Exposing Cache Timing Side-Channel Leaks through Out-of-Order Symbolic Execution

SHENGJIAN GUO*, Baidu Security, USA
YUEQI CHEN*, Pennsylvania State University, USA
JIYONG YU, University of Illinois at Urbana-Champaign, USA
MENG WU, Ant Group, China
ZHIQIANG ZUO, State Key Lab. for Novel Software Technology, Nanjing University, China
PENG LI, Baidu Security, USA
YUEQIANG CHENG, Baidu Security, USA
HUIBO WANG, Baidu Security, USA

As one of the fundamental optimizations in modern processors, the out-of-order execution boosts the pipeline throughput by executing independent instructions in parallel rather than in their program orders. However, due to the side effects introduced by such microarchitectural optimization to the CPU cache, secret-critical applications may suffer from timing side-channel leaks. This paper presents a symbolic execution-based technique, named SymO₃, for exposing cache timing leaks under the context of out-of-order execution. SymO₃ proposes new components that address the modeling, reduction, and reasoning challenges of accommodating program analysis to the software code out-of-order analysis. We implemented SymO₃ upon KLEE and conducted three evaluations on it. Experimental results show that SymO₃ successfully uncovers a set of cache timing leaks in five real-world programs. Also, SymO₃ finds that, in general, program transformation from compiler optimizations shrink the surface to timing leaks. Furthermore, augmented with a speculative execution modeling, SymO₃ identifies five more leaky programs based on the compound analysis.

CCS Concepts: • Security and privacy → Software and application security; • Software and its engineering → Software verification and validation.

Additional Key Words and Phrases: Out-of-order execution, cache timing, side-channel leak, symbolic execution

1 INTRODUCTION

In modern processor chips, CPU cache alleviates the outstanding speed disparity between the processor and the main memory. Generally, processors buffer the recently used memory data in

*These authors contributed equally.

This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2020 Copyright held by the owner/author(s).
2475-1421/2020/11-ART147
https://doi.org/10.1145/3428215

the hierarchical caches for the quick future reuse, thus significantly avoiding latency from visiting memory for the same data. Despite the tremendous importance, CPU cache is prone to cache timing side-channel attacks due to the physically distinguishable timing differences between visiting cache and the memory [Dhem et al. 1998; Gras et al. 2017; Gruss et al. 2016; Kocher 1996; Oren et al. 2015]. Based on the runtime timing statistics of the victims, external adversaries may infer victims’ confidential data to certain extents, by exploring the dependency between the confidentiality and the timing characteristics. Practical cache timing attacks [Aldaya et al. 2019; Kocher et al. 2019; Lipp et al. 2018] have been continuously threatening the cybersecurity of computer systems.

The straightforward way to prevent software from timing attacks is to precisely examine the program’s cache behavior, thus uncovering the weak sites for leakage mitigation. A broad spectrum of studies have investigated the detection of cache timing side-channel leaks, spanning from static analysis [Doychev et al. 2013; Wang et al. 2019], through symbolic execution [Chattopadhyay et al. 2017; Chu et al. 2016; Wang et al. 2017], to dynamic analysis and testing [Basu et al. 2020; He et al. 2019; Nilizadeh et al. 2019; Xiao et al. 2017].

Nevertheless, the program runtime cache behavior is not only determined by its computation logic but also can be implicitly affected by the microarchitectural factors of hardware. The out-of-order execution [Smith and Pleszkun 1985], one of the fundamental optimizations in modern pipelined processors, boosts the pipeline throughput by scheduling proper instructions out-of-order rather than in the program order generated by compilers. Although out-of-order execution has been designed to be transparent to the programs running atop, it indeed dramatically affects the cache state during program execution.

Recent studies [Bulck et al. 2018; Lipp et al. 2018; Weisse et al. 2018] have revealed that due to the side effects introduced by the out-of-order execution to the CPU cache state, sensitive data may leak to external adversaries by deliberately magnifying the effects through cache timing side-channel. To alleviate the leaking risk in secret-critical applications, a systematic leak detection technique for out-of-order execution is highly desirable.

The existing program analysis based leak detection approaches mostly focus on the in-order analysis for both sequential software [Chattopadhyay et al. 2017; Coppens et al. 2009; Doychev et al. 2013; Wang et al. 2017, 2019] and concurrent programs [Barthe et al. 2014; Guo et al. 2018]. Recent works also tried modeling the non-functional speculative execution based on abstract interpretation [Wu and Wang 2019], fuzz testing [Oleksenko et al. 2020], and symbolic execution [Guarnieri et al. 2020; Guo et al. 2020; Wang et al. 2020]. Nevertheless, none of them are capable of analyzing the effects of out-of-order execution on the program cache state. On the one hand, the in-order methods assume that instruction executions must obey the program order, thus missing the opportunities of reasoning the respective out-of-order side effects. On the other hand, the speculative modeling approaches only re-order a bundle of instructions from predicted branches. At the same time, instruction-level out-of-order execution [Smith and Pleszkun 1985] may non-deterministically occur everywhere in program execution.

Achieving the goal of accommodating program analysis to out-of-order execution has to overcome three significant challenges. The first challenge is about representing the complicated microarchitectural out-of-order scenarios in high fidelity, but, inside a user-space analysis software. The second is to enforce the restrictions from both hardware architecture and software semantics in the analysis, to ensure the feasibility of modeled out-of-order behaviors. The last is to systematically identify a set of harmful executions, from the vast out-of-order search space, that may unveil the violations of specific requirements, e.g., the cache timing leakage-free property; while eliminating a large portion of redundant executions. The blend of these challenges demands new ingredients to program analysis, for studying this emerging spectrum of threats that intensify software flaws in the low-level hardware environment.
Towards this end, we propose SymO₃, a symbolic execution-based technique for exposing the cache timing leaks coupled with out-of-order execution. To address the above challenges, we developed innovative solutions in SymO₃, as shown by the four components in the dotted area in Fig. 1. First, we break the complicated whole-path out-of-order situations into the essential two-event reorder modeling, which can be readily integrated into a stateful symbolic execution to produce out-of-order execution of two specific memory instructions. Second, we abstract the hardware and software restrictions into a set of enforceable rules and embed them into the dynamic symbolic executor, to guarantee the correctness of modeled out-of-order behavior. Third, we design a new systematic out-of-order generation method, which seamlessly incorporates the above two components to select suspicious memory event pairs that may cause leaks from the huge out-of-order state space. Finally, we accommodate the constraint-solving based leak analysis into the out-of-order scenarios to reason the existence of timing leaks in our SymO₃ framework. To the best of our knowledge, SymO₃ is the first work that tailors symbolic execution for cache timing leakage detection under the context of out-of-order execution.

Fig. 1 presents the overall flow of SymO₃. Given a program P, which is leakage-free in in-order execution, the sensitive data input, and the non-sensitive plain input, SymO₃ symbolizes the sensitive input and ignites the symbolic execution. During the dynamic exploration, the symbolic executor interacts with the Out-of-Order Generation component which subsequently communicates with the reorder Necessity Analysis and the Reorder Modeling components, to decide the responsive activities on interpreting memory instructions along an execution path. Based on these new constituents, SymO₃ integrates the constraint-solving based leak analysis to perform the timing leak detection under the out-of-order scenarios. Finally, SymO₃ outputs the identified leak cases.

To conclude, we make the following technical contributions:

- The identification and formalization of cache timing side-channel leaks introduced by the CPU out-of-order execution.
- The symbolic execution-based systematic reasoning method, which proposes out-of-order generation, reorder modeling, reorder necessity analysis, and leak analysis for identifying harmful out-of-order behaviors and inputs that lead to timing leaks.
- The implementation and the evaluation, which demonstrate the effectiveness of the proposed SymO₃ technique in leak exposure.

We organize the rest of this paper as follows. Section 2 motivates our work, and section 3 states background knowledge. We present the core techniques of SymO₃ in section 4 and the evaluation in section 5. Section 6 discusses the threats to validity, followed by the related work in section 7. Finally, section 8 concludes this paper.
2 MOTIVATION
This section uses an example to point out the problem. That is, how out-of-order execution can introduce timing leaks into a program, which has no such risks in regular in-order execution.

2.1 The Running Example
Fig. 2(a) shows the program $P$. As listed in lines 2-3, $P$ has six variables $X$, $Y$, $Z$, $i$, $j$, and $k$. Both $X$ and $Y$ store 1-byte unsigned integer; $Z$ is a 255-byte array; $i$, $j$, and $k$ are register variables. We deem that operating variables $X$, $Y$, and $Z$ visits either cache or main memory, while in contrast, accessing register variables $i$, $j$, and $k$ requires no memory or cache visit. Moreover, we treat $X$ as the private data of $P$, and any form of disclosing the value or the range of $X$ from investigating the cache timing information turns to be a side-channel leak.

The load and store operators in $P$ represent memory operations upon the associated operands. For instance, at line 6, the store instruction iteratively writes data to each cell of array $Z$. At line 7, the load instruction loads the value of $X$ to the register variable $j$, which is then used in writing $Z[j]$ at line 8. Note that operating $i$ and $j$ involves register visits only, and we ignore the side effects because of the negligible timing latency.

For brevity, we incorporate a simplified cache $C$ for the cache behavior analysis. $C$ is a fully-associative cache with the least recently used (LRU) replacement policy. This setting means that each local variable in $P$ may associate with arbitrary cache line(s). Moreover, the variable type, the cache availability, and the LRU policy jointly decide the actual cache mappings. We set $C$ to 256 bytes and design each cache line with the exact one-byte capacity. As a result, $C$ has 256 lines in total, and we index them by $\#n$ where $n \in [1, 256]$.

Fig. 2(b) shows how the variables map to cache $C$ when running $P$ in the in-order and out-of-order manner, respectively. Look into the in-order mapping $\psi_{in}$ first. Initially, we have an empty cache $C$. On executing $P$, the first memory write (line 4) associate cache line $\#1$ with $Y$. Next, the for loop repeatedly writes array $Z$ 255 times, mapping all array cells to lines $\#2-\#256$ of $C$. This is because each uint8_t array item uses a whole cache line. Afterward, the load instruction at line 7 reads $X$. Since cache $C$ has been fully filled, this load would first evict $Y$ from the least recently used cache line $\#1$ and then places $X$ into line $\#1$, as annotated by the dashed arrow in Fig. 2(b). Next, the subsequent write to $Z[j]$ must get a cache hit no matter the value of $X$ since the entire array $Z$ remains in $C$. Finally, at line 9, reading $Y$ might evict $Z[0]$ or $Z[1]$ because of three points: $Y$ is no longer in the cache; there is no empty cache line in $C$; the least recently used line could be either line $\#2$ ($j=0$) or $\#3$ ($j=0$) due to the write to $Z[j]$.

This single-path program has a decided cache behavior. The first memory write to $Y$ always triggers a cache cold miss. The following 255 memory writes in the for loop consume the rest space of $C$, and they all cause cache cold misses. The next memory read of $X$ incurs a conflict miss while its succeeding write of $Z[j]$ must get a cache hit, as explained above. Finally, loading $Y$ receives a cache conflict miss. This cache behavior shows $P$ has no timing leaks in the in-order execution — external observers cannot learn the value of $X$ by the constant program execution timing.

2.2 Out-of-Order Execution Brings Timing Leak
As studied above, $P$ has stationary cache timing irrespective of the value of $X$. However, this property could be falsified when taking account of the out-of-order execution.

