highlights the importance of our proposed optimization strategies.

We also examine how the optimizations impact each benchmark in terms of the analysis time in Figure 4(a) and the constraint size in

Figure 4(b). Programs in these two figures are arranged using their orders in Table 1. Tags Base and Opt also stand for the SᴘᴇᴄᴜSʏᴍ versions without and with optimizations, respectively.

Clearly, Figure 4(a) shows that optimizations drastically reduce time cost for 6 out of 15 cases from hundreds of minutes to less than an hour. On the one hand, the Base has more repeated computations while Opt reuses cached intermediate results. On the other hand, formula reduction lowers the cost of solving. Figure 4(b) shows how optimizations reduce the constraint size where AES [4], glibc DES [1], and seed [7] have significant improvements. For the other cases, the memory trace length is commonly short. Thus, our optimizations have no apparent improvements upon them.

Finally, we answer the last research question: our optimizations significantly boost the overall performance by saving 69.3% execution time and detecting 31.8% more timing leaks.

5.3 Threat to Validity

SᴘᴇᴄᴜSʏᴍ works on the LLVM IR. The actual program behavior may differ due to the compiler optimizations and hardware features, which could introduce false positives to SᴘᴇᴄᴜSʏᴍ. More low-level details can further facilitate SᴘᴇᴄᴜSʏᴍ to improve the analysis.

SᴘᴇᴄᴜSʏᴍ considers nested speculative execution since branches may also exist on speculated paths. As the speculative window is usually transient, we treat the nested depth as configurable and set it to *two* in evaluation. Also, SᴘᴇᴄᴜSʏᴍ executes at most the reorder buffer size of instructions on a speculative path. However, experiments show the effectiveness of this approximation.

So far, SᴘᴇᴄᴜSʏᴍ cannot support the analysis of multithreaded programs. Though SʏᴍSC [38] leveraged multithreading for timing leak detection, considering concurrency and speculative execution together greatly amplifies the state space and interleaving space, which pose exceptional challenges for a precise analysis.

Another threat is from the out-of-order execution, which schedules a set of data-independent instructions in an out-of-order manner, rather than sticking to the program order. This processor optimization may affect the accuracy of our analysis. However, incorporating both speculative execution and out-of-order execution requires more dedicated analysis. We leave it as future work.

6 RELATED WORK

Regarding the security impact of processor speculative execution [44], Kocher et al. [46] demonstrated that speculative execution could silently influence the processor cache state, thus giving rise to cache timing attacks, which motivates our SᴘᴇᴄᴜSʏᴍ work.

Symbolic execution [45] has been applying to side-channel analysis because of the original designs of precise reasoning and input generation [9, 10, 13, 61, 63]. To quantify the information leakage through a cache timing side-channel, Chattopadhyay et al. [18, 19] and Basu et al. [12] developed symbolic execution methods to study dependencies between sensitive data and cache behaviors. Wang et al. developed CacheD [74], which targets timing leaks through a trace-based symbolic execution. Furthermore, Brotzman et al. [14] proposed CaSym, a symbolic reasoning method that supports cache analysis upon multiple program paths. Considering the cache effects from thread interleavings, Guo et al. [38] proposed SʏᴍSC to analyze cache timing leaks due to multithreading. These methods, however, all ignore cache changes due to speculative execution.

Abstract interpretation [25] has also been adopting to cache timing side-channel analysis because of its scalable analysis and sound

approximation. CacheAudit [28] approximates cache states by concrete state abstraction. Based on CacheAudit, Barthe et al. [11] track the program cache state affected by the attacker from a separate process. CacheS [73] provides a secret-augmented abstract domain to track program secrets and dependencies. However, CacheAudit and CacheS are unaware of the impact of speculative execution. Though Wu et al. [78] developed a dedicated abstract interpretation method for cache timing leak analysis under speculative execution, the proposed technique is incapable of generating precise speculative flow and inputs to diagnose leaks.

Likewise, grey-box fuzz testing has been used in detecting program undefined behaviors [23], Spectre-type vulnerabilities [58], exposing side channels [57], and revealing timing leaks [39]. However, they are either unaware of speculative execution or relying on sophisticated instrumentations on the control flow. Also, due to the fuzzing nature, these methods primarily count on various heuristics while SᴘᴇᴄᴜSʏᴍ systematically solves for precise inputs.

