Symbolic Predictive Cache Analysis for Out-of-Order Execution

Zunchen Huang (✉) and Chao Wang

University of Southern California, Los Angeles CA 90089, USA
{zunchenh, wang626}@usc.edu

Abstract. We propose a trace-based symbolic method for analyzing cache side channels of a program under a CPU-level optimization called out-of-order execution (OOE). The method is predictive in that it takes the in-order execution trace as input and then analyzes all possible out-of-order executions of the same set of instructions to check if any of them leaks sensitive information of the program. The method has two important properties. The first one is *accurately* analyzing cache behaviors of the program execution under OOE, which is largely overlooked by existing methods for side-channel verification. The second one is *efficiently* analyzing the cache behaviors using an SMT solver based symbolic technique, to avoid explicitly enumerating a large number of out-of-order executions. Our experimental evaluation on C programs that implement cryptographic algorithms shows that the symbolic method is effective in detecting OOE-related leaks and, at the same time, is significantly more scalable than explicit enumeration.

Keywords: program analysis · out-of-order execution · side channel · SMT solver

1 Introduction

There has been growing interest in recent years in detecting side-channel leaks in software using automated program analysis and verification techniques, due to the increased awareness of the threat of real-world side-channel attacks [4,15,18]. These are *side-channel* attacks because they exploit dependencies between sensitive information of the program and non-functional properties of the computing platform, including cache-related timing variations caused by CPU-level optimizations such as pipelining and branch prediction. While there are existing methods for detecting these side channels based on static analysis [6,28,31] and symbolic execution [3, 10–12, 29], they do not accurately model an important CPU-level optimization called out-of-order execution (OOE).

Out-of-order execution is widely adopted by modern CPUs. It is possible for a program to be free of side-channel leaks when instructions are executed in the *program order* but have leaks when they are executed out of order. Here, the program order refers to the order in which instructions appear in the program. However, modeling out-of-order execution during program analysis is a

E. B. Johnsen and M. Wimmer (Eds.): FASE 2022, LNCS 13241, pp. 163–183, 2022.
https://doi.org/10.1007/978-3-030-99429-7_10

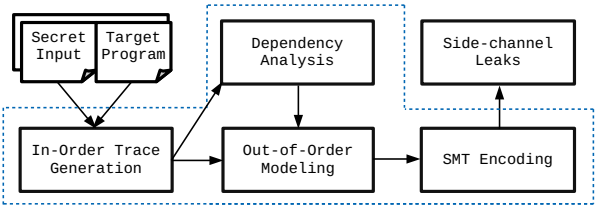

Fig. 1. SPRECA – symbolic predictive analysis for out-of-order execution.

challenging task due to the inherently large number of possible scenarios that must be considered. Generally speaking, instructions within a fixed window (an imaginary window used to model the effect of hardware features including the reorder buffer, issue queue, and load-store queue) may be executed in any order as long as it respects the semantics of the program. Thus, given N instructions, the number of possible execution orders can be as large as $O(N!)$. Since it is practically intractable to examine these execution orders individually, existing methods had to choose from the following two undesired outcomes: if they over-approximate, they may report bogus leaks since some infeasible execution orders will be included; but if they under-approximate, they may miss real leaks since some feasible execution orders will be excluded.

To solve the aforementioned problem, we propose a *trace-based symbolic predictive analysis* to accurately *and* efficiently analyze the OOE related cache behaviors. Here, *accurately* means that our method does not over- or under-approximate the OOE behaviors but precisely encodes these behaviors as a set of logical constraints; *efficiently* means that our method avoids enumerating the out-of-order executions explicitly to avoid the exponential blowup; instead it leverages an off-the-shelf SMT solver to conduct a symbolic analysis of the logical constraints. Our method is *predictive* in that, given an in-order execution trace of the program, it analyzes the cache behaviors of all out-of-order executions of the instructions that appeared in the in-order execution, instead of executing them.

Fig. 1 shows the overall flow of our method, named SPRECA, which takes an annotated C program as input; the annotation marks program inputs as either public or private (secret). Internally, our method has three steps. In the first step, it utilizes the `LLVM` compiler to parse the C program, compute the program dependencies, and use the information to instrument the LLVM bit-code. The instrumented program, at run time, can generate the in-order execution trace. In the second step, our method encodes the set of all possible OOE related cache behaviors as a set of logical constraints, to be solved by an off-the-shelf SMT solver. In the third step, our method checks if there are *secret-dependent* divergent cache behaviors, e.g., an out-of-order execution causing a cache hit for one value of the secret variable but a cache miss for another value of the secret variable.