Let us consider a potential out-of-order execution case. At line 8, the store to $Z[j]$ has to wait until reading $X$ at line 7 finishes so that $j$ is available in the register. These two instructions are data-dependent and have to execute in-order. By contrast, the subsequent load $Y$ accesses a different address. As a result, under the out-of-order execution, once the former two instructions
Exposing Cache Timing Side-Channel Leaks through Out-of-Order Symbolic Execution

---

1 // X: private data
2 uint8_t X,Y,Z[255];
3 reg uint8_t i,j,k;
4 store Y, 128;
5 for(i=0;i<255;i++)
6 store Z[i], 128;
7 j=load X;//0<=X<255
8 store Z[j], 128;
9 k=load Y;

---

(a) The program P

---

get stalled due to the cache miss hazard of reading X, the latter load Y could be scheduled. Indeed, such out-of-order scheduling is likely to happen. Since Y has been in cache C because of the first memory store at line 4, instruction load Y could be ready for execution in a few processor cycles after querying Y from the cache.

The cache mapping $\psi_{out}$ at the bottom of Fig. 2(b) shows this out-of-order case. Again, cache C is initially empty. The first store to Y and then the array iteration, associate cache blocks #1-#256 with Y and Z. After that, following the out-of-order scheduling discussed above, instruction load Y executes and gets a cache hit since Y is still in cache. This hit also updates line #1 to be the most recently used line. Hence, cache line #2 turns to be the least recently used one now. Therefore, when load X (line 6) executes, it first evicts Z[0] from line #2 by X, as annotated by the dotted arrow in Fig. 2(b). Then, writing Z[j] executes, and P terminates.

Switch to the cache behavior analysis. The first 256 memory writes to Y and Z all have cache cold misses. Then the out-of-order execution of load Y results in a cache hit and affects the cache state. The following read of X has to evict Z[0], so it causes a conflict miss. The last memory write of Z[j], however, can lead to two different results: a cache hit if $X \in [1,255)$, or a cache miss if $X=0$ since Z[0] is no longer in the cache.

Thereby, if an adversary observes that running P costs more CPU cycles, she can learn that X has a high potential to be 0. It is because of the unique cache behavior — only X≠0 lets the execution of P have two cache misses, thus distinguishable timing variance — accordingly, out-of-order execution brings a new leak.

2.3 The Out-of-Order Statistics on Running P

It is worth noting that out-of-order execution can take place among various types of instructions. We concentrate on the memory instructions as they directly affect the cache status. Specifically, we only study how the memory read operations can be dynamically issued since the memory write operations are generally allowed to have in-order executions in practice.

Let us denote the processor reorder buffer has $\delta$ entries, which indicates at most $\delta$ consecutive instructions in the program order can enter the reorder buffer at one time, and, may run out-of-order. In other words, if the distance of two reachable instructions, i.e., the number of in-between runtime instructions, is more than $\delta$, then the two instructions cannot simultaneously involve in the out-of-order execution. Here we use $\delta=64$ for the analysis.

The load Y is the last instruction in P. So it may have out-of-order behaviors against at most 49 instructions before it. That is, the beginning of the out-of-order window for load Y is roughly after store Z[231] in the loop. Note that we treat line 5 as an atomic instruction. Similarly, load Y...
Table 1. The cache behaviors of three memory operations under out-of-order execution.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>load X</th>
<th>store Z[j],128</th>
<th>load Y</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>In-order</td>
<td>miss</td>
<td>hit</td>
<td>miss</td>
<td>no-leak</td>
</tr>
<tr>
<td>9→8</td>
<td>miss</td>
<td>hit/miss</td>
<td>miss</td>
<td>leak</td>
</tr>
<tr>
<td>9→7</td>
<td>miss</td>
<td>hit/miss</td>
<td>hit</td>
<td>leak</td>
</tr>
<tr>
<td>9→6</td>
<td>miss</td>
<td>hit/miss</td>
<td>hit</td>
<td>leak</td>
</tr>
<tr>
<td>7→6</td>
<td>miss</td>
<td>hit</td>
<td>miss</td>
<td>no-leak</td>
</tr>
<tr>
<td>7→6→9</td>
<td>miss</td>
<td>hit/miss</td>
<td>miss</td>
<td>leak</td>
</tr>
<tr>
<td>9→6→6→6→9</td>
<td>miss</td>
<td>hit/miss</td>
<td>hit</td>
<td>leak</td>
</tr>
<tr>
<td>7→6→6→9→6</td>
<td>miss</td>
<td>hit/miss</td>
<td>hit</td>
<td>leak</td>
</tr>
</tbody>
</table>

X may also expose out-of-order behaviors against 49 instructions before it since there is no data dependency between load X and its predecessors.

Though program P has only two memory read instructions, the number of distinct out-of-order behaviors based on these two load operations may explode to approximately 49*49=2,401 cases. Moreover, we group these cases into seven equivalent classes w.r.t reading X and Y, and writing Z[j], as shown in Table 1.

Table 1 summarizes the possible out-of-order execution scenarios and the resulting cache behaviors. Operator ↘ means its left-side operand executes before the right-side operand, thus forming an out-of-order situation. For example, 9→8 indicates line 9 executes right before line 8, while 9→6 implies line 9 executes before some store Z[i] instructions in the loop. 7→6→9 shows that line 7 executes before some store Z[i] at line 6, and line 9 runs right after the loop; 7→6→9→6 depicts both line 7 and line 9 runs before some store Z[i], but line 7 has to run before line 9.

In Table 1, not all out-of-order cases are harmful, and 7→6 is the harmless case. However, a high portion (6/7) of out-of-order scenarios could introduce leaks. Interestingly, the appearances of leaks differ. For example, in the case of X=0, only 9→8 and 7→6→9 would have all-miss situations, thus obviously longer timing. Moreover, if X!=0, these two have the same timing to the in-order execution of P, thus no visible leaks. By comparison, if X!=0, the other four groups would get two-hit results and faster timing, while upon X=0, the timing equals the in-order execution timing.

To conclude, the subtle timing leaks in P arise from the simultaneous blend of proper out-of-order schedules and specific secret inputs. To this end, a systematic analysis technique that (1) automatically distinguishes feasible leak-introducing out-of-order schedules, and (2) precisely generates the leak-endorsing inputs would be indispensable. However, this analysis has rarely been achievable in literature, and such shortage motivates our SymO3 work.

3 PRELIMINARIES

This section presents the preliminary knowledge of our work.
3.1 The Out-of-Order Execution

3.1.1 The Concept. Pipelined processors usually divide the execution of instruction into several stages to utilize the pipelines for best throughput. However, it is believed that a deeper pipeline will not contribute more beyond some turning points due to the hardware cost and the control/data hazards. Alternatively, the instruction-level parallelism, especially the out-of-order execution, plays a crucial role in promoting the pipeline performance [Li et al. 2004].

At the high-level, out-of-order execution leverages the dynamic data flow rather than the static program order to schedule instructions. A processor usually maintains a “sliding window” of consecutive instructions. Once an instruction turns to be ready, the processor selects that instruction from the window and schedules it to run. Therefore, the executed orders of independent instructions may disobey the original program order.

Fig. 3 shows an example program $P$ [Etsion 2013] as well as the in-order and out-of-order instruction scheduling strategies on running $P$. Fig. 3(a) lists the six instructions of $P$, labelled by $I_n$ where $n\in[1,6]$. Fig. 3(b) presents the data flow graph in which a node corresponds to an instruction, and the directed edges denote the data dependencies. For example, the edge from node ① to ② indicates instruction $I_2$ depends on $I_1$, in terms of the annotated $r_1$. Fig. 3(c) draws the two schedulings of these instructions on running $P$.

Under the in-order execution, instructions are sequentially executed in the program order (ref. Fig. 3(c)). Though $I_3$, $I_4$, and $I_5$ are data-independent of preceding instructions $I_1$ and $I_2$ (ref. Fig. 3(b)), they cannot execute before these predecessors.

Instead, out-of-order execution exploits the parallelism of scheduling independent instructions. Assuming division computation consumes 20+ cycles while others like addition and subtraction merely need 1-3 cycles. Then the first instruction $I_1$ would cost an inevitably long execution time. However, during such long latency, processors can schedule $I_3$-$I_5$ to run ahead of $I_2$ (ref. Fig. 3(c)). Though $I_5$ relies on both $I_3$ and $I_4$, and it has to execute after them, this schedule still benefits the overall performance. Note that out-of-order execution must preserve the program dependency semantics. Hence, $I_2$ and $I_6$ cannot be scheduled beforehand.

At the microarchitectural level, the instruction execution process can be divided into multiple steps as fetch, decode, issue, execute, and commit, as shown in Fig. 4 [Etsion 2013]. Following the program order, a processor fetches and decodes a set of instructions and issues them to multiple reservation stations (RSs). Once the operand data of an instruction gets ready and no contention appears on the available executing resource, i.e., a specific function unit (FU), the instruction will be moved from the RS to that FU to run. This mechanism, thus, gives birth to out-of-order execution in a first-ready-first-run manner. At the end of the execute step, finished instructions broadcast their computed data to signal the awaiting dependents to proceed. Finally, the finished instructions enter the circular reorder buffer and commit in the program order again.
3.1.2 Our Focus. In this work, our SymO₃ method leverages symbolic execution to examine the cache side-effects from the out-of-order execution of memory instructions. Two recent symbolic execution methods SpecuSym [Guo et al. 2020] and KLEE-Spectre [Wang et al. 2020] model microarchitectural speculative execution for side-channel leak pinpointing and vulnerability detection, which closely relate to SymO₃. However, SymO₃ has threefold distinctions.

The speculative execution studied in [Guo et al. 2020; Wang et al. 2020] and the out-of-order execution studied in SymO₃ are two special instances of the general out-of-order execution, as later discussed in Section 6. To minimize the vagueness, we by default use the term out-of-order execution throughout this paper to refer to the out-of-order scenario studied in SymO₃. Speculative execution beforehand probes a sequence of instructions under a predicted branch leg, while out-of-order execution tries to schedule the ready instructions as soon as possible to the reservation stations. From the microarchitectural angle, they have orthogonal semantics; thus, we composite the cache impacts from both in Section 5.4.

Besides, in Section 4, we establish the formalization and algorithm for the out-of-order execution. They highly differ from those in SpecuSym and KLEE-Spectre, despite all these methods share the fundamental state-forking mechanism from KLEE. Also, SpecuSym and KLEE-Spectre developed forward modeling techniques while in contrast, SymO₃ relies on backward modeling and prunes unnecessary forward executions. However, SymO₃ cannot model the speculative behaviors in SpecuSym and KLEE-Spectre, and vice versa.

Moreover, SymO₃ broadens the spectrum of the dynamic partial order reduction (DPOR) style analysis to the cache analysis in Section 4. Classic DPOR methods like [Flanagan and Godefroid 2005; Yang et al. 2008] are confined to the equivalent classes of thread interleavings. SymO₃ fuses the hardware behaviors and the software restrictions into a synergistic reasoning algorithm upon symbolic execution. Its general out-of-order modeling approach may apply to analogical problems, e.g., the relaxed memory model analysis [Kusano and Wang 2017] with tolerable porting efforts.