Among the existing works, KLEESᴘᴇᴄᴛʀᴇ [71], Sᴘᴇᴄᴛᴇᴄᴛᴏʀ [35], and [82] are closely related to SᴘᴇᴄᴜSʏᴍ. KLEESᴘᴇᴄᴛʀᴇ [71] has an analogous understanding of modeling speculation in symbolic execution. However, KLEESᴘᴇᴄᴛʀᴇ [71] aims at Spectre-style vulnerability; thus, it has a crucial difference in constraint semantics. Sᴘᴇᴄᴛᴇᴄᴛᴏʀ [71] also targets Spectre-type vulnerability. It employs symbolic execution to prove speculative non-interference property or detect violations. Speculative symbolic execution [82] proposed a method to lazily reason branch feasibility until reaching a threshold, which improves the efficiency of the symbolic execution technique.

Besides the above analysis methods, there also exists compiler assisted techniques [29, 68, 70, 72, 77], formal verifications [8, 20, 32, 33, 67, 81], and program synthesis approaches [31, 34, 53] that model or mitigate side-channel leaks. Still, none of them is applicable to cache timing side channels due to speculative execution.

To conclude, existing works may fail to produce inputs, ignore cache affects from speculative execution or leverages speculation for other analyses. In contrast, SᴘᴇᴄᴜSʏᴍ models speculative execution through stateful symbolic execution to generate inputs and pinpoint leak sites, which can help reason the cache timing side-channel leaks caused by speculative execution in depth.

7 CONCLUSIONS

In this paper, we have presented a symbolic execution based method SᴘᴇᴄᴜSʏᴍ for detecting cache timing leaks on running a sensitive data related program under speculative execution. SᴘᴇᴄᴜSʏᴍ systematically explores the program state space and models speculative execution at conditional branches. By cache state modeling as well as cache behavior analysis, SᴘᴇᴄᴜSʏᴍ leverages constraint-solving to search for divergent and opposite cache behaviors at memory access events. Experimental results show that SᴘᴇᴄᴜSʏᴍ successfully detects timing leaks under different cache settings. The proposed optimizations help SᴘᴇᴄᴜSʏᴍ save 69.3% execution time to detect 31.8% more leaks. Comparison between SᴘᴇᴄᴜSʏᴍ and state-of-the-art abstract interpretation based method [78] shows that SᴘᴇᴄᴜSʏᴍ not only successfully detects leaks in the reported programs but also eliminates false positives for a more precise result.

