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 SymO$_3$, for exposing cache timing leaks under the context of *out-of-order* execution. SymO$_3$ 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 SymO$_3$ upon KLEE and conducted three evaluations on it. Experimental results show that SymO$_3$ successfully uncovers a set of cache timing leaks in five real-world programs. Also, SymO$_3$ finds that, in general, program transformation from compiler optimizations shrink the surface to timing leaks. Furthermore, augmented with a speculative execution modeling, SymO$_3$ 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

ACM Reference Format:
Shengjian Guo, Yueqi Chen, Jiyong Yu, Meng Wu, Zhiqiang Zuo, Peng Li, Yueqiang Cheng, and Huibo Wang. 2020. Exposing Cache Timing Side-Channel Leaks through Out-of-Order Symbolic Execution. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 147 (November 2020), 32 pages. https://doi.org/10.1145/3428215

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.

Authors' addresses: Shengjian Guo, Baidu Security, USA, sjguo@baidu.com; Yueqi Chen, Pennsylvania State University, USA, ychen@ist.psu.edu; Jiyong Yu, University of Illinois at Urbana-Champaign, USA, jiyongy2@illinois.edu; Meng Wu, Ant Group, China, bode.wm@antfin.com; Zhiqiang Zuo, State Key Lab. for Novel Software Technology, Nanjing University, China, zqzuo@nju.edu.cn; Peng Li, Baidu Security, USA, lipeng28@baidu.com; Yueqiang Cheng, Baidu Security, USA, chengyueqiang@baidu.com; Huibo Wang, Baidu Security, USA, wanghuibo01@baidu.com.

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.

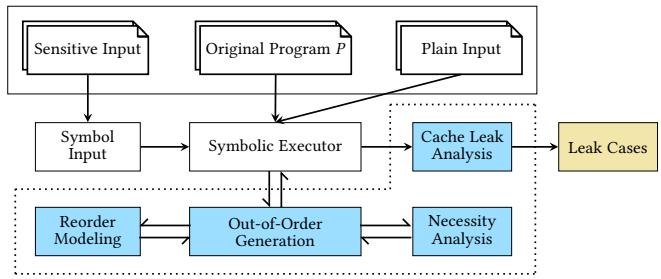


Fig. 1. The overall flow of the SymO$_3$ method.

Towards this end, we propose SymO$_3$, 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 SymO$_3$, 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 SymO$_3$ framework. To the best of our knowledge, SymO$_3$ 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 SymO$_3$. Given a program P, which is leakage-free in in-order execution, the sensitive data input, and the non-sensitive plain input, SymO$_3$ 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, SymO$_3$ integrates the constraint-solving based leak analysis to perform the timing leak detection under the *out-of-order* scenarios. Finally, SymO$_3$ 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 SymO$_3$ 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 SymO$_3$ 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 $n \in [1, 256]$.

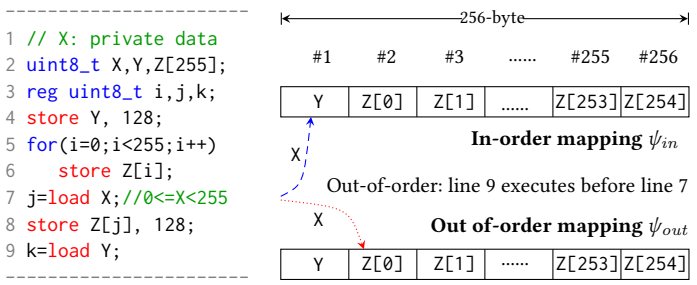
Fig. 2(b) shows how the variables map to cache C when running P in the in-order and *out-of-order* manner, respectively. Look into the in-order mapping ψ_{in} first. Initially, we have an empty cache C. On executing P, the first memory write (line 4) associate cache line #1 with `Y`. Next, the *for* 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



(a) The program P (b) The cache mappings ψ_{in} and ψ_{out}

Fig. 2. Program P and the runtime cache mappings.

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 ψ_{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 $X \neq 0$ 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 δ entries, which indicates at most δ 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 δ, then the two instructions cannot simultaneously involve in the *out-of-order* execution. Here we use δ=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`

Table 1. The cache behaviors of three memory operations under *out-of-order* execution.

Scenario	`load X`	`store Z[j],128`	`load Y`	Result
In-order	miss	hit	miss	no-leak
9↷8	miss	hit/miss	miss	leak
9↷7	miss	hit/miss	hit	leak
9↷6	miss	hit/miss	hit	leak
7↷6	miss	hit	miss	no-leak
7↷6↷9	miss	hit/miss	miss	leak
9↷6↷7↷6	miss	hit/miss	hit	leak
7↷6↷9↷6	miss	hit/miss	hit	leak

`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.

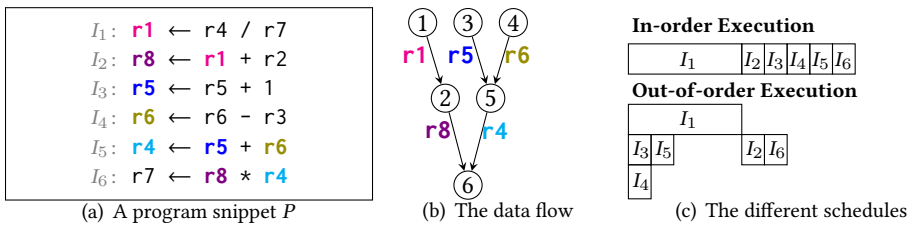
Table 1 summarizes the possible *out-of-order* execution scenarios and the resulting cache behaviors. Operator ↷ means its left-side operand executes before the right-side operand, thus forming an *out-of-order* situation. For example, 9↷8 indicates line 9 executes right before line 8, while 9↷6 implies 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 SymO$_3$ work.

3 PRELIMINARIES

This section presents the preliminary knowledge of our work.



(a) A program snippet P (b) The data flow (c) The different schedules

Fig. 3. High-level view of the program execution.

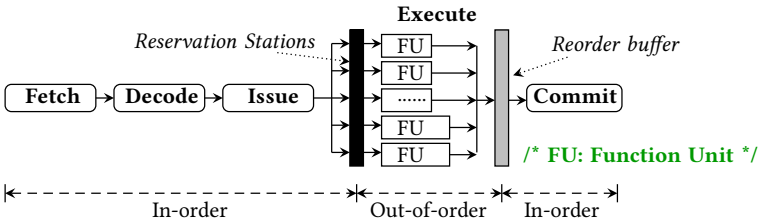


Fig. 4. Microarchitectural view of the instruction execution.

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∈[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 `r1`. 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 SymO$_3$ 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 KLEESpectre [Wang et al. 2020] model microarchitectural speculative execution for side-channel leak pinpointing and vulnerability detection, which closely relate to SymO$_3$. However, SymO$_3$ has threefold distinctions.