3.2 The Cache Basics

CPU cache dedicates to reducing the average latency of visiting memory locations from the CPU. When the CPU requests a memory operation for some data, it first inquiries the data existence in the cache. If not cached, which indicates a cache miss, the request goes to memory to load data into cache, and then returns the data to CPU. If the data is in the cache, which turns to be a cache hit, the request directly retrieves the data without touching the memory.

The modern cache hierarchy often consists of L1, L2, and L3 caches. L1 cache has two sub-types as instruction cache (i-cache) and data cache (d-cache). L1 and L2 caches reside in the same processor core, while L3 cache is shared among multiple cores. In general, the closer to the processor core, the faster cache speed and the smaller cache capacity, and vice versa. For example, the Intel Coffee Lake microarchitecture-based CPU Core i7-8850H has six cores that share a 9MB L3 cache. Meanwhile, each core owns 64KB L1 cache (half i-cache and half d-cache) and 256KB L2 cache.

Program analysis-based cache analysis methods [Basu and Chattopadhyay 2017; Brotzman et al. 2019; Chattopadhyay et al. 2017; Guarnieri et al. 2020; Guo et al. 2018, 2020; Gysi et al. 2019; Wang et al. 2020, 2017, 2019; Wu and Wang 2019] abstract the microarchitectural behaviors of certain instructions to form interceptable models, e.g., the cache-state abstract domain and the constraint formula of cache behaviors, for the reasoning upon software-level techniques. However, modeling the complete cache hierarchy by program analysis yet misses practical solutions. First, the consistency synchronization within the hierarchical caches can rapidly deteriorate the state explosion problem of cache status. Second, latencies of different caches pose extraordinary challenges for the uniform and precise modeling of timing problems. Third, the L3 cache shared by multiple cores largely exaggerates the above two issues. To the best of our knowledge, only one
A 64-bit memory address \( addr \)

\[
\begin{array}{c|c|c}
\text{52 bits} & \text{6 bits} & \text{6 bits} \\
\hline
64-6-6=52 & \log_2(32\text{KB}/64/8)=6 & \log_2(64)=6 \\
\end{array}
\]

\( \text{tag}(addr) \) \hspace{1cm} \( \text{set}(addr) \) \hspace{1cm} \text{Line offset}

Fig. 5. The tag id, associated set and line offset of \( addr \) under a 32KB 8-way cache.

recent static analysis method [Gysi et al. 2019] covers L2 and L3 caches of a fully-associative cache. However, the cost of modeling mainstream set-associative caches remains unknown. Moreover, the out-of-order analysis for timing leak detection still lacks, even under the L1 cache. As a result, in this work we concentrate on the L1 d-cache and use it to represent the cache for short.

In practice, the cached data is organized by cache lines and cache sets. For instance, the 64-byte cache line size of a 32KB d-cache means there are 512 (32KB/64B) lines in total. The 64-byte size also implies the minimum unit of contiguous data that loads from memory each time. The approaches that associate different memory locations to certain cache lines leads to three major cache types as direct-mapped cache, fully-associative cache, and set-associative cache. In our experiments, we evaluate SymO\(_3\) on set-associative caches because of its proven practicability.

In accordance with the latter cache analysis in Section 4.5, we define some formal notations:

- \( addr \) represents the memory address involved in a memory operation instruction.
- \( \text{set}(addr) \) gives the cache set with which \( addr \) associates.
- \( \text{tag}(addr) \) gives the tag id of \( addr \) w.r.t a given set-associative cache.
- \( K \) denotes the cache associativity, e.g., a 8-way set-associative cache indicates each set contains eight cache lines thus \( K=8 \).

Fig. 5 demonstrates the calculations of the tag id and the associated set of a 64-bit memory address \( addr \) under a 32KB 8-way set-associative cache. We assume each cache line here has a 64-byte size. Due to this 64-byte line, the rightmost 6 bits, which represent the location of the 1-byte data of \( addr \) in a mapped cache line, can be calculated by \( \log_2(64)=6 \). Next, based on the 8-way associativity and the 64-byte line size, we compute the number of cache sets to be \( 32\text{KB}/64/8=64 \). Thus \( \text{set}(addr) \) returns the value of the middle 6 bits. Then, the remaining 52 bits belong to the tag id, which can be obtained by \( \text{tag}(addr) \).

Then, we define a function \( \phi \) to formally check whether two addresses \( addr_1 \) and \( addr_2 \) could map to the same cache line:

\[
\phi (addr_1, addr_2) := \text{set}(addr_1) = \text{set}(addr_2) \land \text{tag}(addr_1) = \text{tag}(addr_2)
\]  

(1)

Note that \( \phi \) does not require \( addr_1 \) and \( addr_2 \) to be the same address. Exampling the Fig. 5 cache, visiting \( addr_1 \) could load a block of 64-byte memory data \( d \) to fully fill a cache line \( l \). If \( d \) still exists in \( l \) on visiting \( addr_2 \) afterward and \( \phi \) satisfies, a cache hit appears.

### 3.3 The Cache Timing Leaks

Confidential data of critical programs may potentially leak to the unsafe zone via microarchitectural side-channels when executing the vulnerable program implementations. The indistinguishability property [Yu et al. 2019] formalized the elimination of a microarchitectural side-channel. That is, for any public input \( t \) and any two secret inputs \( \tau \) and \( \tau' \), their execution traces must have no distinguishable physical effects against a specific microarchitecture component:

\[
\psi := \forall t, \tau, \tau'. \text{Obs} (\text{Imp}_P(P(t, \tau))) = \text{Obs} (\text{Imp}_P(P(t, \tau')))
\]  

(2)
Equation (2) characterizes the general abstraction of the side-channel-free property. Function $\text{Obs}$ evaluates the external adversary’s observation ability on the physical effects. $\text{Imp}_p$ represents a microarchitecture implementation. $P$ is the running program that inputs both public and secret data, e.g., an encryption algorithm program that processes the plaintext and a private key to generate the ciphertext.

Projecting the $\text{Imp}_p$ to the cache, we obtain the leakage-free property of cache timing side-channel as follows. The cache timing leaks stem from the timing variance of cache behaviors when executing different inputs. If we treat program $P$ as a function of input $\text{in} := (t, \tau)$, and let $T(P(t, \tau))$ denotes the execution time of an execution trace on $\text{in}$, then the observation $\text{Obs}$ against cache timing can be approximated to the observed execution time of that trace upon input $\text{in}$:

$$\forall t, \tau . \text{Obs}(\text{Imp}_p(P(t, \tau))) \approx \forall t, \tau . \text{Obs}_{\text{timing}}(P(t, \tau))$$  \hfill (3)

$P$ may have multiple traces from various program paths. Since we target timing leaks introduced by out-of-order execution, we assume $P$ is leakage-free under in-order execution, which is:

$$\psi_{\text{in-order}} := \forall t, \tau, \tau' . \text{Obs}_{\text{timing}}(P(t, \tau)) = \text{Obs}_{\text{timing}}(P(t, \tau'))$$  \hfill (4)

Still, $\tau$ and $\tau'$ represent any pair of secrets, and $t$ is the public input. $P$ is free of cache timing side-channel leakage if two arbitrary in-order program traces have indistinguishable execution time. Similarly, we define whether $P$ suffers from new leakage under out-of-order execution by checking the following formula:

$$\psi_{\text{out-order}} := \psi_{\text{in-order}} \land \left( \exists t, \tau, \tau' . \text{Obs}_{\text{timing}}(P_{\text{out-order}}(t, \tau)) \neq \text{Obs}_{\text{timing}}(P_{\text{out-order}}(t, \tau')) \right)$$  \hfill (5)

A new leak appears under the out-of-order execution context if two secret inputs $\tau$ and $\tau'$ do exist and cause distinguishable $\text{Obs}_{\text{timing}}$ in the corresponding out-of-order traces. Note that we primarily care about whether the secret can interfere with the cache state along an out-of-order trace. In the analysis, we configure $t$ to a concrete value to reduce the runtime overhead.

### 3.4 The Threat Model

As manifested in Section 2.2, external adversaries may learn the private data from analyzing the victim’s runtime timing under out-of-order execution. To be capable of accomplishing such a threat, the adversaries have to have privileged access to a malicious process sharing the physical cache with the victim; so those adversaries can measure the interested memory access latency through probe methods [Gruss et al. 2016; Yarom and Falkner 2014; Zhang et al. 2011]. They should also be able to reason about the source code of the victim for possible out-of-order leaks as well as triggering the execution of the victim. We believe this threat model is reasonable as it has been proved feasible in practical attacks, such as [Disselkoen et al. 2017; Osvik et al. 2006; Yarom and Falkner 2014]. It is also widely adopted in recent side-channel reasoning works [Brotzman et al. 2019; Doychev and Köpf 2017; Guo et al. 2020; Wang et al. 2020, 2017; Wichelmann et al. 2018].

### 4 OUT-OF-ORDER SYMBOLIC EXECUTION

This section reviews the baseline symbolic execution and then introduces the core components of our SymO3 which build upon the baseline algorithm.

#### 4.1 The Baseline Symbolic Execution

Symbolic execution [Clarke 1976; King 1976] has witnessed extensive advances [Bergan et al. 2014; Bucur et al. 2011; Cadar et al. 2008; Ciortea et al. 2009; Guo et al. 2015; Pasareanu and Rungta...
Exposing Cache Timing Side-Channel Leaks through Out-of-Order Symbolic Execution

Poeplau and Francillon 2020] in recent years. In this work, we adopt the recursive symbolic execution algorithm and related knowledge from [Guo et al. 2015, 2018] as the baseline upon which we build the new components of SymO₃.

First, we assume that a program \( P \) comprises a finite set of paths, and each path contains a finite sequence of instructions. Let \( \text{inst} \) be short for an instruction then \( \epsilon := (l, \text{inst}, l', \text{nxt}) \) denotes the symbolic event of an \( \text{inst} \). \( l \) and \( l' \) represents the program locations before and after \( \text{inst} \), and \( \text{nxt} \) points to the next symbolic event for execution. The execution of \( P \) on input \( \text{in} := \{t, \tau\} \) explores a series of symbolic events along the program path to which \( \text{in} \) leads.

Second, a symbolic state \( s \) stands for the frontier of exploring a path in \( P \) during the dynamic execution. We also present \( s \) in a tuple as \( s := (\epsilon, \text{pcon}, \hat{m}) \). \( \epsilon \) is the current symbolic event to execute at \( s \); \( \text{pcon} \) is the path condition from the execution entry to \( s \); \( \hat{m} \) contains the memory mappings of variables and their symbolic values at the time reaching \( \epsilon \).

Third, we abstract away the implementation details of symbolic event interpretation and perform our analysis only against the execution of memory events. A memory event must associate with a memory operation instruction, i.e., a memory read like \( \text{v} = \text{load addr} \), or a memory write as \( \text{store addr val} \), where \( \text{addr} \) is a memory address, and \( \text{val} \) denotes a symbolic value. We use \( \text{M-event} \) to refer to this event type. Other kinds of symbolic events, such as branch event and arithmetic events, would execute as usual in symbolic execution since they do not directly impact the cache status.

### 4.2 The Out-of-Order Generation

**Algorithm 1** The Out-of-Order Generation Algorithm

1. **Initially:** Create a global stack \( \text{St} \); start \( \text{StdSymExec} \) with a seed state \( s_0 \) on input \( \{t_0, \tau_0\} \).