ACKNOWLEDGEMENT

REFERENCES

[1] *glibc-2.29.9000.* https://www.gnu.org/software/libc/.
[2] *High Performance SSH/SCP - HPN-SSH.* https://www.psc.edu/hpn-ssh.
[3] *Libgcrypt-1.8.4.* https://gnupg.org/software/libgcrypt/index.html.
[4] *LibTomCrypt.* http://www.libtom.net/LibTomCrypt/.
[5] *OpenSSL-1.1.1c.* https://mta.openssl.org/pipermail/openssl-announce/2019-May/000153.html.
[6] *spectre-attach.* https://github.com/Eugnis/spectre-attack.
[7] *Tegra.* https://android.googlesource.com/kernel/tegra/+/android-8.1.0_r0.113/crypto.
[8] Timos Antonopoulos, Paul Gazzillo, Michael Hicks, Eric Koskinen, Tachio Terauchi, and Shiyi Wei. 2017. 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, PLDI 2017, Barcelona, Spain, June 18-23, 2017.* 362–375.
[9] Lucas Bang, Abdulbaki Aydin, Quoc-Sang Phan, Corina S. Pasareanu, and Tevfik Bultan. 2016. String analysis for side channels with segmented oracles. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016.* 193–204.
[10] Lucas Bang, Nicolás Rosner, and Tevfik Bultan. 2018. Online Synthesis of Adaptive Side-Channel Attacks Based On Noisy Observations. In *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018.* 307–322.
[11] Gilles Barthe, Boris Köpf, Laurent Mauborgne, and Martín Ochoa. 2014. Leakage Resilience against Concurrent Cache Attacks. In *Principles of Security and Trust - Third International Conference, POST 2014.* 140–158.
[12] Tiyash Basu and Sudipta Chattopadhyay. 2017. Testing Cache Side-Channel Leakage. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2017, Tokyo, Japan, March 13-17, 2017.* 51–60.
[13] Tegan Brennan, Seemanta Saha, and Tevfik Bultan. 2018. Symbolic path cost analysis for side-channel detection. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018.* 424–425.
[14] Robert L. Brotzman, Shen Liu, Danfeng Zhang, Gang Tan, and Mahmut T. Kandemir. 2019. CaSym: Cache Aware Symbolic Execution for Side Channel Detection and Mitigation. In *IEEE Symposium on Security and Privacy*.
[15] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018.* 991–1008.
[16] Ashokkumar C., Ravi Prakash Giri, and Bernard L. Menezes. 2016. Highly Efficient Algorithms for AES Key Retrieval in Cache Access Attacks. In *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016.* 261–275.
[17] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings.* 209–224.
[18] Sudipta Chattopadhyay. 2017. Directed Automated Memory Performance Testing. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS.* 38–55.
[19] Sudipta Chattopadhyay, Moritz Beck, Ahmed Rezine, and Andreas Zeller. 2017. Quantifying the information leak in cache attacks via symbolic execution. In *Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE 2017, Vienna, Austria, September 29 - October 02, 2017.* 25–35.
[20] Jia Chen, Yu Feng, and Isil Dillig. 2017. Precise Detection of Side-Channel Vulnerabilities using Quantitative Cartesian Hoare Logic. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017.* 875–890.
[21] Qichang Chen, Liqiang Wang, Zijiang Yang, and Scott D. Stoller. 2009. HAVE: Detecting Atomicity Violations via Integrated Dynamic and Static Analysis. In *Fundamental Approaches to Software Engineering, 12th International Conference, FASE 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings.* 425–439.
[22] Tong Chen, Jin Lin, Xiaoru Dai, Wei-Chung Hsu, and Pen-Chung Yew. 2004. Data Dependence Profiling for Speculative Optimizations. In *Compiler Construction, 13th International Conference, CC 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings.* 57–72.
[23] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. 2019. SAVIOR: Towards Bug-Driven Hybrid Testing. *CoRR* abs/1906.07327 (2019).
[24] Lori A. Clarke. 1976. A program testing system. In *Proceedings of the 1976 Annual Conference, Houston, Texas, USA, October 20-22, 1976.* 488–491.
[25] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977.* 238–252.
[26] Jean-François Dhem, François Koeune, Philippe-Alexandre Leroux, Patrick Mestré, Jean-Jacques Quisquater, and Jean-Louis Willems. 1998. A Practical Implementation of the Timing Attack. In *Smart Card Research and Applications, This International Conference, CARDIS '98, Louvain-la-Neuve, Belgium, September 14-16, 1998, Proceedings.* 167–182.
[27] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean M. Tullsen. 2017. Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017.* 51–67.
[28] Goran Doychev, Dominik Feld, Boris Köpf, Laurent Mauborgne, and Jan Reineke. 2013. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. In *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013.* 431–446.
[29] Goran Doychev and Boris Köpf. 2017. Rigorous analysis of software countermeasures against cache attacks. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017.* 406–421.
[30] Vasisht Duddu, Debasis Samanta, D. Vijay Rao, and Valentina E. Balas. 2018. Stealing Neural Networks via Timing Side Channels. *CoRR* abs/1812.11720 (2018). arXiv:1812.11720 http://arxiv.org/abs/1812.11720
[31] Hassan Eldib and Chao Wang. 2014. Synthesis of Masking Countermeasures against Side Channel Attacks. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings.* 114–130.
[32] Hassan Eldib, Chao Wang, and Patrick Schaumont. 2014. SMT-Based Verification of Software Countermeasures against Side-Channel Attacks. In *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings.* 62–77.
[33] Hassan Eldib, Chao Wang, Mostafa M. I. Taha, and Patrick Schaumont. 2014. QMS: Evaluating the Side-Channel Resistance of Masked Software from Source Code. In *The 51st Annual Design Automation Conference 2014, DAC '14, San Francisco, CA, USA, June 1-5, 2014.* 209:1–209:6.
[34] Hassan Eldib, Meng Wu, and Chao Wang. 2016. Synthesis of Fault-Attack Countermeasures for Cryptographic Circuits. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II.* 343–363.
[35] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. 2018. Spectector: Principled Detection of Speculative Information Flows. *CoRR* abs/1812.08639 (2018). arXiv:1812.08639 http://arxiv.org/abs/1812.08639
[36] David Gullasch, Endre Bangerter, and Stephan Krenn. 2011. Cache Games - Bringing Access-Based Cache Attacks on AES to Practice. In *32nd IEEE Symposium on Security and Privacy, S&P 2011, 22-25 May 2011, Berkeley, California, USA.* 490–505.
[37] Shengjian Guo, Markus Kusano, Chao Wang, Zijiang Yang, and Aarti Gupta. 2015. Assertion guided symbolic execution of multithreaded programs. In *ACM SIGSOFT Symposium on Foundations of Software Engineering.* 854–865.
[38] Shengjian Guo, Meng Wu, and Chao Wang. 2018. Adversarial symbolic execution for detecting concurrency-related cache timing leaks. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018.* 377–388.
[39] Shaobo He, Michael Emmi, and Gabriela F. Ciocarlie. 2019. ct-fuzz: Fuzzing for Timing Leaks. *CoRR* abs/1904.07280 (2019).
[40] Sanghyun Hong, Michael Davinroy, Yigitcan Kaya, Stuart Nevans Locke, Ian Rackow, Kevin Kulda, Dana Dachman-Soled, and Tudor Dumitras. 2018. Security Analysis of Deep Neural Networks Operating in the Presence of Cache Side-Channel Attacks. *CoRR* abs/1810.03487 (2018). arXiv:1810.03487 http://arxiv.org/abs/1810.03487
[41] Xing Hu, Ling Liang, Lei Deng, Shuangchen Li, Xinfeng Xie, Yu Ji, Yufei Ding, Chang Liu, Timothy Sherwood, and Yuan Xie. 2019. Neural Network Model Extraction Attacks in Edge Devices by Hearing Architectural Hints. *CoRR* abs/1903.03916 (2019). arXiv:1903.03916 http://arxiv.org/abs/1903.03916
[42] Ralf Hund, Carsten Willems, and Thorsten Holz. 2013. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013.* 191–205.
[43] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gülmezoglu, Thomas Eisenbarth, and Berk Sunar. 2019. SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks. *CoRR* abs/1903.00446 (2019). arXiv:1903.00446 http://arxiv.org/abs/1903.00446
[44] Kozo Kimura, Kosuki Yoshioka, and Tokuzo Kiyohara. 1996. Speculative execution processor. (April 23 1996). US Patent 5,511,172.
[45] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394.
[46] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.
[47] Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology - CRYPTO '96, 16th Annual*