The *speculative execution* studied in [Guo et al. 2020; Wang et al. 2020] and the *out-of-order execution* studied in SymO$_3$ 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 SymO$_3$. *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 KLEESpectre, despite all these methods share the fundamental state-forking mechanism from KLEE. Also, SpecuSym and KLEESpectre developed forward modeling techniques while in contrast, SymO$_3$ relies on backward modeling and prunes unnecessary forward executions. However, SymO$_3$ cannot model the speculative behaviors in SpecuSym and KLEESpectre, and vice versa.

Moreover, SymO$_3$ 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. SymO$_3$ 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 expolsion 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

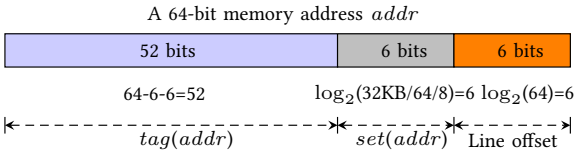


Fig. 5. The tag id, associated set and line offset of *addr* under a 32KB 8-way cache.

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.
- $set(addr)$ gives the cache set with which $addr$ associates.
- $tag(addr)$ 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 32KB/64/8=64. Thus $set(addr)$ returns the value of the middle 6 bits. Then, the remaining 52 bits belong to the tag id, which can be obtained by $tag(addr)$.

Then, we define a function ϕ to formally check whether two addresses $addr_1$ and $addr_2$ could map to the same cache line:

$$\phi\,(addr_1, addr_2) := \; set(addr_1) = set(addr_2) \;\wedge\; tag(addr_1) = tag(addr_2) \tag{1}$$

Note that ϕ 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 ϕ 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 τ and τ', their execution traces must have no distinguishable physical effects against a specific microarchitecture component:

$$\psi := \forall t, \tau, \tau' \,.\, Obs\;(Imp_\mu(P(t,\tau)) = Obs\;(Imp_\mu(P(t,\tau')) \tag{2}$$

Equation (2) characterizes the general abstraction of the side-channel-free property. Function Obs evaluates the external adversary's observation ability on the physical effects. Imp_μ 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 Imp_μ 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 $in := (t, \tau)$, and let $T(P(t, \tau))$ denotes the execution time of an execution trace on in, then the observation Obs against cache timing can be approximated to the observed execution time of that trace upon input in:

$$\forall t, \tau \ . \ Obs\ (Imp_\mu(P(t, \tau)) \approx \forall t, \tau \ . \ Obs_{timing}(P(t, \tau)) \tag{3}$$

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:

$$\psi_{in\_order} := \forall t, \tau, \tau' \ . \ Obs_{timing}(P(t, \tau)) = Obs_{timing}(P(t, \tau')) \tag{4}$$

Still, τ and τ' 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:

$$\psi_{out\_order} := \psi_{in\_order} \wedge \left(\exists t, \tau, \tau' \ . \ Obs_{timing}(P_{out\_order}(t, \tau)) \neq Obs_{timing}(P_{out\_order}(t, \tau'))\right) \tag{5}$$

A new leak appears under the *out-of-order* execution context if two secret inputs τ and τ' do exist and cause distinguishable Obs_{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 need 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 SymO$_3$ 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

2010; 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 SymO$_3$.

First, we assume that a program P comprises a finite set of paths, and each path contains a finite sequence of instructions. Let *inst* be short for an instruction then $\epsilon{:=}\langle l, inst, l', nxt\rangle$ denotes the symbolic event of an *inst*. l and l' represents the program locations before and after *inst*, and nxt points to the next symbolic event for execution. The execution of P on input $in{:=}\{t, \tau\}$ explores a series of symbolic events along the program path to which *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 $s{:=}\langle \epsilon, pcon, \hat{m}\rangle$. ϵ is the current symbolic event to execute at s; $pcon$ 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 ϵ.

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 `v=load addr`, or a memory *write* as `store addr val`, where *addr* is a memory address, and `val` denotes a symbolic value. We use 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 St; start StdSymExec with a seed state s_0 on input {t_0, τ_0}.
2:  procedure StdSymExec (Symbolic_State state)
3:      St.push(state);
4:      if state.ϵ is a M-event then                    ▷ The memory event interpretation
5:          O3Gen(state);                               ▷ The entry of new SymO3 analysis
6:          StdSymExec(NextSymState (state));
7:      else
8:          StdSymExec(NextSymState (state));           ▷ Other events execute as usual
9:      end if
10:     St.pop();
11: end procedure
12:
13: function O3Gen (Symbolic_State crt)
14:     crt.done ← crt.done ∪ crt.ϵ;
15:     if crt.ϵ is a memory read event then
16:         Let e be the closest event in crt.trace that RNA(crt, e) returns true;   ▷ RNA analysis
17:         if e ≠ null then
18:             prv ← the symbolic state that was about to execute e;
19:             if crt.ϵ ∉ prv.done then
20:                 ReorderModeling (crt, prv);         ▷ Invoke the out-of-order modeling
21:             end if
22:         end if
23:     end if
24:     crt.trace ← crt.trace · crt.ϵ;
25: end function
```

In this section, we introduce the main procedure of SymO$_3$ 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$_3$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$_3$Gen, we extend the symbolic state s to be $s{:=}\langle \epsilon, pcon, \hat{m}, trace, done\rangle$. 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$_3$Gen first updates the $done$ set of state crt (line 14), meaning that event $crt.\epsilon$ has been processed in this backtracking, to avoid future re-analysis. Then, if $crt.\epsilon$ is a memory read event, O$_3$Gen backwardly searches $crt.trace$ for the closest event e that may run *out-of-order* with $crt.\epsilon$, and the *Reorder Necessity Analysis* (RNA, ref. Section 4.3) can return $true$ (line 16). If such e does not exist, O$_3$Gen appends $crt.\epsilon$ to $crt.trace$ (line 24) and exits.

In contrast, if the desired event e does exist, then at line 18, O$_3$Gen retrieves a prior symbolic state prv who was about to execute e. More importantly, if $crt.\epsilon$ is not in the $done$ set of state prv, O$_3$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$_3$Gen also updates $crt.trace$ at line 24.

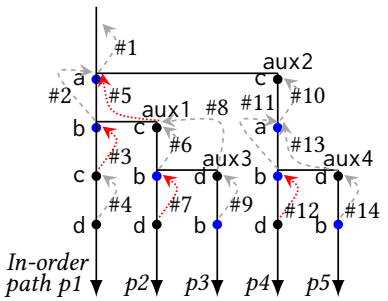


Fig. 6. The *out-of-order* generation and the preserved *four* new states

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 O_3Gen, 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 O_3Gen #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 O_3Gen 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, O_3Gen #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, O_3Gen #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. O_3Gen #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, O_3Gen 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 $b \notin$ 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* $\{e_1, e_2, ..., e_m\}$. *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 $P_1, P_2, ..., P_{X_n}$. X_n 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 e_{n+1} to an event sequence P from $P_1, P_2, ..., P_{X_n}$ 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 e_{n+1} is younger than $e_1, e_2, ..., e_n$ 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 $\{P_1, P_2, ..., P_{X_n}\}$.

(4) To construct all possible execution sequences consisting of $e_1, e_2, ..., e_{n+1}$, we start by appending e_{n+1} to every sequence $P \in \{P_1, P_2, ..., P_{X_n}\}$, and do O_3GEN on the final state of P for e_{n+1} based on Algorithm 1. O_3GEN 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_3GEN for all possible sequences with n instructions $(e_1, e_2, ..., e_n)$ 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.ϵ 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.ϵ and e are unreachable in G then      ▷ The dependency check
6:              if crt.ϵ 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 $crt.\epsilon$ in the program order.

Then, if $d \notin$ (0,`rbs`), RNA directly returns $false$ at line 11, meaning that *out-of-order* execution of $crt.\epsilon$ 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 $crt.\epsilon$ 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., $crt.\epsilon$ 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 $crt.\epsilon$ cannot impact each other regarding the cache status, RNA returns $false$. If not, we deem the necessity of *out-of-order* modeling and let RNA return $true$.

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 `load X` and `load Y` be *two* (including the former). Thus, $d \in$ (0,`rbs`) satisfies because `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 $true$.

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 $false$.

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

Algorithm 3 The Reorder Modeling of Two Symbolic Events.