2. **procedure** \( \text{StdSymExec} (\text{Symbolic\_State } \text{state}) \)
   3. \( \text{St}.\text{push}(\text{state}); \)
   4. **if** \( \text{state.}\epsilon \) is a \( \text{M}\)-event **then**
   5. \( \text{O₃Gen}(\text{state}); \)
   6. \( \text{StdSymExec}(\text{NextSymState}(\text{state})); \)
   7. **else**
   8. \( \text{StdSymExec}(\text{NextSymState}(\text{state})); \)
   9. **end if**
   10. \( \text{St}.\text{pop}(); \)
   11. **end procedure**

12. **function** \( \text{O₃Gen} (\text{Symbolic\_State } \text{crt}) \)
   13. \( \text{crt.}\text{done} \leftarrow \text{crt.}\text{done} \cup \text{crt.}\epsilon; \)
   14. **if** \( \text{crt.}\epsilon \) is a memory read event **then**
   15. \( \text{let} \ e \text{ be the closest event in } \text{crt.}\text{trace} \text{ that RNA}(\text{crt, e}) \text{ returns true; } \)
   16. **if** \( e \neq \text{null} \) **then**
   17. \( \text{prv} \leftarrow \text{the symbolic state that was about to execute } e; \)
   18. **if** \( \text{crt.}\epsilon \notin \text{prv.}\text{done} \) **then**
   19. \( \text{ReorderModeling}(\text{crt, prv}); \)
   20. **end if**
   21. **end if**
   22. **end if**
   23. \( \text{crt.}\text{trace} \leftarrow \text{crt.}\text{trace} \cdot \text{crt.}\epsilon; \)
   24. **end function**

In this section, we introduce the main procedure of SymO\textsubscript{3} which starts at the interpretation point of a M-event. We observe that, fundamentally, the feasible out-of-order behaviors among a window of instructions can be decomposed to the blend of multiple two-instruction reorderings. However, a straightforward reordering enumeration of potential instruction pairs can quickly result in the state explosion. Also, certain rules of the environmental restrictions should be applied to ensure the feasibility of the various reorderings. Moreover, a large portion of these pairs might be redundant in terms of the unchanged cache state. Inspired by the dynamic partial order reduction [Flanagan and Godefroid 2005; Yang et al. 2008, 2010], which searches for equivalent classes of thread interleavings in concurrency analysis, we design a new algorithm Out-of-order Generation, to systematically generate unique out-of-order symbolic states for the cache analysis.

4.2.1 The Algorithm. Algorithm 1 presents the proposed algorithm, which builds upon the baseline symbolic execution procedure StdSymExec. We retain the unchaned part of StdSymExec, e.g., the handling of other symbolic events except the memory events, and the sub-procedure NextSymState which processes a symbolic event and returns the new state. Details of StdSymExec can be found in [Guo et al. 2015, 2018]. The main changes start at line 5 — a new function O\textsubscript{3}Gen which ignites the generation of instruction-level out-of-order executions by a backward analysis against the current state state.

To keep pace with O\textsubscript{3}Gen, we extend the symbolic state s to be \(s := (e, pcon, \hat{m}, \text{trace}, \text{done})\). During the exploration of s, trace records the executed memory events; done records the set of scheduled events at s since the out-of-order modeling now may schedule multiple events at the same program location. Generally, O\textsubscript{3}Gen first updates the done set of state \(crt\) (line 14), meaning that event \(crt.e\) has been processed in this backtracking, to avoid future re-analysis. Then, if \(crt.e\) is a memory read event, O\textsubscript{3}Gen backwardly searches \(crt.trace\) for the closest event \(e\) that may run out-of-order with \(crt.e\), and the Reorder Necessity Analysis (RNA, ref. Section 4.5) can return true (line 16). If such \(e\) does not exist, O\textsubscript{3}Gen appends \(crt.e\) to \(crt.trace\) (line 24) and exits.

In contrast, if the desired event \(e\) does exist, then at line 18, O\textsubscript{3}Gen retrieves a prior symbolic state \(prv\) who was about to execute \(e\). More importantly, if \(crt.e\) is not in the done set of state \(prv\), O\textsubscript{3}Gen sends \(crt\) and \(e\) to function ReorderModeling for the two-event out-of-order behavior modeling (ref. Section 4.4). In the end, O\textsubscript{3}Gen also updates \(crt.trace\) at line 24.

**Fig. 6.** The out-of-order generation and the preserved four new states

4.2.2 The Example. Fig. 6 shows an example for Algorithm 1. Without out-of-order execution, there is an in-order execution path \(p1\) of four memory read events a, b, c, and d. We assume that b only depends on a and d only depends on c; but, c is independent of both a and b. Then, the example is to, under the given assumptions, systematically generate new symbolic states to model the distinct out-of-order behaviors along with exploring the in-order path \(p1\).

Following Algorithm 1, the forward symbolic execution procedure StdSymExec calls the first
O₃Gen, marked as #1, on approaching event a. Here we use the path-event convention, to name a
symbolic state. So the state right before a is p1-a. The O₃Gen #1 puts a into the p1-a.done, but it
cannot find an event e before a because of the empty p1-a.trace. So #1 quickly returns after placing
a into p1-a.trace.

Whereafter, the second O₃Gen invocation, #2, starts on arriving b. As b relies on a, the RNA
judges false (line 16, Algorithm 1), which means b cannot run before a. We color the backward
arrow in gray to mark such infeasible reordering. Similarly, both p1-b.trace and p1-b.done now
contains b after #2.

Going to c, O₃Gen #3 finds that c and the closest event b qualify the RNA rules (ref. Algorithm
2). First, the in-between distance, one, is less than the reorder buffer size of three. Second, b and c
are data-independent; thus, they are unreachable in the data dependency graph built from a, b, and
c. Third, due to the assumed fully-associative cache, b and c have a mutual effect against the cache
state. Lastly, c is not in the done set of p1-b who only contains b. As a result, the ReorderModeling
function (line 20, Algorithm 1) forks a new state aux1 from p1-b, reorders c and b, and places aux1
to the global state pool for future exploration.

Proceeding to d, O₃Gen #4 also returns early due to the assumed dependency between c and d.
After the execution goes to the end of p1, StdSymExec pops out states p1-d, p1-c, and p1-b due to
the recursion, and uses aux1 for the new exploration. At this point, aux1 is to execute c. O₃Gen #5
duplicates a new state aux2 from aux1, to model the out-of-order situation between c and a. After
#5, StdSymExec schedules c and continues to b. Again, O₃Gen tries backtracking from b to c (#6).
Though this time #6 can get through the RNA rules, it fails the checking at line 19 of Algorithm 1
because aux1.done already contains b from inheriting the done set of p1-b. Thereby, the condition
b   ∈    aux1.done is unsatisfiable, and #6 cannot trigger ReorderModeling for this reason.

Repeating StdSymExec until the global state stack St gets empty, our Out-of-order Generation
algorithm systematically explores unique out-of-order event pairs and prunes infeasible executions.
Moreover, it integrates the new auxiliary states into symbolic executor in a uniform way. In the
result of Fig. 6, our algorithm backtracks fourteen times but only preserves four distinct states that
represent unique out-of-order behaviors, as shown by the dotted arrow lines in Fig. 6.

4.2.3 The Proof. To show that Algorithm 1, when combined with Algorithm 3 and 2, covers all
feasible out-of-order program executions, we prove the following theorem based on induction:

**Theorem 4.1.** Suppose an in-order execution of a program with m instructions generates an event
sequence \{e₁, e₂, ..., eₘ\}. Algorithm 1 generates the corresponding event sequences for any valid
out-of-order execution of the program.

Consider the first n instructions (in program order):

1. When n=1, Theorem 4.1 holds because one event only has one event sequence, regardless of
   in-order or out-of-order.
2. Assume Theorem 4.1 holds for the first n instructions; we denote all possible event sequences
   as P₁, P₂, ..., Pₓₙ. Xₙ represents the total number of valid executions, including both in-order
   and out-of-order.
3. When considering the first n + 1 instructions, we first notice that any valid event sequence
   can be only constructed by inserting eₙ₊₁ to an event sequence P from P₁, P₂, ..., Pₓₙ without
   interchanging the order of any two events in the original sequence P, modulo the constraints
   specified in Algorithm 2. The proof is straightforward: since eₙ₊₁ is younger than e₁, e₂, ..., eₙ
   in program order, its execution won’t affect the execution of the prior n instructions. Therefore,
in any valid execution event sequence of the first \( n + 1 \) instructions, subtracting \( e_{n+1} \) should yield a valid event sequence of the first \( n \) instructions which appears in \( \{P_1, P_2, \ldots, P_{X_n}\} \).

(4) To construct all possible execution sequences consisting of \( e_1, e_2, \ldots, e_{n+1} \), we start by appending \( e_{n+1} \) to every sequence \( P \in \{P_1, P_2, \ldots, P_{X_n}\} \), and do \( O_3 \) on the final state of \( P \) for \( e_{n+1} \) based on Algorithm 1. \( O_3 \) swaps the \( e_{n+1} \) with the previous event along the sequence \( P \) unless the distance or data dependency constraint check in Algorithm 2 fails. This ensures that we cover all possible event sequence based on \( P \), since \( e_{n+1} \) cannot be moved forward due to the RNA rules, and doing \( O_3 \) for all possible sequences with \( n \) instructions (\( e_1, e_2, \ldots, e_n \)) is sufficient due to the analysis in (2).

### 4.3 The Reorder Necessity Analysis

**Algorithm 2 The Reorder Necessity Analysis (RNA)**

```
1: function RNA (Symbolic_State crt, Event e)
2:     d ← the distance between crt.e and e in the program order;
3:     if \( d > 0 \) \&\& \( d < rbs \) then ▶ The distance check
4:         G ← the data dependency graph of crt.trace;
5:         if crt.e and e are unreachable in G then ▶ The dependency check
6:             if crt.e and e have mutual cache effects then ▶ The cache effect check
7:                 return true;
8:         end if
9:     end if
10: end if
11: return false;
12: end function
```

This section proposes the Reorder Necessity Analysis (RNA), to assist the above Out-of-order Generation algorithm. Due to the microarchitectural restrictions and the software data dependency, only a subset of memory instructions may get involved in out-of-order execution. Therefore, we design Algorithm 2 to enforce the necessity analysis against two candidate events.

In general, Algorithm 2 conducts a threefold analysis. The primary function RNA takes a symbolic state \( crt \) and a symbolic event \( e \) as inputs. At line 2, RNA first calculates the distance \( d \) — the number of events between \( e \) to \( crt.e \) in the program order.

Then, if \( d \not\in (0, rbs) \), RNA directly returns \( false \) at line 11, meaning that out-of-order execution of \( crt.e \) and \( e \) is infeasible. Otherwise, RNA continues to data dependency analysis (lines 4-5). Here \( rbs \) shorts for the processor reorder buffer size. Learned from Fig. 4, instructions enter and leave the reorder buffer in the program order. Only the head instruction in this circular buffer has retired can its direct successor become the head. Then an awaiting instruction can enter this buffer. Thus, if the number of in-between events of \( crt.e \) and \( e \) reaches or exceeds \( rbs \), the two events have no chance to perform the out-of-order execution.

Going to the data dependency analysis, at line 4 RNA constructs a data dependency graph from \( crt.trace \). Note that based on symbolic execution, we can obtain dynamic variable values to build a precise graph of \( G \) like Fig. 3(b). If two events, i.e., \( crt.e \) and \( e \), cannot reach each other in the directed graph \( G \), we affirm that they are independent and proceed to the third analysis, which checks cache state interference. Otherwise, RNA also returns \( false \).