International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings. 104–113.

[48] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *2nd IEEE/ACM International Symposium on Code Generation and Optimization, 20-24 March 2004, San Jose, CA, USA.* 75–88.

[49] Tong Li, Carla Schlatter Ellis, Alvin R. Lebeck, and Daniel J. Sorin. 2005. Pulse: A Dynamic Deadlock Detection Mechanism Using Speculative Execution. In *Proceedings of the 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA.* 31–44.

[50] Xianfeng Li, Tulika Mitra, and Abhik Roychoudhury. 2003. Accurate timing analysis by modeling caches, speculation and their interaction. In *Proceedings of the 40th Design Automation Conference, DAC 2003, Anaheim, CA, USA, June 2-6, 2003.* 466–471.

[51] Xianfeng Li, Tulika Mitra, and Abhik Roychoudhury. 2005. Modeling Control Speculation for Timing Analysis. *Real-Time Systems* 29, 1 (2005), 27–58.

[52] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018.* 973–990.

[53] Matteo Maffei and Mark Ryan (Eds.). 2017. *Principles of Security and Trust - 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings.* Lecture Notes in Computer Science, Vol. 10204. Springer.

[54] Sparsh Mittal. 2019. A survey of techniques for dynamic branch prediction. *Concurrency and Computation: Practice and Experience* 31, 1 (2019).

[55] Andreas Moshovos and Gurindar S. Sohi. 1997. Streamlining Inter-Operation Memory Communication via Data Dependence Prediction. In *Proceedings of the Thirtieth Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 30, Research Triangle Park, North Carolina, USA, December 1-3, 1997.* 235–245.

[56] Alexandru Nicolau. 1989. Run-Time Disambiguation: Coping with Statically Unpredictable Dependencies. *IEEE Trans. Computers* 38, 5 (1989), 663–678.

[57] Shirin Nilizadeh, Yannic Noller, and Corina S. Pasareanu. 2019. DiffFuzz: differential fuzzing for side-channel analysis. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019.* 176–187.