```
1:  function ReorderModeling (Symbolic_State crt, Symbolic_State prv)
2:      aux ← fork a new symbolic state from prv and put it to state pool;
3:      e ← declare an event variable that points to aux.ϵ;
4:      q1, q2 ← declare two empty queues for symbolic events;
5:      while e ≤ crt.ϵ do
6:          if e is dependent of aux.ϵ then
7:              q1.back().nxt = e;
8:              q1.push(e);
9:          else
10:             q2.back().nxt = e;
11:             q2.push(e);
12:         end if
13:         e ← the next event along the execution path to crt;
14:     end while
15:     aux.ϵ ← q2.front();                    ▷ Update the current event of aux
16:     q2.back().nxt = q1.front();            ▷ Concatenate two event lists
17:     q1.back().nxt = crt.ϵ.nxt;             ▷ Redirect to the next event
18:     Clean temporary queues q1 and q2;
19: end function
```

At the beginning, ReorderModeling forks a new auxiliary state aux from its second input prv (line 4), to act as the modeling transmitter. After the duplication, aux has the same state snapshot

to prv. Then, at lines 3-4, we define a temporary symbolic event e that points to $aux.\epsilon$, and two empty queues q_1 and q_2. Next, ReorderModeling iteratively checks the dependency of e and $aux.\epsilon$ (line 6), and updates e to the next event (lines 13) along the trajectory between prv and crt, until e reaches $crt.\epsilon$ 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 $aux.\epsilon$ (line 6), we placed it to q_1 (line 8). Similarly, we put events that are irrelevant to $aux.\epsilon$ 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 $aux.\epsilon$ 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 $aux.\epsilon$. 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 $crt.\epsilon$ (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.

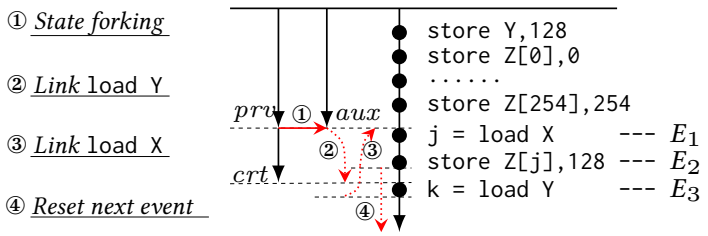


Fig. 7. *Out-of-order* modeling of events E_1 and E_3

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, $aux.\epsilon$ 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 $aux.\epsilon$ 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 $crt.\epsilon$, 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:
11: function CacheLeakAnalysis (Symbolic_State crt)
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 O_3Gen on a memory event $state.\epsilon$, we check whether it may lead to a cache timing leak. That is, before symbolically executing $state.\epsilon$, we analyze if, under some specific inputs, the cache behavior of $state.\epsilon$ differs from its original behavior under in-order execution. SymO_3 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 $crt.\epsilon$ is relevant to the sensitive input (line 12), we build the constraint σ that describes the leak condition of $crt.\epsilon$ (line 13), and solve $\sigma \wedge pcon$ 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:

$$\sigma := \exists in.\left(\alpha_{in}(e_i) \neq \Delta_{e_i}\right) \tag{6}$$

To be specific, given a memory event e_i, if there exists an input in that makes $\alpha_{in}(e_i)$, the cache hit constraint of e_i under *out-of-order* execution, compute a different value from the in-order cache behavior Δ_{e_i}, then there is a leak at e_i. Here Δ_{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 $\alpha_{in}(e_i)$, we need two more notations:

- The *trace* constituent of a symbolic state consists of the executed memory events $\{e_1, ..., e_n\}$ 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 $\alpha_{in}(e_i)$ as:

$$\alpha_{in}(e_i) := \left(\exists x \in [0, i) \;\middle|\; \phi\left(a_x, a_i\right)\right) \wedge \left(\forall y \in (x, i) \;\middle|\; \neg\phi\left(a_x, a_y\right)\right) \wedge \sum_{z=x+1}^{i-1} \left(set(a_z) = set(a_i)\right) < \mathsf{K} \tag{7}$$

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 ϕ (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_i. More importantly, such a number has to be less than the cache associativity K. This sub-constraint is to prevent from evicting 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 SymO$_3$ with a set of real-world benchmarks. We implement SymO$_3$ on the latest KLEE 2.0 [Cadar et al. 2008] and LLVM 6.0 [Lattner and Adve 2004]. SymO$_3$ 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 SymO$_3$. 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 SymO$_3$ against these benchmarks, we design the following three research questions:

- **RQ1**: Is SymO$_3$ 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 SymO$_3$ detection result?
- **RQ3**: Is SymO$_3$ 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

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.

Name	Source	LoC	#.MemV	#.In	Name	Source	LoC	#.MemV	#.In
AES	[LibTomCrypt 2019]	1,838	898	16	fixfrac	[libfixedtimefixedpoint 2017]	63	28	59
AES	[mbedTLS 2017]	281	245	272	hash	[Rapier and Bennett 2008]	320	5297	64
adpcm	[Gustafsson et al. 2010]	916	426	24	keyvalue	[GDK 2018]	62	18	4