The last analysis checks the mutual effect on the cache state between the given events (lines 6-7). The inversed execution of two independent events may not necessarily be useful if the cache state,
e.g., the cache content or the most recently used line of a cache set, remains the same no matter which event runs first. If events \( e \) and \( \text{crt}.e \) cannot impact each other regarding the cache status, RNA returns \textit{false}. If not, we deem the necessity of out-of-order modeling and let RNA return \textit{true}.

We use Fig. 7, which depicts the out-of-order scenario of the motivating example, to explain Algorithm 2. First, by observing the in-order events on the right-side in-order trace, we count the distance between two events \( \text{load X} \) and \( \text{load Y} \) be two (including the former). Thus, \( d \in (0, \text{rbs}) \) satisfies because \( \text{rbs} \) assumes to be 64 in the example. Second, since \( X \) and \( Y \) are not aliasing, the memory read events of them are data-independent. Third, recall that the motivating example uses a fully-associative cache. We view such a cache as a particular single-set cache to which any memory operation might cause status change. Consequently, it is necessary to reorder the two events, and RNA correspondingly returns \textit{true}.

Note that the analysis under fully-associative cache costs more computations but misses no suspicious cases. Moreover, for regular set-associative caches, memory events associated with different sets may get RNA to return \textit{false}.

4.4 The Reorder Modeling

As discussed, the complicated out-of-order execution along a program path can essentially break into the combination of multiple two-event cases. We design the basic reorder modeling method of two given events in Algorithm 3, which can embed into the Out-of-order Generation algorithm introduced in Section 4.2.

Function \texttt{ReorderModeling} inputs two symbolic states, \texttt{crt} and \texttt{prv}, and tries to schedule them out-of-order, which is to let event \texttt{crt}.\texttt{e} run before \texttt{prv}.\texttt{e}. Note that \texttt{crt} must be forwardly reachable from \texttt{prv}, and event \texttt{crt}.\texttt{e} has to be independent of \texttt{prv}.\texttt{e}. Otherwise, \texttt{crt}.\texttt{e} cannot execute beforehand.

**Algorithm 3** The Reorder Modeling of Two Symbolic Events.

1: function \texttt{ReorderModeling} (Symbolic State \texttt{crt}, Symbolic State \texttt{prv})
2: \hspace{1em} \texttt{aux} \leftarrow fork a new symbolic state from \texttt{prv} and put it to state pool;
3: \hspace{1em} \texttt{e} \leftarrow declare an event variable that points to \texttt{aux}.\texttt{e};
4: \hspace{1em} \texttt{q}1, \texttt{q}2 \leftarrow declare two empty queues for symbolic events;
5: \hspace{1em} \texttt{while} \texttt{e} \preceq \texttt{crt}.\texttt{e} \texttt{do} \hspace{1em} \texttt{if} \texttt{e} is dependent of \texttt{aux}.\texttt{e} \texttt{then}
6: \hspace{2em} \texttt{q}1.\texttt{back}.\texttt{nxt} = \texttt{e};
7: \hspace{2em} \texttt{q}1.\texttt{push}(\texttt{e});
8: \hspace{1em} \texttt{else}
9: \hspace{2em} \texttt{q}2.\texttt{back}.\texttt{nxt} = \texttt{e};
10: \hspace{2em} \texttt{q}2.\texttt{push}(\texttt{e});
11: \hspace{1em} \texttt{end if}
12: \hspace{1em} \texttt{e} \leftarrow the next event along the execution path to \texttt{crt};
13: \hspace{1em} \texttt{end while}
14: \hspace{1em} \texttt{aux}.\texttt{e} \leftarrow \texttt{q}2.\texttt{front}(); \hspace{1em} ▷ Update the current event of \texttt{aux}
15: \hspace{1em} \texttt{q}2.\texttt{back}.\texttt{nxt} = \texttt{q}1.\texttt{front}(); \hspace{1em} ▷ Concatenate two event lists
16: \hspace{1em} \texttt{q}1.\texttt{back}.\texttt{nxt} = \texttt{crt}.\texttt{e}.\texttt{nxt}; \hspace{1em} ▷ Redirect to the next event
17: \hspace{1em} \texttt{q}1.\texttt{push}(\texttt{e});
18: \hspace{1em} \texttt{Clean temporary queues} \texttt{q}1 \texttt{and} \texttt{q}2;
19: \hspace{1em} \texttt{end function

At the beginning, \texttt{ReorderModeling} forks a new auxiliary state \texttt{aux} from its second input \texttt{prv} (line 4), to act as the modeling transmitter. After the duplication, \texttt{aux} has the same state snapshot...
to $prv$. Then, at lines 3-4, we define a temporary symbolic event $e$ that points to $aux.e$, and two empty queues $q_1$ and $q_2$. Next, $ReorderModeling$ iteratively checks the dependency of $e$ and $aux.e$ (line 6), and updates $e$ to the next event (lines 13) along the trajectory between $prv$ and $crt$, until $e$ reaches $crt.e$ at the trajectory tail (line 5).

The while loop (lines 5-14) separates the iterated events into two queues. If the event that $e$ references to is data-dependent of $aux.e$ (line 6), we placed it to $q_1$ (line 8). Similarly, we put events that are irrelevant to $aux.e$ into $q_2$ (line 11). Besides, before pushing $e$ to either $q_1$ or $q_2$, we link the neighboring events by associating the queue tail event with $e$ (lines 7 and 10).

After the loop, $ReorderModeling$ performs three updates. First, it resets $aux.e$ to the head event of $q_2$ (line 15), indicating that $aux$ now is to execute the first event of the linked events that are irrelevant to the original $aux.e$. Second, it makes $q_2$’s tail event point to $q_1$’s head event (line 16) to reconnect the two event sequences. Finally, it sets the successor of $q_1$’s rear event to the next event of $crt.e$ (line 17), to finish rebuilding the event chain between $aux$ and $crt$. In the future, once the symbolic executor schedules state $aux$, it would execute along the reconstructed event chain from $aux$ to $crt$, which realizes the artificial out-of-order behavior.

In general, $ReorderModeling$ mimics the two-event out-of-order execution by exploiting the stateful instinct of symbolic execution. Fig. 7 shows the modeling of the studied case in Section 2.2. To schedule $load Y$ before $load X$, let us name the two $load$ events to be $E_1$ and $E_3$, respectively. Following Algorithm 3, we divide the modeling progress into four steps in Fig. 7. The motivating program $P$ has only one path; thus, we depict the path by a sequence of memory instructions. The black nodes represent the corresponding memory events.

In the example, we assume that the invocation of $ReorderModeling$ happens at state $crt$, who is about to execute $E_3$. The other state, $prv$, was to execute $E_1$. Since $crt$ is forwardly reachable from $prv$ and $E_1$ and $E_1$ are data-independent, $ReorderModeling$ can start with the two states $prv$ and $crt$ safely.

As shown in Fig. 7, step ① first creates $aux$, a duplicate of state $prv$. Now, $aux.e$ is $E_1$. After the while loop in $ReorderModeling$, we get $q_1:=[E_2, E_1]$ and $q_2:=[E_1]$. Next, step ② resets $aux.e$ to the front event of $q_2$, which turns to be $E_1$. Later, step ③ makes the $nxt$ event of $E_1$ be $E_1$ because $E_1$ is the head event in $q_1$, and $E_1$ is also the back event of $q_2$. Finally, step ④ redirects the successor of $q_1$’s tail event, $E_1$, to be the $nxt$ event of $crt.e$, which goes to the end of the path. So far, the event chain reorganization finishes. In the future, the symbolic executor would first steer $aux$ to $E_1$ and produce the out-of-order behavior accordingly.

### 4.5 Cache Leak Analysis

This section leverages the constraint-solving based approach to reason about the existence of leaks. Specifically, we adopt the per trace approach [Basu and Chattopadhyay 2017; Chattopadhyay 2017;...
Chattopadhyay et al. 2017; Guo et al. 2018, 2020; Wang et al. 2020], to conduct leak analysis against the memory events on each *out-of-order* execution trace.

Algorithm 4 The Cache Leak Analysis

1: procedure StdSymExec (Symbolic_State state)
2:     
3:     if state.ε is a M-event then
4:         CACHELEAKANALYSIS(state); ▷ Start the leak analysis
5:         O₃GEN(state);
6:     
7:     end if
8:     
9: end procedure

10: function CACHELEAKANALYSIS (Symbolic_State crt)
11:     
12:     if crt is an out-of-order state && crt.ε is relevant to sensitive input then
13:         σ ← form the leak constraint for event crt.ε;
14:         Solve σ ∧ crt.pcon for satisfiable inputs;
15:     end if
16: end function

Algorithm 4 describes the leak analysis. Generally, before starting the O₃Gen on a memory event state.ε, we check whether it may lead to a cache timing leak. That is, before symbolically executing state.ε, we analyze if, under some specific inputs, the cache behavior of state.ε differs from its original behavior under in-order execution. SymO₃ accomplishes such functionality by adding CACHELEAKANALYSIS into the StdSymExec (line 4). In this new function, if the state crt is generated from the out-of-order modeling, and crt.ε is relevant to the sensitive input (line 12), we build the constraint σ that describes the leak condition of crt.ε (line 13), and solve σ ∧ pcon for possible solutions (line 14).

Recall that we assume the target program is timing leakage-free in in-order execution; thus, each in-order memory event causes either must-hit or must-miss. Then, σ states the condition for the existence of a different cache behavior:

\[
\sigma := \exists in. (a_{in}(e_i) \neq \Delta_{e_i})
\] (6)

To be specific, given a memory event \(e_i\), if there exists an input \(in\) that makes \(a_{in}(e_i)\), the cache hit constraint of \(e_i\) under out-of-order execution, compute a different value from the in-order cache behavior \(\Delta_{e_i}\), then there is a leak at \(e_i\). Here \(\Delta_{e_i}\) is a constant hit/miss value, e.g., 1 for hit and 0 for the miss, that can be obtained by an in-order run with arbitrary input.

To formally present \(a_{in}(e_i)\), we need two more notations:

- The trace constituent of a symbolic state consists of the executed memory events \(\{e_1, \ldots, e_n\}\) in the execution order, where \(n\) is the total number of the events.
- \(a_i\), where \(i \in [1, n]\), represents the memory address involved in \(e_i\).

Next, we present the precise definition of \(a_{in}(e_i)\) as:

\[
a_{in}(e_i) := (\exists x \in [0, i] \mid \phi (a_x, a_i)) \land (\forall y \in (x, i) \mid \neg \phi (a_x, a_y)) \land \sum_{z=x+1}^{i-1} (set(a_z) = set(a_i)) < K
\] (7)
Per a memory event $e_i$, we first search its parent trace for the closest event $e_x$ before $e_i$, who and $e_i$ may map to the same cache line, $l$, by calling the function $\phi$ (Equation 1). Second, each in-between event $e_y$ should not access the same line $l$. The third sub-constraint relies on the cache replacement policy. Without loss of generality, we leverage the N-way set-associative cache and the LRU replacement policy. So, we count the number of addresses visited between $e_x$ and $e_i$. Moreover, these addresses must associate with the same cache set to $a_x$. More importantly, such a number has to be less than the cache associativity $K$. This sub-constraint is to prevent from eviction the $a_x$ data out of the cache, to promise a cache hit for $e_i$.