The main contribution of our work is symbolically modeling the OOE related cache behaviors accurately and efficiently. We design the SMT encoding (to be presented in Section 5) carefully to make it compact. For example, a straightforward encoding of all possible permutations of N instructions would lead to an SMT formula of size $O(N^3)$, since any instruction may have any other instruction as its predecessor and, as a result, the update function must be encoded for each predecessor's cache state and the current cache state. Our method, in contrast, avoids most of these update functions by leveraging the program dependency relations recorded in the in-order execution trace to prune away the infeasible permutations.

Our method differs significantly from the method of Guo et al. [10,11] based on symbolic execution. While their method also uses symbolic analysis, they only made the program input symbolic, whereas the out-of-order executions are still enumerated explicitly (this is evident based on their use of a technique designed for speeding up explicit enumeration, called *partial order reduction*). In other words, for each out-of-order execution, they had to generate an SMT formula to check if it has divergent cache behaviors; as a result, they did not avoid the exponential blowup. In contrast, our method generates a single SMT formula to encode all possible out-of-order executions associated with the in-order execution. In addition to being more efficient, our single-formula based encoding can be more easily adapted to model other CPU-level optimizations by slightly modifying how dependencies are encoded as logical constraints.

We have implemented our method in a software tool by leveraging the open-source LLVM compiler [17] and the Z3 SMT solver [19]. Specifically, we use LLVM to parse the C program, compute the program dependencies, and instrument the bit-code, to generate the in-order execution trace at run time. We use Z3 to implement symbolic analysis of the out-of-order executions. We evaluated our method on a set of C programs from `OpenSSL` that implement well-known block ciphers and cryptographic hash functions. The experimental results show that our method, by accurately modeling the OOE related cache behaviors, can detect OOE-related side-channel leaks that otherwise would have been missed. The results also show that our SMT solver-based symbolic analysis is significantly more scalable than explicit enumeration.

To summarize, this paper makes the following contributions:

- We propose a trace-based symbolic predictive analysis for detecting OOE related cache side-channel leaks.
- We rely on an off-the-shelf SMT solver to *accurately* and *efficiently* analyze the out-of-order executions associated with an in-order execution trace.
- We demonstrate the effectiveness of our method on C programs from an open-source library that implements well-known cryptographic algorithms.

The remainder of this paper is organized as follows. First, we motivate our work using examples in Section 2. Then, we provide the technical background in Section 3. Next, we present our method in Sections 4 and 5, followed by the experimental results in Section 6. We review the related work in Section 7. Finally, we give our conclusions in Section 8.

2 Motivation

In this section, we use examples to illustrate the cache behaviors of the in-order execution and an out-of-order execution. We also explain the high-level idea of our trace-based symbolic analysis.

2.1 The Example Program

Fig. 2 shows the code snippet which, for ease of presentation, is written in a mixture of C and simplified assembly language. Here, assume $i \in \{0, 1, 2\}$ is a secret variable and each array element $A[i]$ occupies 4 bytes in memory. Furthermore, while our method handles realistic cache size and configurations, in this motivating example, we assume the cache has only one set, consisting of 3 cache lines, with each cache line holding only 4 bytes. We assume the cache is fully associative, and uses the LRU (least recently used) replacement policy. Under these assumptions, each array element $A[i]$ occupies an entire cache line.

```
1     load  A[0];
2     load  A[1];
3     load  A[2];
4     store A[i]; /* Can the secret value i affect the cache behavior? */
5     load  B;
```

Fig. 2. An example program where the value of i is a secret.

2.2 The Execution Order

The order in which instructions are written in a program is called the *program order*. During the in-order execution, instructions are executed according to their program order. Without loss of generality, we assume that there are two types of instructions: memory-related instructions such as Load and Store, and non-memory-related instructions, such as ALU and branch instructions. As far as this work is concerned, our focus is on memory-related instructions because non-memory instructions do not affect cache behavior [1].