blowfish	[LibTomCrypt 2019]	467	321	8	lblock	[Dinu et al. 2015]	949	1233	10
Camellia	[Tegra 2018]	1,324	10235	16	ocb	[LibTomCrypt 2019]	377	260	28
chacha20	[LibTomCrypt 2019]	776	6870	36	PRESENT	[Dinu et al. 2015]	215	57	19
chaskey	[Dinu et al. 2015]	255	192	32	Salsa	[Tegra 2018]	279	481	28
DES	[glibc 2019]	547	687	16	Seed	[Tegra 2018]	487	1753	16
DES	[Libgcrypt 2018]	337	984	8	str2key	[OpenSSL 2019]	371	89	16
ecc	[FourQLib 2020]	224	251	320	stc	[Lee et al. 1997]	494	242	1024
encoder	[LibTomCrypt 2019]	134	37	100	trie	[freeradius 2020]	162	51	32
FCrypt	[Dellinger et al. 2011]	621	798	12	unicode	[GDK 2018]	839	22	4

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, SymO$_3$ 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, SymO$_3$ 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, SymO$_3$ analyzes the traces efficiently, thus costing tolerable time overhead. On the other hand, many traces cannot attribute to leaks, and SymO$_3$ 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 3. The leak detection results under three *RBS* settings

Name	Cache	*RBS*:32			*RBS*:64			*RBS*:128		
		Time (m)	#.Trace	#.C/D	Time (m)	#.Trace	#.C/D	Time (m)	#.Trace	#.C/D
AES[LibTomCrypt 2019]	32K4W	87.03	54837	0/36	105.02	96232	0/42	101.30	102304	0/42
	32K8W	180.33	68114	0/31	208.11	115546	0/33	245.15	128867	0/33
AES[mbedTLS 2017]	32K4W	202.93	100658	0/83	365.68	106940	0/85	363.76	110803	0/91
	32K8W	125.79	59846	0/85	232.14	89211	0/85	221.70	93237	0/91
adpcm[Gustafsson et al. 2010]	32K4W	24.88	49093	0/0	30.83	67902	0/0	32.17	85562	0/0
	32K8W	20.64	25156	0/0	29.01	33888	0/0	30.19	45758	0/0
chaskey[Dinu et al. 2015]	32K4W	< 0.01	1	0/0	< 0.01	1	0/0	< 0.01	1	0/0
	32K8W	< 0.01	1	0/0	< 0.01	1	0/0	< 0.01	1	0/0
blowfish[LibTomCrypt 2019]	32K4W	10.47	18470	0/0	11.32	21169	0/0	12.08	22369	0/0
	32K8W	10.73	18470	0/0	10.81	21169	0/0	11.74	22369	0/0
Camellia[Tegra 2018]	32K4W	720	17084	0/0	720	33686	0/0	720	39870	0/0
	32K8W	720	24603	0/0	720	46205	0/0	720	52830	0/0
chacha20[LibTomCrypt 2019]	32K4W	0.48	1	0/0	0.51	1	0/0	0.83	1	0/0
	32K8W	0.48	1	0/0	0.51	1	0/0	0.77	1	0/0
DES[OpenSSL 2019]	32K4W	189.94	1	0/0	208.08	1	0/0	204.17	1	0/0
	32K8W	158.94	1	0/0	197.29	1	0/0	254.50	1	0/0
DES[glibc 2019]	32K4W	1.17	1	0/0	1.62	1	0/0	2.10	1	0/0
	32K8W	1.18	1	0/0	1.26	1	0/0	2.07	1	0/0
DES[Libgcrypt 2018]	32K4W	34.30	1459	0/120	39.87	2563	0/120	52.02	2962	0/120
	32K8W	62.83	1459	0/120	73.34	2563	0/120	74.83	2962	0/120
ecc[FourQLib 2020]	32K4W	< 0.01	1	0/0	< 0.01	1	0/0	< 0.01	1	0/0
	32K8W	< 0.01	1	0/0	< 0.01	1	0/0	< 0.01	1	0/0
encoder[LibTomCrypt 2019]	32K4W	< 0.01	1	0/0	< 0.01	1	0/0	< 0.01	1	0/0
	32K8W	< 0.01	1	0/0	< 0.01	1	0/0	< 0.01	1	0/0
FCrypt[Dellinger et al. 2011]	32K4W	327.45	49252	0/92	525.39	50722	0/92	558.74	51304	0/92
	32K8W	238.05	34964	0/92	389.18	42909	0/96	464.43	43803	0/96
fixfrac[libfixedtimefixedpoint 2017]	32K4W	0.02	1	0/0	0.02	1	0/0	0.02	1	0/0
	32K8W	< 0.01	1	0/0	0.02	1	0/0	0.02	1	0/0
hash[Rapier and Bennett 2008]	32K4W	19.45	1	0/0	25.27	1	0/0	35.30	1	0/0
	32K8W	26.32	1	0/0	32.06	1	0/0	42.89	1	0/0
keyvalue[GDK 2018]	32K4W	< 0.01	1	0/0	< 0.01	1	0/0	< 0.01	1	0/0
	32K8W	< 0.01	1	0/0	< 0.01	1	0/0	< 0.01	1	0/0
lblock[Dinu et al. 2015]	32K4W	600.94	11078	0/0	720	11078	0/0	698.04	11078	0/0
	32K8W	1.11	1	0/0	1.44	1	0/0	1.24	1	0/0
ocb[LibTomCrypt 2019]	32K4W	< 0.01	1	0/0	< 0.01	1	0/0	< 0.01	1	0/0
	32K8W	< 0.01	1	0/0	< 0.01	1	0/0	< 0.01	1	0/0
PRESENT[Dinu et al. 2015]	32K4W	1.06	1	0/0	1.10	1	0/0	1.10	1	0/0
	32K8W	0.68	1	0/0	1.57	1	0/0	0.68	1	0/0
Salsa[Tegra 2018]	32K4W	0.04	1	0/0	0.05	1	0/0	0.08	1	0/0
	32K8W	0.04	1	0/0	0.05	1	0/0	0.09	1	0/0
Seed[Tegra 2018]	32K4W	117.42	55109	0/207	138.12	86954	0/210	149.86	132902	0/214
	32K8W	720	85365	0/219	720	163266	0/237	720	223443	0/242
stc[Lee et al. 1997]	32K4W	< 0.01	1	0/0	< 0.01	1	0/0	< 0.01	1	0/0
	32K8W	< 0.01	1	0/0	< 0.01	1	0/0	< 0.01	1	0/0
trie[freeradius 2020]	32K4W	< 0.01	1	0/0	0.01	1	0/0	< 0.01	1	0/0
	32K8W	< 0.01	1	0/0	0.01	1	0/0	< 0.01	1	0/0
unicode[GDK 2018]	32K4W	11.18	162318	0/0	22.01	222578	0/0	34.64	290581	0/0
	32K8W	7.08	112651	0/0	8.66	147254	0/0	15.25	172955	0/0

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, SymO$_3$ 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. SymO$_3$ 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 SymO$_3$ 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 SymO$_3$.

Based on the experimental results, we answer **RQ1**: within 12 hours, SymO$_3$ 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 SymO$_3$ detection results.

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

Name	Cache	-O1				-O2				-O3			
		#.Trace	#.RNA	Time(m)	#.C/D	#.Trace	#.RNA	Time(m)	#.C/D	#.Trace	Time(m)	#.RNA	#.C/D
`AES-tom`	32K4W	46594	186791	720	0/28	14755	290485	567.5	0/22	19269	523.2	245214	0/58
	32K8W	73694	120633	720	0/0	47331	158063	720	0/0	52620	720	127040	0/0
`AES-mbed`	32K4W	1	26	720	0/0	1	26	720	0/0	1	720	26	0/0
	32K4W	1	26	720	0/0	1	26	720	0/0	1	720	26	0/0
`DES-gct`	32K4W	7902	478500	654.0	0/159	5652	472748	691.3	0/145	11402	720	475748	0/145
	32K4W	14567	439473	720	0/0	17156	430778	720	0/0	17356	720	440828	0/128