Note that our leak analysis differs from SpecuSym [Guo et al. 2020] in two aspects. First, it addresses all memory events on an out-of-order path, whereas SpecuSym will not analyze the cache behaviors of speculated events. Second, SpecuSym reasons for the sequentially executed events in speculation. Our new leak analysis can even analyze the speculated events in an out-of-order manner, and we show the result in experiments.

5 EVALUATION

This section evaluates SymO3 with a set of real-world benchmarks. We implement SymO3 on the latest KLEE 2.0 [Cadar et al. 2008] and LLVM 6.0 [Lattner and Adve 2004]. SymO3 adds four new components into the original KLEE framework.

5.1 Benchmarks and Research Questions

We use a diverse set of 24 benchmarks collected from recent works [Chattopadhyay et al. 2017; Guo et al. 2018, 2020; Gysi et al. 2019; Wang et al. 2020; Wu and Wang 2019] for the evaluation of SymO3. These C benchmarks consist of cipher programs, elliptic curve computation programs, worst-case execution time benchmarks, embedded media computation benchmarks, authenticating library code, GDK library code, etc.

Table 2 provides more information of the benchmarks. For each one, we have the benchmark name (Name), its origin (Source), the lines of C code (LoC), the number of memory visit events (#MemV), and the length of sensitive input in bytes (#In). #In also indicates the size of the symbolic input. In each benchmark, we use the klee_make_symbolic function to initialize the sensitive input into symbolic value and compile the program into LLVM bitcode.

We conduct the experiments on an Amazon EC2 c5.12xlarge instance, running the Ubuntu 16.04 64-bit Server Linux with a 48-core vCPU and 96GB RAM. The time threshold for running each benchmark is twelve hours. To evaluate SymO3 against these benchmarks, we design the following three research questions:

- **RQ1**: Is SymO3 capable of detecting the cache timing leaks by its software-based analysis?
- **RQ2**: Does program transformation, e.g., compiler optimization, have a significant effect on the SymO3 detection result?
- **RQ3**: Is SymO3 also able to support speculative execution during its dynamic analysis?

5.2 The Leak Detection

5.2.1 The Configurations. In this evaluation, we use two set-associative caches: a 32 KB 4-way cache and a 32 KB 8-way cache, namely 32K4W and 32K8W in the Cache column of Table 3. In both caches, each cache line has the 64-byte capacity. These cache statistics are close to the L1 data cache parameters in modern processors like Intel Skylake [Intel 2016] series. Thus, we deem that they are reasonable settings.

We configure the reorder buffer size to be 32, 64, and 128, as shown by the RBS value at the top row of Table 3. We use this variant to test the RBS impact on the leak detection. Note that the
reorder buffer in Intel Skylake [Intel 2016] has 224 entries in total. Since each LLVM instruction may correspond to multiple binary instructions [Poeplau and Francillon 2019] and even more micro-instructions [Abel and Reineke 2019], our three RBS settings are comparable to the reorder buffer size in real-world processors. Also, we use the -O0 compiler optimization level for the benchmark compilation in this evaluation.

For each benchmark, we record the analysis time in minutes (Time (m)), the number of out-of-order traces (#.Trace) derived from in-order executions, and the number of detected leaks (#.C/D) in Table 3. The leak introduced by out-of-order execution may exhibit two forms. We name the first one as consistently different, which means the new behavior is always different — e.g., always-hit in in-order execution but always-miss in out-of-order execution, and vice versa. We call the second type divergently different, which means the new behavior could be cache-hit under some inputs but cache-miss under some other inputs. Table 3 uses #.C/D to distinguish such two leak types. C-type leaks solely ascribe from the out-of-order schedules, and D-type leaks root from the blend of out-of-order influences and the sensitive inputs.

5.2.2 The Results. Table 3 presents the experimental results. Overall, among all the RBS settings, SYMO3 can detect leaks in five programs AES [LibTomCrypt 2019], AES [mbedtls 2017], DES [Libgcrypt 2018], FCrypt [Dellinger et al. 2011], and Seed [Tegra 2018]. We summarize our experimental findings in four points.

First, SYMO3 found no C-type leaks in all programs. This fact implies, in most cases, the out-of-order execution poisons the input data to form subtle leaks rather than directly causing leaks by the out-of-order schedules. Also, the used -O0 option avoids aggressive transformation in the compilation, which preserves the original memory-access patterns as much as possible. To investigate if the optimized compilation could cause any experimental difference, we conduct a comparison in Section 5.3.

Second, generally, the total analysis time, the amount of out-of-order traces, and the number of detected leaks all rise in line with the increasing RBS size. It is because larger RBS allows more instructions to enter the out-of-order window, which results in potentially vaster reorder state space. However, compared to the sharply raised number of #.Trace, the increasing rate of leaks and analysis time is gentle. On the one hand, SYMO3 analyzes the traces efficiently, thus costing tolerable time overhead. On the other hand, many traces cannot attribute to leaks, and SYMO3 precisely eliminates them to retain a minimum set of leaks.

Third, the increased cache associativity does not always correspond to more leaks. In AES [LibTomCrypt 2019], the 32K8W cache results in fewer leaks while in DES [Libgcrypt 2018], all the results

---

Table 2. The benchmark information: Name, Source, Lines of Code (LoC), the Number of Memory Visit Operations (#.MemV), and the length of each Sensitive Input (#.In) in bytes.

<table>
<thead>
<tr>
<th>Name</th>
<th>Source</th>
<th>LoC</th>
<th>#.MemV</th>
<th>#.In</th>
<th>Name</th>
<th>Source</th>
<th>LoC</th>
<th>#.MemV</th>
<th>#.In</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES</td>
<td>[mbedtls 2017]</td>
<td>281</td>
<td>245</td>
<td>272</td>
<td>hash</td>
<td>[Rapier and Bennett 2008]</td>
<td>320</td>
<td>5297</td>
<td>64</td>
</tr>
<tr>
<td>blowfish</td>
<td>[LibTomCrypt 2019]</td>
<td>467</td>
<td>321</td>
<td>8</td>
<td>lblock</td>
<td>[Dinu et al. 2015]</td>
<td>949</td>
<td>1233</td>
<td>10</td>
</tr>
<tr>
<td>chaskey</td>
<td>[Dinu et al. 2015]</td>
<td>255</td>
<td>192</td>
<td>32</td>
<td>salsa</td>
<td>[Tegra 2018]</td>
<td>279</td>
<td>481</td>
<td>28</td>
</tr>
<tr>
<td>encoder</td>
<td>[LibTomCrypt 2019]</td>
<td>134</td>
<td>37</td>
<td>100</td>
<td>trie</td>
<td>[freeradius 2020]</td>
<td>162</td>
<td>51</td>
<td>32</td>
</tr>
<tr>
<td>FCrypt</td>
<td>[Dellinger et al. 2011]</td>
<td>621</td>
<td>798</td>
<td>12</td>
<td>unicode</td>
<td>[GDK 2018]</td>
<td>839</td>
<td>22</td>
<td>4</td>
</tr>
</tbody>
</table>
Table 3. The leak detection results under three RBS settings

<table>
<thead>
<tr>
<th>Name</th>
<th>Cache</th>
<th>RBS:32</th>
<th>RBS:64</th>
<th>RBS:128</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time (m)</td>
<td>#.Trace</td>
<td>#.C/D</td>
<td>Time (m)</td>
</tr>
<tr>
<td>AES[LibTomCrypt 2019]</td>
<td>32K4W</td>
<td>180.33</td>
<td>68114</td>
<td>0/31</td>
</tr>
<tr>
<td>AES[LibTomCrypt 2019]</td>
<td>32K8W</td>
<td>202.93</td>
<td>100658</td>
<td>0/83</td>
</tr>
<tr>
<td>adpcm[Gustafsson et al. 2010]</td>
<td>32K4W</td>
<td>24.88</td>
<td>49093</td>
<td>0/0</td>
</tr>
<tr>
<td>chacha20[LibTomCrypt 2019]</td>
<td>32K4W</td>
<td>1.17</td>
<td>1</td>
<td>0/0</td>
</tr>
<tr>
<td>Camellia[Tegra 2018]</td>
<td>32K4W</td>
<td>720</td>
<td>17084</td>
<td>0/0</td>
</tr>
<tr>
<td>DES[OpenSSL 2019]</td>
<td>32K4W</td>
<td>189.94</td>
<td>1</td>
<td>0/0</td>
</tr>
<tr>
<td>DES[OpenSSL 2019]</td>
<td>32K4W</td>
<td>158.94</td>
<td>1</td>
<td>0/0</td>
</tr>
<tr>
<td>fRBS[freeradius 2020]</td>
<td>32K4W</td>
<td>32.74</td>
<td>49252</td>
<td>0/92</td>
</tr>
<tr>
<td>fRBS[glibc 2019]</td>
<td>32K4W</td>
<td>238.05</td>
<td>34964</td>
<td>0/92</td>
</tr>
<tr>
<td>fRBS[Fixer 2018]</td>
<td>32K4W</td>
<td>34.30</td>
<td>1459</td>
<td>0/128</td>
</tr>
<tr>
<td>fRBS[Fixer 2018]</td>
<td>32K4W</td>
<td>62.83</td>
<td>1459</td>
<td>0/120</td>
</tr>
<tr>
<td>fRBS[Fixer 2018]</td>
<td>32K4W</td>
<td>&lt; 0.01</td>
<td>1</td>
<td>0/0</td>
</tr>
<tr>
<td>fRBS[Fixer 2018]</td>
<td>32K4W</td>
<td>&lt; 0.01</td>
<td>1</td>
<td>0/0</td>
</tr>
<tr>
<td>fRBS[Fixer 2018]</td>
<td>32K4W</td>
<td>&lt; 0.01</td>
<td>1</td>
<td>0/0</td>
</tr>
<tr>
<td>fRBS[Fixer 2018]</td>
<td>32K4W</td>
<td>&lt; 0.01</td>
<td>1</td>
<td>0/0</td>
</tr>
<tr>
<td>fRBS[Fixer 2018]</td>
<td>32K4W</td>
<td>&lt; 0.01</td>
<td>1</td>
<td>0/0</td>
</tr>
<tr>
<td>fRBS[Fixer 2018]</td>
<td>32K4W</td>
<td>&lt; 0.01</td>
<td>1</td>
<td>0/0</td>
</tr>
<tr>
<td>fRBS[Fixer 2018]</td>
<td>32K4W</td>
<td>&lt; 0.01</td>
<td>1</td>
<td>0/0</td>
</tr>
<tr>
<td>fRBS[Fixer 2018]</td>
<td>32K4W</td>
<td>&lt; 0.01</td>
<td>1</td>
<td>0/0</td>
</tr>
<tr>
<td>fRBS[Fixer 2018]</td>
<td>32K4W</td>
<td>&lt; 0.01</td>
<td>1</td>
<td>0/0</td>
</tr>
<tr>
<td>fRBS[Fixer 2018]</td>
<td>32K4W</td>
<td>&lt; 0.01</td>
<td>1</td>
<td>0/0</td>
</tr>
<tr>
<td>fRBS[Fixer 2018]</td>
<td>32K4W</td>
<td>&lt; 0.01</td>
<td>1</td>
<td>0/0</td>
</tr>
<tr>
<td>fRBS[Fixer 2018]</td>
<td>32K4W</td>
<td>&lt; 0.01</td>
<td>1</td>
<td>0/0</td>
</tr>
</tbody>
</table>

are the same. In another AES [mbedTLS 2017], 32K8W brings two more leaks under RBS:32; however, the results are the same under both RBS:64 and RBS:128. By contrast, FCrypt suffers from slightly more leaks under 32K8W with RBS:64 and RBS:128 but no difference with RBS:32. Moreover, SYMO₃ detects more leaks under 32K8W cache in Seed [Tegra 2018]. Intuitively, larger cache associativity means more cache lines per set, which may reduce the eviction risk for an address. However, the tradeoff is that more memory addresses can now map to the same cache set. After studying the bitcode, we find that the in-between instructions of two same-address memory events may directly impact the cache state. Thereby, the results of changed cache associativity values vary against per-program semantics.
Fourth, the timing leaks from *out-of-order* execution are hard to detect though they can appear in various caches. SymO3 finishes 23 of the 24 programs within the 12-hour threshold. One exception is Camellia [Tegra 2018], which has the most significant number of 10,235 memory read events, and SymO3 found no leaks until timeout. The other one is Seed [Tegra 2018], which exposed more than 200 leaks before timeout under 32K8W. Moreover, out of the 144 experiments which test the combination of different cache associativity and RBS, only 30 experiments expose leaks. Among the 3.9+ million traces of all the 144 experiments, the leaks only appear in less than 0.1% of the total traces. This phenomenon explicitly illustrates the elusiveness of the leaks, which also highlights the detection ability of SymO3.