Fig. 3 compares the in-order execution on the left with a possible out-of-order execution on the right. The out-of-order execution is a permutation of instructions of the in-order execution that, at the same time, must respect the semantics of the original program. In both of these two execution traces, each row represents an instruction and its associated memory address. Note that while a program may have if-else statements and thus multiple paths, an execution trace corresponds to only one program path.

[1] Non-memory instructions may impose ordering constraints over memory-related instructions. These constraints are computed by our method, and used to constrain the analysis of out-of-order executions; details are in Section 4.

```
1   In-order                                          Out-of-order
2   I1: load  0x77ef5bd0     /*A[0]*/      ||   I1: load  0x77ef5bd0     /*A[0]*/
3   I2: load  0x77ef5bd4     /*A[1]*/      ||   I2: load  0x77ef5bd4     /*A[1]*/
4   I3: load  0x77ef5bd8     /*A[2]*/      ||   I3: load  0x77ef5bd8     /*A[2]*/
5   I4: store 0x77ef5bd0,... /*A[i]*/      ||   I5: load  0x77ef5bdc     /*B   */
6   I5: load  0x77ef5bdc     /*B   */      ||   I4: store 0x77ef5bd0,... /*A[i]*/
```

Fig. 3. Two execution orders of the example program in Fig. 2.

2.3 The Cache State

Given a program execution, regardless of whether it is the in-order execution or one of the out-of-order executions, it is straightforward to compute changes of the cache state at each step. The cache state of our running example can be defined as a tuple $S = \langle Age(A[0]),\ Age(A[1]),\ Age(A[2]),\ Age(B)\rangle$, consisting of the ages of cache lines associated with the four program variables. Since we assume that the cache holds at most 3 variables (lines) at any moment if a variable is inside the cache, its age must be 0, 1, or 2; and if it is evicted from the cache, its age must be 3. Initially, the cache state is $S_0 = \langle -1, -1, -1, -1\rangle$, where -1 is a special symbol meaning it is not loaded into cache yet.

In-Order Cache Behavior As shown in Fig. 4 for the in-order execution, executing the first instruction `load` $A[0]$ changes the cache state to $S_{I_1} = \langle 0, -1, -1, -1\rangle$ from S_0, where S_{I_1} is the cache state after executing I_1. That is, variable $A[0]$ now occupies the youngest cache line. Similarly, after executing the first three instructions, the cache state becomes $S_{I_3} = \langle 2, 1, 0, -1\rangle$, meaning that $A[2]$ occupies the youngest cache line and $A[0]$ occupies the oldest cache line. Thus, executing the instruction `store` $A[i]$ results in a *cache hit* regardless of whether $i = 0, 1$, or 2. At this moment, the age of variable B remains -1 since it has not yet been loaded to the cache.

```
1   In-order (for i==1)                                  In-order (for i==0)
2   S_I1 = ⟨ 0,-1,-1,-1⟩ /*A[0] ColdMiss*/      ||   S_I1 = ⟨ 0,-1,-1,-1⟩ /*A[0] ColdMiss*/
3   S_I2 = ⟨ 1, 0,-1,-1⟩ /*A[1] ColdMiss*/      ||   S_I2 = ⟨ 1, 0,-1,-1⟩ /*A[1] ColdMiss*/
4   S_I3 = ⟨ 2, 1, 0,-1⟩ /*A[2] ColdMiss*/      ||   S_I3 = ⟨ 2, 1, 0,-1⟩ /*A[2] ColdMiss*/
5   S_I4 = ⟨ 2, 0, 1,-1⟩ /*A[i] Hit     */      ||   S_I4 = ⟨ 0, 2, 1,-1⟩ /*A[i] Hit     */
6   S_I5 = ⟨ 3, 1, 2, 0⟩ /*B    ColdMiss*/      ||   S_I5 = ⟨ 1, 3, 2, 0⟩ /*B    ColdMiss*/
```

Fig. 4. Cache behavior of the *in-order* execution does not depend on the secret value i; that is, for all $i = 0, 1, 2$, accessing $A[i]$ results in a cache hit.