[58] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. 2019. SpecFuzz: Bringing Spectre-type vulnerabilities to the surface. *CoRR* abs/1905.10311 (2019). arXiv:1905.10311 http://arxiv.org/abs/1905.10311

[59] Soner Önder and Rajiv Gupta. 1999. Dynamic Memory Disambiguation in the Presence of Out-of-Order Store Issuing. In *Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 32, Haifa, Israel, November 16-18, 1999.* 170–176.

[60] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2006, Proceedings.* 1–20.

[61] Corina S. Pasareanu, Quoc-Sang Phan, and Pasquale Malacaria. 2016. Multi-run Side-Channel Analysis Using Symbolic Execution and Max-SMT. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016.* 387–400.

[62] Corina S. Pasareanu and Neha Rungta. 2010. Symbolic PathFinder: symbolic execution of Java bytecode. In *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010.* 179–180.

[63] Quoc-Sang Phan, Lucas Bang, Corina S. Pasareanu, Pasquale Malacaria, and Tevfik Bultan. 2017. Synthesis of Adaptive Side-Channel Attacks. *IACR Cryptology ePrint Archive* 2017 (2017), 401.

[64] Prakash Prabhu, Ganesan Ramalingam, and Kapil Vaswani. 2010. Safe programmable speculative parallelism. In *PLDI.* 50–61.

[65] C. V. Ramamoorthy and Hon Fung Li. 1977. Pipeline Architecture. *ACM Comput. Surv.* 9, 1 (1977), 61–102.

[66] Glenn Reinman and Brad Calder. 1998. Predictive Techniques for Aggressive Load Speculation. In *Proceedings of the 31st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 31, Dallas, Texas, USA, November 30 - December 2, 1998.* 127–137.

[67] Marcelo Sousa and Isil Dillig. 2016. Cartesian hoare logic for verifying k-safety properties. In *ACM SIGPLAN Conference on Programming Language Design and Implementation.* 57–69. DOI : http://dx.doi.org/10.1145/2908080.2908092

[68] Chungha Sung, Brandon Paulsen, and Chao Wang. 2018. CANAL: a cache timing analysis framework via LLVM transformation. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018.* 904–907.

[69] Eran Tromer, Dag Arne Osvik, and Adi Shamir. 2010. Efficient Cache Attacks on AES, and Countermeasures. *J. Cryptology* 23, 1 (2010), 37–71.

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

[71] Guanhua Wang, Sudipta Chattopadhyay, Arnab Kumar Biswas, Tulika Mitra, and Abhik Roychoudhury. 2019. KLEESPECTRE: Detecting Information Leakage

[72] Jingbo Wang, Chungha Sung, and Chao Wang. 2019. Mitigating power side channels during compilation. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019.* 590–601.

[73] Shuai Wang, Yuyan Bao, Xiao Liu, Pei Wang, Danfeng Zhang, and Dinghao Wu. 2019. Identifying Cache-Based Side Channels through Secret-Augmented Abstract Interpretation. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019.* 657–674.

[74] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. 2017. CacheD: Identifying Cache-Based Timing Channels in Production Software. In *26th USENIX Security Symposium (USENIX Security 17).* 235–252.

[75] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. 2018. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution. *Technical report* (2018).

[76] Jan Wichelmann, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. 2018. MicroWalk: A Framework for Finding Side Channels in Binaries. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018.* 161–173.

[77] Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. 2018. Eliminating timing side-channel leaks using program repair. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018.* 15–26.

[78] Meng Wu and Chao Wang. 2019. Abstract Interpretation under Speculative Execution. In *ACM SIGPLAN Conference on Programming Language Design and Implementation.* 57–69.

[79] Mengjia Yan, Christopher W. Fletcher, and Josep Torrellas. 2018. Cache Telepathy: Leveraging Shared Resource Attacks to Learn DNN Architectures. *CoRR* abs/1808.04761 (2018). arXiv:1808.04761 http://arxiv.org/abs/1808.04761

[80] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.* 719–732.

[81] Jun Zhang, Pengfei Gao, Fu Song, and Chao Wang. 2018. SCInfer: Refinement-Based Verification of Software Countermeasures Against Side-Channel Attacks. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II.* 157–177.

[82] Yufeng Zhang, Zhenbang Chen, and Ji Wang. 2012. Speculative Symbolic Execution. In *23rd IEEE International Symposium on Software Reliability Engineering, ISSRE 2012, Dallas, TX, USA, November 27-30, 2012.* 101–110.