`FCrypt-chrn`	32K4W	11306	165848	263.2	0/62	10255	60418	377.9	0/63	10255	368.9	60418	0/63
	32K4W	32745	118715	653.4	0/80	15463	34988	613.1	0/63	15463	609.6	34988	0/63
`Seed-tegra`	32K4W	14255	374481	720	0/143	37971	284301	720	0/0	31761	720	291174	0/0
	32K4W	55432	216618	720	0/0	78245	121536	720	0/0	59786	720	174221	0/0
`encoder-tom`	32K4W	6	18	<0.01	1/0	6	18	<0.01	1/0	6	<0.01	18	1/0
	32K4W	6	18	<0.01	1/0	6	18	<0.01	1/0	6	<0.01	18	1/0

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 SP_1, SP_2, SP_3, and SP_4, 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**

Name	Cache	-O1				-O2				-O3			
		#.Trace	#.RNA	Time(m)	#.C/D	#.Trace	#.RNA	Time(m)	#.C/D	#.Trace	#.RNA	Time(m)	#.C/D
AES-tom	32K4W	19639	219664	720	0/0	1	23	720	0/0	23002	290521	720	0/57
	32K8W	52812	293236	720	0/0	1	23	720	0/0	72797	164890	720	0/0
AES-mbed	32K4W	1	33	720	0/0	1	33	720	0/0	1	33	720	0/0
	32K4W	1	33	720	0/0	1	33	720	0/0	1	33	720	0/0
DES-gct	32K4W	11402	546435	661.6	0/158	23924	531729	720	0/167	4064	537522	720	3/127
	32K4W	27044	473760	720	0/0	12064	487022	720	0/0	17441	503102	720	0/0
FCrypt-chrn	32K4W	17428	243286	258.3	0/62	10255	63754	403.6	0/63	10255	63754	367.4	0/63
	32K4W	43021	167628	612.9	0/80	16256	36353	720	0/63	16256	36353	720	0/63
Seed-tegra	32K4W	23874	419281	720	0/0	35682	375927	720	0/0	33768	370319	720	0/0
	32K4W	71027	226922	720	0/0	60651	289303	720	0/0	87724	148155	720	0/0
encoder-tom	32K4W	6	18	<0.01	1/0	6	18	<0.01	1/0	6	18	<0.01	1/0
	32K4W	6	18	<0.01	1/0	6	18	<0.01	1/0	6	18	<0.01	1/0

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

Name	Cache	-O1				-O2				-O3			
		#.Trace	#.RNA	Time(m)	#.C/D	#.Trace	#.RNA	Time(m)	#.C/D	#.Trace	#.RNA	Time(m)	#.C/D
AES-tom	32K4W	85009	230585	720	0/32	1	23	720	0/0	18775	358836	720	0/52
	32K8W	95405	203619	720	0/0	1	23	720	0/0	82189	89786	720	0/0
AES-mbed	32K4W	1	41	720	0/0	1	41	720	0/0	1	41	720	0/0
	32K4W	1	41	720	0/0	1	41	720	0/0	1	41	720	0/0
DES-gct	32K4W	12226	595582	720	0/149	16966	587557	720	0/146	11392	588247	720	0/144
	32K4W	28526	532193	720	0/144	34694	500157	720	0/0	46713	480691	720	0/0
FCrypt-chrn	32K4W	25045	318438	206.1	0/69	10255	68668	403.1	0/63	10255	68668	413.7	0/63
	32K4W	45542	183158	505.8	0/80	16309	37098	642.4	0/63	16309	37098	651.8	0/63
Seed-tegra	32K4W	14897	568528	720	0/0	71223	303988	720	0/0	58798	426485	720	0/0
	32K4W	65311	327425	720	0/0	79011	348981	720	0/0	87641	257529	720	0/0
encoder-tom	32K4W	6	18	<0.01	1/0	6	18	<0.01	1/0	6	18	<0.01	1/0
	32K4W	6	18	<0.01	1/0	6	18	<0.01	1/0	6	18	<0.01	1/0

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 bellow the diagonal line indicates a larger -O0 result, and vice versa. For instance, in SP_1 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 SP_3 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 SP_2, we cannot observe a similar distribution in SP_1 which compares the generated traces. Furthermore, in SP_1, the

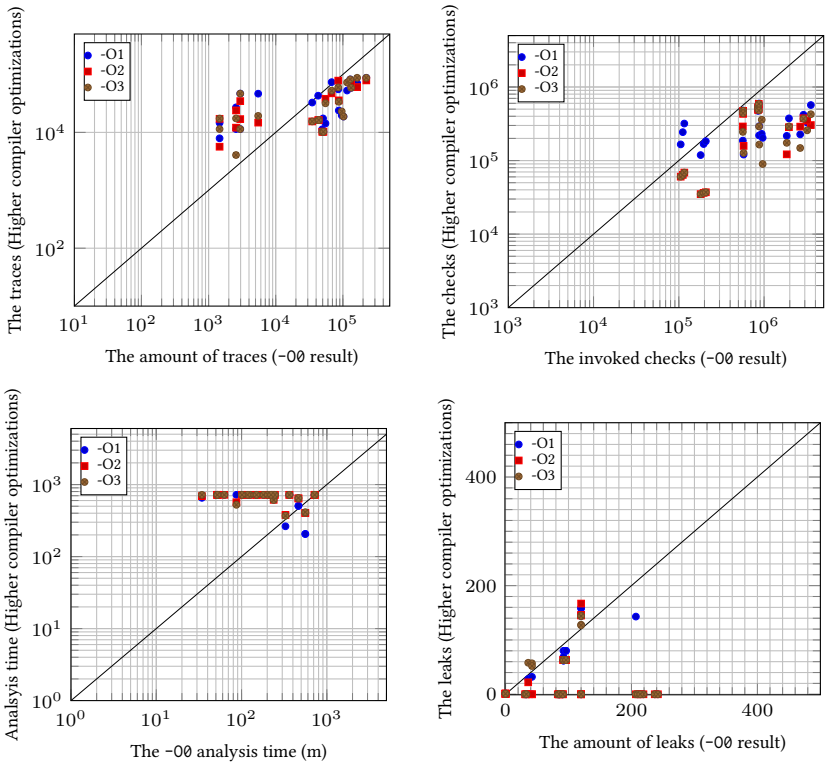


Fig. 8. Higher compiler optimization results versus -O0 result.

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_4 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

Table 7. The differences caused by Speculative Execution

Name	Cache	Time (m)	#.Trace	#.C/D
AES[LibTomCrypt 2019]	32K4W	+90.41	+920700	+3/+25
	32K8W	+64.00	+1222135	0/+36
blowfish[LibTomCrypt 2019]	32K4W	+40.76	+39870	0/+2
	32K8W	+48.66	+70056	0/0
chacha20[LibTomCrypt 2019]	32K4W	+374.50	+91224	+5/0
	32K8W	+377.29	+91543	+61/0
encoder[LibTomCrypt 2019]	32K4W	< +0.01	+6	+1/+5
	32K8W	< +0.01	+6	+1/+5
DES[OpenSSL 2019]	32K4W	+511.92	0	+5/0
	32K8W	+ 522.71	0	+10/0
DES[glibc 2019]	32K4W	+2.29	0	+6/+16
	32K8W	+2.49	0	+1/+26

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, SymO$_3$ even witnesses four *C-type* leaks, which never appear in Table 3.

5.4 The Compound Analysis with Speculation

Recall that SymO$_3$ 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 participant *out-of-order* execution. In this section, we conduct a compound analysis, which augments SymO$_3$ 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 SymO$_3$ 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 SymO$_3$ and evaluated its effect upon the original SymO$_3$.

Table 7 presents the result of the enhanced SymO$_3$ against the original SymO$_3$ framework. Let us name the former as SymO$_3$+ 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 SymO$_3$ and SymO$_3$+ in terms of the amount and locations of detected leaks. Columns 3-5 show the differential results between SymO$_3$ and SymO$_3$+. The result shows that 6 out of 24 programs now have new leaks. Furthermore, we analyze the results in more detail as follows.

Overall, SymO$_3$+ 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 SymO$_3$ detected leaks in five programs shown in Table 3. By comparison, SymO$_3$+ detected new leaks in six programs, including the `AES` [LibTomCrypt 2019], which was also reported by SymO$_3$. The two programs that SymO$_3$+ identified no leak, but SymO$_3$ detected leaks are