Out-of-Order Cache Behavior There can be many out-of-order executions, or permutations of instructions, corresponding to an in-order execution. While they must preserve the semantics of the in-order execution, they do not have to preserve its cache behavior. Thus, even if the in-order execution does not have

divergent cache behaviors (with respect to a secret variable), one of the out-of-order executions may have divergent cache behaviors. As shown in Fig. 5 for this particular out-of-order execution that reorders `store A[i]` and `load B`, when $i \neq 0$, accessing $A[i]$ results in a cache hit, but when $i = 0$, it results in a cache miss.

```
1   Out-of-order (for i==1)                          Out-of-order (for i==0)
2   S_I1'= ⟨ 0,-1,-1,-1⟩ /*A[0] ColdMiss*/     ||   S_I1'= ⟨ 0,-1,-1,-1⟩ /*A[0] ColdMiss*/
3   S_I2'= ⟨ 1, 0,-1,-1⟩ /*A[1] ColdMiss*/     ||   S_I2'= ⟨ 1, 0,-1,-1⟩ /*A[1] ColdMiss*/
4   S_I3'= ⟨ 2, 1, 0,-1⟩ /*A[2] ColdMiss*/     ||   S_I3'= ⟨ 2, 1, 0,-1⟩ /*A[2] ColdMiss*/
5   S_I5'= ⟨ 3, 2, 1, 0⟩ /*B    ColdMiss*/     ||   S_I5'= ⟨ 3, 2, 1, 0⟩ /*B    ColdMiss*/
6   S_I4'= ⟨ 3, 0, 2, 1⟩ /*A[i] Hit     */     ||   S_I4'= ⟨ 0, 3, 2, 1⟩ /*A[i] Miss    */
```

Fig. 5. Cache behavior of the *out-of-order* execution depends on the secret value i; that is, accessing $A[i]$ results in a cache hit when $i \neq 0$ but a cache miss when $i = 0$.

2.4 The Side-channel Leak

Whenever the cache behavior of an execution (regardless of whether it is the in-order execution or an out-of-order execution) depends on the value of a secret variable, it is called a side-channel leak. This is a security risk because, in modern CPUs, a cache hit only takes 1-3 CPU cycles whereas a cache miss may take up to a hundred CPU cycles. By observing the difference in the execution time of a victim program, the attacker may be able to deduce a certain amount of information about the secret.

In our running example, since `store A[i]` is dependent on the value of the secret variable i, we need to check if executing `store A[i]` leads to divergent cache behaviors. During the in-order execution, the answer is no, since it results in a cache hit for all $i = 0, 1$, and 2. Thus, the in-order execution has no side-channel leak. During one of the out-of-order executions, however, the answer is yes, since it results in a cache hit for some value of i but a cache miss for some other value of i. Thus, the out-of-order execution has a leak.

Generally speaking, there are two types of side-channel analysis techniques: approximate and accurate. While over- or under-approximation may be fast, it leads to poor results, i.e., reporting bogus leaks or missing real leaks. Thus, we are only concerned with accurate analysis techniques. In this context, while it is possible to examine each individual out-of-order execution, it will lead to exponential blowup. Our method, in contrast, encodes the cache behaviors of all out-of-order executions in a single logical formula. The formula is then solved using an efficient, off-the-shelf SMT solver to avoid an exponential blowup.

3 Preliminaries

In this section, we present the technical background related to our analysis of the out-of-order executions and divergent cache behaviors.

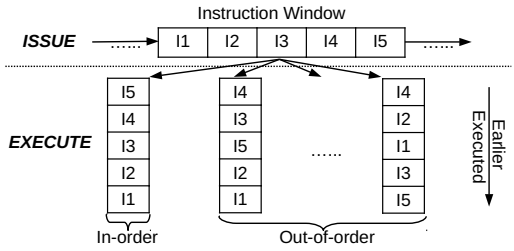

Fig. 6. The instruction window and the different execution orders.

3.1 The Execution Model

Recall that modern CPUs may execute instructions of a program in any order as long as the end result remains the same. The default order is the *program order*, i.e., the order in which instructions appear in the program. For performance reasons, however, the CPU does not always follow the program order, because some instructions may be significantly slower than others and, instead of waiting for the slower instructions to complete, the CPU may choose to execute some subsequent instructions as long as the program semantics is preserved.