Based on the experimental results, we answer RQ1: within 12 hours, SymO3 is capable of disclosing hundreds of cache timing leaks in 5 out of 24 benchmarks. It also supports different cache settings and different RBS parameter to perform rich comparisons of the results.

### 5.3 The Program Transformation Impact

In section 5.2, we used -O0 as the default compiler optimization option to retain the original program structure as much as possible for leak analysis. However, in practice, people may prefer a higher optimization level in software release for better performance, assuming the aggressive transformation would correctly preserve the software semantics. Recent works studied the performance impact of code transformation [Cadar 2015; Chen et al. 2018; Dong et al. 2015] in symbolic execution. Nevertheless, how the transformation could silently affect the software non-functional properties remain unknown. In this section, we experiment with more options, i.e., -O1, -O2, and -O3, to study how the compiler optimization can roughly affect the SymO3 detection results.

<table>
<thead>
<tr>
<th>Name</th>
<th>Cache</th>
<th>-O1</th>
<th>-O2</th>
<th>-O3</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES-tom</td>
<td>32K4W</td>
<td>46594</td>
<td>186791</td>
<td>720</td>
</tr>
<tr>
<td></td>
<td>32K8W</td>
<td>73694</td>
<td>120633</td>
<td>720</td>
</tr>
<tr>
<td>AES-mbed</td>
<td>32K4W</td>
<td>1</td>
<td>26</td>
<td>720</td>
</tr>
<tr>
<td></td>
<td>32K8W</td>
<td>1</td>
<td>26</td>
<td>720</td>
</tr>
<tr>
<td>DES-gct</td>
<td>32K4W</td>
<td>7902</td>
<td>475800</td>
<td>654.0</td>
</tr>
<tr>
<td></td>
<td>32K8W</td>
<td>14507</td>
<td>439473</td>
<td>720</td>
</tr>
<tr>
<td>FCrypt-chrn</td>
<td>32K4W</td>
<td>11306</td>
<td>165848</td>
<td>263.2</td>
</tr>
<tr>
<td></td>
<td>32K8W</td>
<td>32745</td>
<td>118715</td>
<td>653.4</td>
</tr>
<tr>
<td>Seed-tegra</td>
<td>32K4W</td>
<td>14255</td>
<td>374481</td>
<td>720</td>
</tr>
<tr>
<td></td>
<td>32K8W</td>
<td>55432</td>
<td>216618</td>
<td>720</td>
</tr>
<tr>
<td>encoder-tom</td>
<td>32K4W</td>
<td>6</td>
<td>18</td>
<td>&lt;0.01</td>
</tr>
<tr>
<td></td>
<td>32K8W</td>
<td>6</td>
<td>18</td>
<td>&lt;0.01</td>
</tr>
</tbody>
</table>

Table 4. The experimental results of -O1, -O2, and -O3 options under RBS:32

Table 4, Table 5, and Table 6 present the comprehensive leak detection results performed with three reorder buffer sizes, two cache associativity values, and three compiler optimization levels. We exclude the benchmarks with unchanged leaky results from Table 3, and record the amount of *out-of-order* traces (#.Trace), the times of invoked RNA checks (#.RNA), the total analysis time (Time(m)), and the two types of leaks (#.C/D) for each of the retained benchmark. For brevity, we use alias names to represent the programs and their sources. For example, AES-tom comes from [LibTomCrypt 2019]; AES-mbed belongs to [mbedTLS 2017]; DES-gct is from [Libgcrypt 2018]; FCrypt-chrn reside in [Dellinger et al. 2011]; Seed-tegra corresponds to [Tegra 2018]; and encode-tom exists in [LibTomCrypt 2019].

Fig. 8 visualizes the overall performance variances of higher compiler optimizations against -O0. Let us name the four scatter plots as SP1, SP2, SP3, and SP4, following the left-to-right and top-to-bottom orders. Each scatter plot aggregates the data columns from the above three tables.
Table 5. The experimental results of -O1, -O2, and -O3 options under RBS: 64

<table>
<thead>
<tr>
<th>Name</th>
<th>Cache</th>
<th>-O1</th>
<th>-O2</th>
<th>-O3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#.Trace</td>
<td>#.RNA</td>
<td>Time(m)</td>
<td>C/D</td>
</tr>
<tr>
<td>AES-tom</td>
<td>32K4W</td>
<td>19639</td>
<td>219664</td>
<td>720</td>
</tr>
<tr>
<td></td>
<td>32K8W</td>
<td>52812</td>
<td>295236</td>
<td>720</td>
</tr>
<tr>
<td>AES-mbed</td>
<td>32K4W</td>
<td>1</td>
<td>33</td>
<td>720</td>
</tr>
<tr>
<td>DES-gct</td>
<td>32K4W</td>
<td>1</td>
<td>33</td>
<td>720</td>
</tr>
<tr>
<td>FCrypt-chrn</td>
<td>32K4W</td>
<td>1</td>
<td>33</td>
<td>720</td>
</tr>
<tr>
<td>Seed-tegra</td>
<td>32K4W</td>
<td>1</td>
<td>33</td>
<td>720</td>
</tr>
<tr>
<td>encoder-tom</td>
<td>32K4W</td>
<td>6</td>
<td>18</td>
<td>&lt;0.01</td>
</tr>
</tbody>
</table>

Table 6. The experimental results of -O1, -O2, and -O3 options under RBS: 128

<table>
<thead>
<tr>
<th>Name</th>
<th>Cache</th>
<th>-O1</th>
<th>-O2</th>
<th>-O3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#.Trace</td>
<td>#.RNA</td>
<td>Time(m)</td>
<td>C/D</td>
</tr>
<tr>
<td>AES-tom</td>
<td>32K4W</td>
<td>85009</td>
<td>230585</td>
<td>720</td>
</tr>
<tr>
<td></td>
<td>32K8W</td>
<td>95004</td>
<td>230585</td>
<td>720</td>
</tr>
<tr>
<td>AES-mbed</td>
<td>32K4W</td>
<td>1</td>
<td>41</td>
<td>720</td>
</tr>
<tr>
<td>DES-gct</td>
<td>32K4W</td>
<td>12226</td>
<td>595582</td>
<td>720</td>
</tr>
<tr>
<td>FCrypt-chrn</td>
<td>32K4W</td>
<td>25043</td>
<td>318438</td>
<td>206.1</td>
</tr>
<tr>
<td>Seed-tegra</td>
<td>32K4W</td>
<td>1487</td>
<td>565828</td>
<td>720</td>
</tr>
<tr>
<td>encoder-tom</td>
<td>32K4W</td>
<td>6</td>
<td>18</td>
<td>&lt;0.01</td>
</tr>
</tbody>
</table>

and the corresponding columns in Table 3, where the x-axis shows the running statistics of the -O0 result, and the y-axis states, under the same experimental configurations, the results after compiler optimizations. We use three colors to annotate these optimization options. Thus, each point represents a comparison between a higher optimization result and the -O0 result. A point below the diagonal line indicates a larger -O0 result, and vice versa. For instance, in SP1 which compares the amount of out-of-order traces, the point with the lowest y-axis value means this -O3 analysis generate more traces than the -O0 result. Next, we explain the following findings.

First, the code transformation by compiler optimizations does impact the leak detection. The two extreme cases are AES-mbed and encoder-tom. The -O0 version of the former suffers from leaks under all caches. However, it becomes leak-free after compiler optimization, no matter -O1, -O2, or -O3. By contrast, the non-optimized encoder-tom has no leaks in all caches, but the compiler optimizations introduce a new C-type leak, which appears in all the three experimental tables. Other programs partially have leaks under specific combinations of cache configuration and compiler optimization. For example, in the three data tables, six out of eighteen AES-tom entries endorse the leaks. Moreover, such results of DES-gct and Seed-tegra are 11/18 and 1/18, respectively.

Second, in general, higher compiler optimizations increase the total analysis time on these benchmarks. In Table 3, 93.75% of the analysis finishes in 12 hours. However, within this time threshold, now only 36.1% of the analysis can finish in Table 4, Table 5, and Table 6. In Fig. 8, the SP3 clearly shows this trend where only a small set of the points are below the diagonal line. Also, although the -O0 analysis calls much more times of RNA checks, as depicted in the SP2, we cannot observe a similar distribution in SP1 which compares the generated traces. Furthermore, in SP1, the
The amount of traces (−O0 result)
The traces (Higher compiler optimizations)

- O1
- O2
- O3

The amount of leaks (−O0 result)
The leaks (Higher compiler optimizations)

- O1
- O2
- O3

The invoked checks (−O0 result)
The checks (Higher compiler optimizations)

- O1
- O2
- O3

The analysis time (m)
The −O0 analysis time (m)

- O1
- O2
- O3

The analysis time (Higher compiler optimizations)

Fig. 8. Higher compiler optimization results versus −O0 result.

points above the diagonal line expose farther distances to such line, indicating more traces in the optimized analysis, thus more time needed.

Third, despite the increasing analyzing time, the overall amount of leaks after compiler optimizations decreases. In $SP_1$ which compares the found leaks, we observe that most of the points are below the diagonal line, indicating fewer leaks after optimizations. By reading the bitcode, we confirm the primary reason for this fact is from the reduced number of memory instructions. To be specific, the optimizations try to replace memory visits with register visits as much as possible to reduce the performance latency. Instead of loading and storing some memory values time and again, the compiler utilizes the adequate register slots for faster access. In this way, the compiler eliminates unnecessary memory visits and make the nearby memory events mostly visit different addresses. Then, given the same $RBS$, the memory events residing in the reorder buffer are more likely to have cache mutual effects. Thus, the leak surface shrinks.

Fourth, the C-type leaks now appear in DES−gct and encoder−tom. Recall that SymO$_3$ detects no such C-type leaks in Table 3. The optimization has changed the code layout, making the out-of-order schedules directly cause C-type leaks in the two optimized programs. Considering the limited number of only four C-type leaks, we again learn the rareness of such leaks. However, SymO$_3$ can precisely reveal them.