```
980  ulong32 Tks0[], Tks1[], Tks2[], Tks3[]; //static array
...
1223 ulong32 temp;
1312 ulong32* rk = skey->rijndael.eK; //skey->rijndael.eK: private data
1313 ulong32* rrk = skey->rijndael.eK + (28 + keylen) - 4;
...
1354 temp = rrk[3];
1355 rk[3] =
1356   Tks0[byte(temp, 3)] ∧
1357   Tks1[byte(temp, 2)] ∧
1358   Tks2[byte(temp, 1)] ∧
1359   Tks3[byte(temp, 0)];     //leaking site
```

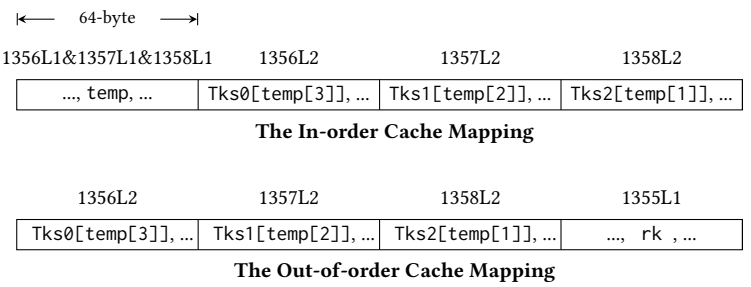


Fig. 9. The code snippet and cache mappings in AES [LibTomCrypt 2019].

DES [LibTomCrypt 2019] and Seed [Tegra 2018]. By contrast, another two DES programs, DES [OpenSSL 2019] and DES [glibc 2019], now have leaks in SymO$_3$+. 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, SymO$_3$+ 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 SymO$_3$ 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**: SymO$_3$ 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 SymO$_3$ results in Table 3.

5.5 The Case Study

SymO$_3$ 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 "1356L1-1357L1-1358L1-1356L2-1357L2-1358L2-1355L1-1359L1", 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 `0x105dc88cfa5f8739d59f3f6b67aaa4a`, 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-deep 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, SymO$_3$ 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; Skakkebæk 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.

Traditional side-channel analysis usually utilizes abstract interpretation based techniques [Doychev et al. 2013; Wang et al. 2019], symbolic execution methods [Bang et al. 2016, 2018; Basu and Chattopadhyay 2017; Chattopadhyay 2017; Chattopadhyay et al. 2017; Guo et al. 2018; Phan et al. 2017] or other formal approaches [Antonopoulos et al. 2017; Chen et al. 2017; Metta et al. 2016].

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 SymO$_3$ 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

A. Abel and J. Reineke. uops.info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, pages 673–686, 2019.

A. C. Aldaya, C. P. García, L. M. A. Tapia, and B. B. Brumley. Cache-timing attacks on RSA key generation. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(4):213–242, 2019. doi: 10.13154/tches.v2019.i4.213-242. URL https://doi.org/10.13154/tches.v2019.i4.213-242.

T. Antonopoulos, P. Gazzillo, M. Hicks, E. Koskinen, T. Terauchi, and S. Wei. Decomposition instead of self-composition for proving the absence of timing channels. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 362–375, 2017.

L. Bang, A. Aydin, Q. Phan, C. S. Pasareanu, and T. Bultan. String analysis for side channels with segmented oracles. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 193–204, 2016.

L. Bang, N. Rosner, and T. Bultan. Online synthesis of adaptive side-channel attacks based on noisy observations. In *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*, pages 307–322, 2018.

G. Barthe, B. Köpf, L. Mauborgne, and M. Ochoa. Leakage resilience against concurrent cache attacks. In *Principles of Security and Trust - Third International Conference, POST 2014*, pages 140–158, 2014.

T. Basu and S. Chattopadhyay. Testing cache side-channel leakage. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2017, Tokyo, Japan, March 13-17, 2017*, pages 51–60, 2017.

T. Basu, K. Aggarwal, C. Wang, and S. Chattopadhyay. An exploration of effective fuzzing for side-channel cache leakage. *Softw. Test., Verif. Reliab.*, 30(1), 2020.

T. Bergan, D. Grossman, and L. Ceze. Symbolic execution of multithreaded programs from arbitrary program contexts. In *OOPSLA*, pages 491–506, 2014.

R. Brotzman, S. Liu, D. Zhang, G. Tan, and M. T. Kandemir. Casym: Cache aware symbolic execution for side channel detection and mitigation. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 505–521. IEEE, 2019.

S. Bucur, V. Ureche, C. Zamfir, and G. Candea. Parallel symbolic execution for automated real-world software testing. In *European Conference on Computer Systems, Proceedings of the Sixth European conference on Computer systems, EuroSys 2011, Salzburg, Austria, April 10-13, 2011*, pages 183–198, 2011.

J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018.*, pages 991–1008, 2018.

D. Burger and T. M. Austin. The simplescalar tool set, version 2.0. *ACM SIGARCH computer architecture news*, 25(3):13–25, 1997.

C. Cadar. Targeted program transformations for symbolic execution. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 906–909, 2015.

C. Cadar, D. Dunbar, and D. R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224, 2008.

S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*, pages 380–394, 2012.

S. Chattopadhyay. Directed automated memory performance testing. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS*, pages 38–55, 2017.

S. Chattopadhyay, M. Beck, A. Rezine, and A. Zeller. Quantifying the information leak in cache attacks via symbolic execution. In *Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE 2017, Vienna, Austria, September 29 - October 02, 2017*, pages 25–35, 2017.

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

J. Chen, W. Hu, L. Zhang, D. Hao, S. Khurshid, and L. Zhang. Learning to accelerate symbolic execution via code transformation. In *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands*, pages 6:1–6:27, 2018.

D. Chu, J. Jaffar, and R. Maghareh. Precise cache timing analysis via symbolic execution. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Vienna, Austria, April 11-14, 2016*, pages 293–304, 2016.

L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, and G. Candea. Cloud9: a software testing service. *Operating Systems Review*, 43(4):5–10, 2009.

L. A. Clarke. A program testing system. In *Proceedings of the 1976 Annual Conference, Houston, Texas, USA, October 20-22, 1976*, pages 488–491, 1976.

A. Colin and I. Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems*, 18 (2-3):249–274, 2000.

B. Coppens, I. Verbauwhede, K. D. Bosschere, and B. D. Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *2009 30th IEEE Symposium on Security and Privacy*, pages 45–60, May 2009. doi: 10.1109/SP.2009.19.

M. Dellinger, P. Garyali, and B. Ravindran. *chronos*. Chronos linux: a besteffort real-time multiprocessor linux kernel, 2011.