Instruction Window As shown in Fig. 6, we use an imaginary *instruction window* to abstract the behavior of various hardware components inside the CPU for supporting out-of-order execution. The size of this instruction window depends on the CPU, including but not limited to the sizes of its reorder buffer, issue queue, and load-store queue. For this work, however, there is no need to delve into the hardware details. Instead, it suffices to assume that within this imaginary window of N instructions, the CPU may choose any execution order as long as the end result remains the same.

Data Hazards To make sure that the end result remains the same, only the out-of-order executions that respect the data dependencies of the original program are allowed. In the computer architecture literature, violations of such dependencies are called *hazards*. Specifically, there are three types of hazards, named RAW (read after write), WAR (write after read), and WAW (write after read), respectively. It is worth noting that RAR (read after read) is not a hazard.

3.2 The Cache Model

Without loss of generality, we assume the cache has K cache lines in total and each cache line has 64 bytes. The cache lines are further divided into M sets, which means each set has (K/M) cache lines. The memory is also divided into 64-byte blocks, each of which is mapped to a unique set. Within the same set, however, the 64-byte block may occupy any of the cache lines. Thus, within the set, it is called *fully associative*; overall, the entire cache is called *set associative*. In this context, a *fully associative* cache is a special case (K-way set associative), while a *direct mapped* cache is another special case (1-way set associative).

The Cache State The cache state is a tuple $S_I = \langle Age(v_1), \ldots, Age(v_n)\rangle$, where each $v_i \in Vars$ $(1 \leq i \leq n)$ is a variable in the program, and $Age(v_i)$ is the age of the cache line associated with v_i. $Vars$ is the set of all variables. Here, we use the subscript in S_I to indicate that it is the cache state resulting from executing the instruction I. Assume that K is the number of cache lines in a set. The domain of $Age(v_i)$ is $\{0, 1, \ldots, K, -1\}$, where an age from 0 to $K-1$ means the variable is inside the cache, while K means the variable is evicted from cache and -1 means it has never been loaded into cache.

We assume that the cache uses the LRU (least recently used) replacement policy. Given a cache state S_I and an instruction I', the new cache state $S_{I'}$ is computed by the $Update(S_I, I')$ function. Assuming that $v \in Vars$ is the variable used by the instruction I', $u_1 \in Vars$ is another variable whose age was younger than v in S_I, and $u_2 \in Vars$ is yet another variable whose age was older than v in S_I, we compute the new cache state $S_{I'} = \langle Age'(v_1), \ldots, Age'(v_n)\rangle$ as follows:

- $Age'(v) = 0$;
- $Age'(u_1) = Age(u_1) + 1$;
- $Age'(u_2) = Age(u_2)$.

That is, the most recently used variable (v) occupies the youngest cache line, any variable (u_1) whose age was younger than v in S_I increases its age by 1, and any variable (u_2) whose age was older than v in S_I keeps its age unchanged.

3.3 The Side-channel Leak Condition

Whenever there is a dependency between the secret and some divergent cache behaviors of an execution, there is a side-channel leak. Thus, there are two requirements. First, there must be divergent cache behaviors, i.e., memory-related instruction causing a cache miss for some input value but a cache hit for some other input value. Second, the input value causing divergent cache behaviors must be a secret, e.g., a password, security token, or cryptographic key.

Thus, the side-channel leak condition can be defined as follows:

$$\exists\, E, I, v_1, v_2 \;.\; CacheStatus(E, I, v_1) \neq CacheStatus(E, I, v_2)$$

Here, E denotes an execution, and $I \in E$ is an instruction in E; v_1 and v_2 are two values of a secret variable $v_s \in Vars$; and $CacheStatus(E, I, v_s)$ is a function that returns the cache status (hit or miss) when instruction I is executed in E using v_s.

4 Analyzing the In-Order Execution

In this section, we present our method for generating, and then analyzing the in-order execution trace. There are two tasks. The first one is to compute the dependencies of memory-related instructions. The second one is to compute the default cache states. Both the dependencies and the default cache states will be used during our symbolic analysis of the out-of-order executions.

4.1 Computing the Dependencies

There are two types of dependencies associated with the in-order execution of a program: explicit dependencies and implicit dependencies.