Now, we answer the RQ2: the compiler optimizations pose significant impacts on the leak detection. In general, the optimizations increase the overall detection time and decrease the total
amount of identified leaks. After optimizations, one benchmark has no leaks, while another non-leaky program suffers from a new C-type leak. In two programs, SYMO₃ even witnesses four C-type leaks, which never appear in Table 3.

5.4 The Compound Analysis with Speculation

Recall that SYMO₃ primarily focuses on the out-of-order execution of independent memory instructions, which considers no branch prediction and the corresponding speculative execution. However, in practice, the speculated memory instructions may also occupy the available reorder buffer slots and participate out-of-order execution. In this section, we conduct a compound analysis, which augments SYMO₃ with a simplified branch prediction, to study the side effects introduced by the speculative execution to the cache.

SpecuSYM [Guo et al. 2020] is a symbolic execution tool that targets cache timing leaks under speculative execution. We adopt the idea of SpecuSYM [Guo et al. 2020] into our SYMO₃ framework. To be specific, we consider two cases at each branch— the right prediction and a misprediction. Upon the right prediction, we allow a limited number of speculated instructions to enter the symbolic trace for out-of-order execution. Regarding the misprediction, after speculation, we will continue the instructions under the right branch, but forbid them to reorder with preceding instructions. In this way, we realized a simplified version of SpecuSYM into SYMO₃ and evaluated its effect upon the original SYMO₃.

Table 7 presents the result of the enhanced SYMO₃ against the original SYMO₃ framework. Let us name the former as SYMO₃⁺ for short. We compile the benchmarks with option -O0 to avoid the interferences of compiler optimizations. Also, we use the same cache parameters for previous experiments for consistency purposes. We set RBS to 64 since it is closer to the practical situation. We also exclude the programs with unchanged results between SYMO₃ and SYMO₃⁺ in terms of the amount and locations of detected leaks. Columns 3–5 show the differential results between SYMO₃ and SYMO₃⁺. The result shows that 6 out of 24 programs now have new leaks. Furthermore, we analyze the results in more detail as follows.

Overall, SYMO₃⁺ shows an increasing trend in the execution time and the out-of-order trace amount, due to the addition of speculated events into the symbolic execution paths. Moreover, the number of leaky programs and their leaks also increase, indicating that speculative execution indeed exacerbates the leak situations.

Note that SYMO₃ detected leaks in five programs shown in Table 3. By comparison, SYMO₃⁺ detected new leaks in six programs, including the AES [LibTomCrypt 2019], which was also reported by SYMO₃. The two programs that SYMO₃⁺ identified no leak, but SYMO₃ detected leaks are
ulong32 Tks0[], Tks1[], Tks2[], Tks3[]; //static array
...
ulong32 temp;
ulong32* rk = skey->rijndael.eK; //skey->rijndael.eK: private data
ulong32* rrk = skey->rijndael.eK + (28 + keylen) - 4;
...
ulong32 temp = rrk[3];
rk[3] = Tks0[byte(temp, 3)] ∧ Tks1[byte(temp, 2)] ∧ Tks2[byte(temp, 1)] ∧ Tks3[byte(temp, 0)]; //leaking site

Fig. 9. The code snippet and cache mappings in AES [LibTomCrypt 2019].

DES [LibTomCrypt 2019] and Seed [Tegra 2018]. By contrast, another two DES programs, DES [OpenSSL 2019] and DES [glibc 2019], now have leaks in SymO₃⁺. This fact shows that the latter two DES implementations are more vulnerable due to the speculated events, while out-of-order execution can impact the former two programs.

The most significant differences are from AES [LibTomCrypt 2019] and chacha20 [LibTomCrypt 2019]. Based on the significant amount of increased traces, SymO₃⁺ identifies 64 and 66 more leaks in them, respectively. Also, all chacha20 [LibTomCrypt 2019] leaks are C-type leaks. The blowfish [LibTomCrypt 2019] merely has two more D-type leaks under two caches, despite the increased 109,926 traces. The encoder [LibTomCrypt 2019] has six new traces and six new leaks in both caches. These four benchmarks show that speculative execution can give rise to both new out-of-order behaviors and new timing leaks.

The rest two DES benchmarks present different results. Though speculative execution raises no new out-of-order traces, it brings both C-type and D-type leaks, to DES [OpenSSL 2019] and DES [glibc 2019]. Recall that SymO₃ reports no timing leaks in these benchmarks in Table 3; this situation means speculative execution can undoubtedly change the cache state by interfering with other memory events.

Based on the above experimental findings, we now answer RQ3: SymO₃ can support speculative execution in its dynamic analysis. The experimental results show that the modeled speculative execution increases new out-of-order behaviors and raises new timing leaks compared to the original SymO₃ results in Table 3.

5.5 The Case Study
SymO₃ develops an approach that not only pinpoints the problematic memory instructions but also generates concrete inputs and specific execution orders to manifest the leaks. For illustration,
we exemplify the results of SymO$_3$ by studying a leak in AES [LibTomCrypt 2019] with the -O0 compilation option based on RBS:32 and a 32K4W cache.

Fig. 9 extracts the code snippet of setting up the forward key in AES [LibTomCrypt 2019]. The four Tks arrays are constant S-Boxes while skey->rijandael.eK is the private data from the user input. In normal in-order execution, this program does not manifest any leak at line 1359 as the cache is always-hit no matter what the private data is. However, the cache timing leaks could emerge when the out-of-order execution takes effect. At line 1354, the algorithm assigns the variable temp with 4-byte private data symbolized by SymO$_3$. Among lines 1356-1359, the four S-Boxes and the temp array are independently accessed.

Thus, the execution order of these load instructions might be shuffled by out-of-order execution to some extent, depending on which one gets ready faster in real execution. In this study, we assume that each instruction has the same chance to be scheduled beforehand. For example, the independent memory load events within lines 1356-1359 may execute in various orders.

Take the code Tks0[byte(temp, 3)] at line 1356 as the example, which can break into two memory load events. We name the events as 1356L1 and 1356L2. The first load, 1356L1, reads the 1-byte data of address temp+3, while the second load, 1356L2, uses the retrieved 1-byte value as the index to read Tks0. The code from lines 1357 to 1359 has the same patterns, and we similarly name their memory events. Another example, the rk[3]= at line 1355, consists of one load event of rk and one store event to rk+3. We name them as 1355L1 and 1355S1. Thereby, an ideal in-order event sequence would faithfully follow the program order to be "1356L1-1356L2-1357L1-1357L2-1358L1-1358L2-1359L1-1359L2-1355L1-1355S1".

As analyzed by SymO$_3$, when the code from 1356 to 1359 executes in the order of "1356L1-1356L2-1357L1-1358L1-1358L2-1359L1", a leak appears at 1359L1. The reason for this leak is because of the beforehand executions of 1357L1, 1358L1, and 1355L1. As displayed in Figure 9, in the in-order execution, 1356L1 loads the whole 4-byte temp data into the cache. The following reading operations of arrays Tks0, Tks1, and Tks2 would not evict the recently used data temp from the cache; thus, they load data into the remaining three lines of the set.

In this case, event 1359L1, which corresponds to byte(temp, 0), would have cache always-hit since temp is always in the cache at the time executing 1359L1. By contrast, in the reported new execution order, 1358L1 executes before 1356L2, where the subsequent four events, i.e., 1356L2-1357L2-1358L2-1355L1) jointly occupy the cache lines. Now, when reaching the last event 1359L1, temp may no longer stay in the cache and make 1359L1 cause a miss. By constraint solving, SymO$_3$ confirms that when the private data turns to be 0x105dc88cfa5f8739d59f3f6b67aa4a, 1359L1 misses the cache and triggers a leak.

6 THREATS TO VALIDITY

SymO$_3$ does not consider aggressive memory execution schemes like store-to-load forwarding [Sha et al. 2005], memory disambiguation [Önder and Gupta 1999], and memory dependence speculation [Moshovos and Sohi 2000]. Also, we assume that the memory model has sequential consistency. Taking these features into consideration needs further investigation.

SymO$_3$ takes the LLVM [Lattner and Adve 2004] IR to perform the symbolic analysis. As certain optimizations could be enabled when generating the machine code from the IR; the instruction flow that our analysis follows may vary from the actual flow in the executable. The inconsistency could cause inaccuracy of the analysis. An alternative is to decompile the machine code into the IR [Cha et al. 2012; KLEE-Native 2019; Saudel and Salwan 2013; Stephens et al. 2016] for further analysis. However, the accuracy loss is inevitable. The balance between the low-level precise data flow and the high-level program semantics still deserves an in-depth study.
Another threat lies with the choice of reorder buffer size for out-of-order execution. As different architectures are configured with different settings, the constant window parameter may lead to the imprecision of analysis. Lastly, SymO3 does not offer the capability of analyzing multi-threaded programs, and supporting both out-of-order execution and concurrency results in prohibitively huge state space, which requires further research in future work.

7 RELATED WORK
Recent side-channel based attacks [Bulck et al. 2018; Guarnieri et al. 2020; Lipp et al. 2018], originated from the out-of-order execution, have received tremendous attention from both industry and academia. The side effects of out-of-order execution have been studied in the real-time system domain since the 1990s. A real-time system program should guarantee to finish within a specified time-bound, and the out-of-order execution plays a vital role in program execution time estimation.

Antonie et al. [Colin and Puaut 2000] presented the worst-case execution time (a.k.a. WCET) analysis techniques for processors with branch prediction. Li et al. [Li et al. 2006] proposed an abstract model for WCET estimation for the out-of-order superscalar pipelines. Some commercial tools for WCET estimation, such as aiT [Wilhelm et al. 2010], also take into account of out-of-order pipelines and caches during the analysis. Recently, Wu et al. [Wu and Wang 2019] developed abstract analysis for cache state under speculative execution, which can be applied to both WCET estimation and side-channel leak detection.

In out-of-order execution analysis, various static analysis techniques have been adopted, e.g., the discussed timing analysis [Colin and Puaut 2000; Wu and Wang 2019] on out-of-order pipeline and cache. Besides, [Lahiri et al. 2002; Skakkebøk et al. 1998] applied formal methods to prove the correctness of out-of-order execution. The processor simulation tools [Burger and Austin 1997; Pai et al. 1997; Schnarr and Larus 1998] also modeled out-of-order execution, but they mainly focused on the precision and performance of the simulation.


Still, none of them takes the out-of-order execution into account. Recent work [Guo et al. 2020; Wu and Wang 2019] have proposed static and dynamic approaches in modeling microarchitectural speculative execution at the software level. However, they addressed the branch prediction problems relevant to speculative execution while this work concentrates on the finer-granularity instruction-level out-of-order scheduling.

8 CONCLUSION
In this paper, we extended the applicability of symbolic execution to out-of-order execution for cache timing leak detection. By developing new modeling, reduction, and analysis components, we proposed the SymO3 framework, which successfully identified cache timing leaks in real-world cipher programs. Although the primary goal here is to discover the cache timing side-channel leaks, the technical attempts made to tailor symbolic execution for out-of-order execution is general. We look forward to adopting the techniques for other analysis scenarios.

ACKNOWLEDGEMENTS
This work was partially supported by the National Science Foundation of China (No. 61932021 61802168, and 61972290), and the Natural Science Foundation of Jiangsu Province (No. BK20191247).
REFERENCES