J. Dhem, F. Koeune, P. Leroux, P. Mestré, J. Quisquater, and J. Willems. A practical implementation of the timing attack. In *Smart Card Research and Applications, This International Conference, CARDIS '98, Louvain-la-Neuve, Belgium, September 14-16, 1998, Proceedings*, pages 167–182, 1998.

D. Dinu, Y. L. Corre, D. Khovratovich, L. Perrin, J. Großschädl, and A. Biryukov. *triathlon*. Triathlon of lightweight block ciphers for the internet of things., 2015.

C. Disselkoen, D. Kohlbrenner, L. Porter, and D. M. Tullsen. Prime+abort: A timer-free high-precision L3 cache attack using intel TSX. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017.*, pages 51–67, 2017.

S. Dong, O. Olivo, L. Zhang, and S. Khurshid. Studying the influence of standard compiler optimizations on symbolic execution. In *26th IEEE International Symposium on Software Reliability Engineering, ISSRE 2015, Gaithersbury, MD, USA, November 2-5, 2015*, pages 205–215, 2015.

G. Doychev and B. Köpf. Rigorous analysis of software countermeasures against cache attacks. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 406–421, 2017.

G. Doychev, D. Feld, B. Köpf, L. Mauborgne, and J. Reineke. Cacheaudit: A tool for the static analysis of cache side channels. In *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, pages 431–446, 2013.

Y. Etsion. *Computer Architecture: Out-of-order Execution*. https://iis-people.ee.ethz.ch/~gmichi/asocd/addinfo/Out-of-Order_execution.pdf, 2013.

C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In J. Palsberg and M. Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 110–121. ACM, 2005.

FourQLib. *FourQLib*. https://github.com/Microsoft/FourQlib/, 2020.

freeradius. *freeradius*. https://freeradius.org/, 2020.

GDK. *GDK*. GDK Library https://developer.gnome.org/gdk3/3.22/, 2018.

glibc. *glibc-2.29.9000*. https://www.gnu.org/software/libc/, 2019.

B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida. ASLR on the line: Practical cache attacks on the MMU. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017.

D. Gruss, C. Maurice, K. Wagner, and S. Mangard. Flush+flush: A fast and stealthy cache attack. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721*, DIMVA 2016, page 279–299, Berlin, Heidelberg, 2016. Springer-Verlag. ISBN 9783319406664.

M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez. Spectector: Principled detection of speculative information flows. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1–19. IEEE, 2020. doi: 10.1109/SP40000.2020.00011. URL https://doi.org/10.1109/SP40000.2020.00011.

S. Guo, M. Kusano, C. Wang, Z. Yang, and A. Gupta. Assertion guided symbolic execution of multithreaded programs. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 854–865, 2015.

S. Guo, M. Wu, and C. Wang. Adversarial symbolic execution for detecting concurrency-related cache timing leaks. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pages 377–388, 2018.

S. Guo, Y. Chen, P. Li, Y. Cheng, H. Wang, M. Wu, and Z. Zuo. Specusym: Speculative symbolic execution for cache timing leak detection. In *Proceedings of the 42th International Conference on Software Engineering: Companion Proceeedings, ICSE 2020, Seoul, South Korea*, pages 1235–1247, 2020. doi: 10.1145/3377811.3380428. URL https://doi.org/10.1145/3377811.3380428.

J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. *WCET@mdh*. The Mälardalen WCET Benchmarks – Past, Present and Future, 2010.

T. Gysi, T. Grosser, L. Brandner, and T. Hoefler. A fast analytical model of fully associative caches. In K. S. McKinley and K. Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 816–829. ACM, 2019.

S. He, M. Emmi, and G. F. Ciocarlie. ct-fuzz: Fuzzing for timing leaks. *CoRR*, abs/1904.07280, 2019.

Intel. *The SkyLake Microarchitecture.* https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf, 2016.

J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

KLEE-Native. *Binary Symbolic Execution with KLEE-Native.* https://github.com/lifting-bits/klee#klee-native, 2019.

P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

P. C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, pages 104–113, 1996.

M. Kusano and C. Wang. Thread-modular static analysis for relaxed memory models. In E. Bodden, W. Schäfer, A. van Deursen, and A. Zisman, editors, *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 337–348. ACM, 2017.

S. K. Lahiri, S. A. Seshia, and R. E. Bryant. Modeling and verification of out-of-order microprocessors in uclid. In *International Conference on Formal Methods in Computer-Aided Design*, pages 142–159. Springer, 2002.

C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd IEEE/ACM International Symposium on Code Generation and Optimization, 20-24 March 2004, San Jose, CA, USA*, pages 75–88, 2004.

C. Lee, M. Potkonjak, and W. Mangione-Smith. *mediabench.* MediaBench: a tool for evaluating and synthesizing multimedia and communications systems, 1997.

X. Li, A. Roychoudhury, and T. Mitra. Modeling out-of-order processors for software timing analysis. In *Proceedings of the 25th IEEE Real-Time Systems Symposium, 5-8 December 2004, Lisbon, Portugal*, pages 92–103, 2004.

X. Li, A. Roychoudhury, and T. Mitra. Modeling out-of-order processors for WCET analysis. *Real-Time Systems*, 34(3): 195–227, 2006.

libfixedtimefixedpoint. *libfixedtimefixedpoint.* A library for doing constant-time fixed-point numeric operations: https://github.com/kmowery/libfixedtimefixedpoint/, 2017.

Libgcrypt. *Libgcrypt-1.8.4.* https://gnupg.org/software/libgcrypt/index.html, 2018.

LibTomCrypt. *LibTomCrypt.* http://www.libtom.net/LibTomCrypt/, 2019.

M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 973–990, 2018.

mbedTLS. *mbedTLS.* https://tls.mbed.org/code/releases/, 2017.

R. Metta, M. Becker, P. Bokil, S. Chakraborty, and R. Venkatesh. TIC: a scalable model checking based approach to WCET estimation. In T. Kuo and D. B. Whalley, editors, *Proceedings of the 17th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools, and Theory for Embedded Systems, LCTES 2016, Santa Barbara, CA, USA, June 13 - 14, 2016*, pages 72–81. ACM, 2016. doi: 10.1145/2907950.2907961. URL https://doi.org/10.1145/2907950.2907961.

A. Moshovos and G. S. Sohi. Memory dependence speculation tradeoffs in centralized, continuous-window superscalar processors. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture, Toulouse, France, January 8-12, 2000*, pages 301–312, 2000.