Explicit Dependencies Explicit dependencies refer to data conflicts that can be directly observed during the execution, by looking at the actual addresses of memory blocks used by the instructions at run time. Consider the in-order execution example in Fig. 3 (left). Since both instructions I_4 and I_1 access the memory block at the address 0x77ef5bd0, and at least one of them is a `store` operation, these two instructions have an explicit dependency; that is, they cannot be reordered during out-of-order.

```
1       load r1 A[0]   /*LD A[0]*/
2       mul  r1 5      /* */
3       add  r2 r1     /* */
4       mov  r3 r2     /* */
5       ...
6       store A[1] r2  /*ST A[1]*/
```

Fig. 7. Example implicit dependency that cannot be observed in the execution trace.

Implicit Dependencies Implicit dependencies, on the other hand, refer to data conflicts that cannot be directly observed during the in-order execution. Fig. 7 shows an example. The code snippet shows that `store A[1]` is dependent on `load A[0]`, through the def-use chain of (register) variables `r1-r3`. Since non-memory instructions (`mul, add, mov` in this example) do not show up in the logged execution trace, their constraints on the memory instructions would have been lost if we do not compute and record them explicitly into the execution trace.

In our method, we compute the implicit dependencies by statically analyzing the LLVM bit-code of the program before instrumenting the bit-code to add *self-logging* capabilities. Then, we execute the instrumented code to obtain the trace. As a result, the implicit dependencies will be captured in the execution trace as a special relation (DEP_{sta}). Static program analysis has a global view of the program and thus is well suited for computing the implicit dependencies. Inside LLVM, the bit-code is represented in a Single Static Assignment (SSA) format, meaning each variable is defined only once, which makes it possible to efficiently compute the implicit dependencies [20].

In addition to the implicit dependencies (DEP_{sta}) computed by static analysis, we also compute the explicit dependencies (DEP_{dyn}) based on the actual addresses appeared in the execution trace: for each memory address, instructions that use the address are checked to see if they have data hazards (RAW, WAR, or WAW). For instructions that have data hazards, their relative execution order during in-order execution cannot be violated; otherwise, the original program semantics may be changed.

Given both the statically computed DEP_{sta} and the dynamically computed DEP_{dyn}, we compute their transitive closure to obtain $DEP = (DEP_{sta} \cup DEP_{dyn})^*$, which represents the complete set of dependency constraints that must be respected at all time, to ensure that the out-of-order executions examined by our symbolic analysis are feasible.

The fact that static analysis is *conservative* in nature will not affect the correctness of our subsequent symbolic analysis. Since not all memory-addressing instructions can be statically resolved, as shown by the example instruction `store A[i]` in Fig. 2, static analysis may soundly over-approximate the possible dependencies of memory-related instructions. This is not a problem because it guarantees that, as long as two instructions are marked as *independent*, it is always safe to reorder these instructions during out-of-order execution. This is crucial for ensuring that leaks detected by our method are feasible.

4.2 Computing the Default Cache States

Given the in-order execution trace, we perform an in-order simulation to compute the default cache states, which will be used during our symbolic analysis of the out-of-order executions.

We regard the in-order execution trace as a sequence of instructions $\mathcal{T}_{ino} = \{I_1, \ldots, I_n\}$. The type of each instruction may be Load, Store, Symbolic Load, or Symbolic Store. Each Load/Store instruction is associated with an actual memory address. Each Symbolic Load/Store instruction is associated with a range of addresses that it may use.

Starting with an initial cache state S_0, we compute the sequence of cache states $\mathcal{T}_{cache} = \{S_0, S_{I_1} \ldots, S_{I_n}\}$ using the update function defined in Section 3.2. While the update function in Section 3.2 uses the LRU replacement policy, other cache replacement policies can also be implemented easily.

The result of in-order simulation will be given to our symbolic analysis, to examine the set of all possible out-of-order executions. Here, an out-of-order execution, denoted $\mathcal{T}_{ooe} = \{I'_1, ..., I'_n\}$, is a permutation of instructions of the in-order execution. That is, for all $1 \leq i \leq n$ and instruction $I_i \in \mathcal{T}_{ino}$, there exists $1 \leq j \leq n, i \neq j$ such that $I'_j \in \mathcal{T}_{ooe}$ and $I'_j = I_i$, and vice versa.

5 Analyzing the Out-of-Order Executions

In this section, we present our method for symbolically analyzing the out-of-order executions.