S. Nilizadeh, Y. Noller, and C. S. Pasareanu. Diffuzz: differential fuzzing for side-channel analysis. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 176–187, 2019.

O. Oleksenko, B. Trach, M. Silberstein, and C. Fetzer. Specfuzz: Bringing spectre-type vulnerabilities to the surface. In S. Capkun and F. Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 1481–1498. USENIX Association, 2020.

S. Önder and R. Gupta. Dynamic memory disambiguation in the presence of out-of-order store issuing. In *Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 32, Haifa, Israel, November 16-18, 1999*, pages 170–176, 1999.

OpenSSL. *OpenSSL-1.1.1c.* https://mta.openssl.org/pipermail/openssl-announce/2019-May/000153.html, 2019.

Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 1406–1418, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450338325. doi: 10.1145/2810103.2813708. URL https://doi.org/10.1145/2810103.2813708.

D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: The case of AES. In *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2006, Proceedings*, pages 1–20, 2006.

V. S. Pai, P. Ranganathan, and S. V. Adve. Rsim: An execution-driven simulator for ilp-based shared-memory multiprocessors and uniprocessors. *IEEE Technical Committee on Computer Architecture Newsletter*, 1997.

C. S. Pasareanu and N. Rungta. Symbolic pathfinder: symbolic execution of java bytecode. In *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, pages 179–180, 2010.

Q. Phan, L. Bang, C. S. Pasareanu, P. Malacaria, and T. Bultan. Synthesis of adaptive side-channel attacks. *IACR Cryptology ePrint Archive*, 2017:401, 2017.

S. Poeplau and A. Francillon. Systematic comparison of symbolic execution systems: intermediate representation and its generation. In *Proceedings of the 35th Annual Computer Security Applications Conference, ACSAC 2019, San Juan, PR, USA, December 09-13, 2019*, pages 163–176, 2019.

S. Poeplau and A. Francillon. Symbolic execution with symcc: Don't interpret, compile! In S. Capkun and F. Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 181–198. USENIX Association, 2020.

C. Rapier and B. Bennett. High speed bulk data transfer using the SSH protocol. In *Proceedings of the 15th ACM Mardi Gras conference: From lightweight mash-ups to lambda grids: Understanding the spectrum of distributed computing requirements, applications, tools, infrastructures, interoperability, and the incremental adoption of key capabilities, Baton Rouge, Louisiana, USA, January 29 - February 3, 2008*, page 11, 2008.

F. Saudel and J. Salwan. *Triton: A Dynamic Binary Analysis Framework*. https://triton.quarkslab.com/, 2013.

E. Schnarr and J. R. Larus. Fast out-of-order processor simulation using memoization. *ACM SIGPLAN Notices*, 33(11):283–294, 1998.

T. Sha, M. M. K. Martin, and A. Roth. Scalable store-load forwarding via store queue index prediction. In *38th Annual IEEE/ACM International Symposium on Microarchitecture, 12-16 November 2005, Barcelona, Spain*, pages 159–170, 2005.

J. U. Skakkebæk, R. B. Jones, and D. L. Dill. Formal verification of out-of-order execution using incremental flushing. In *International Conference on Computer Aided Verification*, pages 98–109. Springer, 1998.

J. E. Smith and A. R. Pleszkun. Implementation of precise interrupts in pipelined processors. In *Proceedings of the 12th Annual Symposium on Computer Architecture, Boston, MA, USA, June 1985*, pages 36–44, 1985.

N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*, 2016.

Tegra. *Kernel tree for NVIDIA Tegra family SOICs on Android*. https://android.googlesource.com/kernel/tegra/+/android-8.1.0_r0.113/crypto, 2018.

G. Wang, S. Chattopadhyay, A. K. Biswas, T. Mitra, and A. Roychoudhury. Kleespectre: Detecting information leakage through speculative cache attacks via symbolic execution. *ACM Trans. Softw. Eng. Methodol.*, 29(3):14:1–14:31, 2020. doi: 10.1145/3385897. URL https://doi.org/10.1145/3385897.

S. Wang, P. Wang, X. Liu, D. Zhang, and D. Wu. Cached: Identifying cache-based timing channels in production software. In E. Kirda and T. Ristenpart, editors, *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, pages 235–252. USENIX Association, 2017.

S. Wang, Y. Bao, X. Liu, P. Wang, D. Zhang, and D. Wu. Identifying cache-based side channels through secret-augmented abstract interpretation. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 657–674, 2019.

O. Weisse, J. Van Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch, and Y. Yarom. Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution. *Technical report*, 2018.

J. Wichelmann, A. Moghimi, T. Eisenbarth, and B. Sunar. Microwalk: A framework for finding side channels in binaries. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*, pages 161–173, 2018.

R. Wilhelm, S. Altmeyer, C. Burguière, D. Grund, J. Herter, J. Reineke, B. Wachter, and S. Wilhelm. Static timing analysis for hard real-time systems. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 3–22, 2010.

M. Wu and C. Wang. Abstract interpretation under speculative execution. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 57–69, 2019.

Y. Xiao, M. Li, S. Chen, and Y. Zhang. Stacco: Differentially analyzing side-channel traces for detecting ssl/tls vulnerabilities in secure enclaves. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 859–874, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349468. doi: 10.1145/3133956.3134016. URL https://doi.org/10.1145/3133956.3134016.

Y. Yang, X. Chen, G. Gopalakrishnan, and R. M. Kirby. Efficient stateful dynamic partial order reduction. In *Model Checking Software, 15th International SPIN Workshop, Los Angeles, CA, USA, August 10-12, 2008, Proceedings*, pages 288–305, 2008.

Y. Yang, X. Chen, G. Gopalakrishnan, and R. M. Kirby. Distributed dynamic partial order reduction. *STTT*, 12(2):113–122, 2010.

Y. Yarom and K. Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, pages 719–732, 2014.

J. Yu, L. Hsiung, M. E. Hajj, and C. W. Fletcher. Data oblivious ISA extensions for side channel-resistant and high performance computing. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*, 2019.

Y. Zhang, A. Juels, A. Oprea, and M. K. Reiter. Homealone: Co-residency detection in the cloud via side-channel analysis. In *2011 IEEE symposium on security and privacy*, pages 313–328. IEEE, 2011.