5.1 Symbolic Encoding

Our method uses a single logical formula (Φ) to encode the behaviors of all out-of-order executions of instructions within a sliding window of size N, together with the condition under which an out-of-order execution has secret-dependent, divergent cache behaviors. It guarantees that Φ is satisfiable if and only if there

exists such a side-channel leak in the sliding window of size N. Thus, when setting the value of N, there is a trade-off between coverage and scalability.

Before explaining how Φ is constructed from the in-order execution trace, however, we need to define the notations used in the symbolic encoding.

- **Sliding Window:** We focus on a sliding window of N instructions appeared in the in-order execution trace. Within this window, instructions may be executed in any order as long as they respect the DEP relation; outside of this window, instructions are executed in-order.
- **Program Counter:** We use $(N+1)$ variables $PC\_I_0, PC\_I_1, \ldots, PC\_I_N$ to represent the time when we execute the N instructions $I_1, \ldots, I_N$. The special variable PC_I_0 represents the start time, and each PC_I_i (where $1 \leq i \leq N$) represents the time immediately after I_i is executed.
- **Age of Address after Executing an Instruction:** We use $Age\_addr_k\_I_i$ to represent the cache line age of a memory block at $addr_k$ after we execute instruction I_i. Thus, for all memory addresses $addr_1, \ldots, addr_M$, we have integer variables $Age\_addr_1\_I_i, \ldots, Age\_addr_M\_I_i$ for all $0 \leq i \leq N$.

With these notations, we define the formula Φ as a conjunction of the following subformulas:

$$\Phi = \Phi_{pc} \wedge \Phi_{cs} \wedge \Phi_{ics} \wedge \Phi_{rep} \wedge \Phi_{dep} \wedge \Phi_{divc}$$

where Φ_{pc} is the program counter constraint, Φ_{cs} is the cache state constraint, Φ_{ics} is the initial cache state constraint, Φ_{rep} is the cache replacement constraint, Φ_{dep} is the dependency constraint, and Φ_{divc} is the divergence condition constraint.

Program Counter Constraint (Φ_{pc}) To get a total order of the N instructions, we require that, for all $0 \leq i \leq N$, the value of PC_I_i is unique; furthermore, we require $0 \leq PC\_I_i \leq N$. Thus, the constraint is defined as

$$\Phi_{pc} = \bigwedge_{0 \leq i \leq N} (0 \leq PC\_I_i \leq N) \wedge \bigwedge_{0 \leq i,j \leq N \text{ and } i \neq j} (PC\_I_i \neq PC\_I_j)$$

Cache State Constraint (Φ_{cs}) Let MAX be the cache's associativity, or the maximal number of cache lines that can be mapped to a memory address. After executing an instruction I_i, if $0 \leq Age\_addr_k\_I_i < MAX$, it means the memory block at $addr_k$ is inside the cache; but if $Age\_addr_k\_I_i = MAX$, it means the memory block is evicted out of the cache [2]. Thus, the constraint is defined as

$$\Phi_{cs} = \bigwedge_{0 \leq i \leq N \text{ and } 0 \leq k \leq M} (-1 \leq Age\_addr_k\_I_i \leq MAX)$$

[2] $Age\_addr_k\_I_i = -1$ means it has never been loaded to the cache yet.

Initial Cache State Constraint (Φ_{ics}) Before the first instruction is executed, the cache must be set to a proper initial state. In other words, variables $Age\_addr_1\_I_0, \ldots, Age\_addr_M\_I_0$ must be initialized based on the default cache states computed by in-order simulation (Section 4.2). Thus, the constraint is defined as

$$\Phi_{ics} = \bigwedge_{0 \le k \le M} (Age\_addr_k\_I_0 = init\_age\_addr_k)$$

Replacement Constraint (Φrep) Assuming that instruction I_j is immediately before I_i during an out-of-order execution, we define the cache line ages after executing I_i based on their ages after executing the predecessor instruction I_j. Let $addr_k$ be the address used by I_i, $addr_{k1}$ be any address whose age was younger than that of $addr_k$ immediately before executing I_i, and $addr_{k2}$ be any address whose age was older than that of $addr_k$. According to the update function defined in Section 3.2, we set $Age\_addr_k\_I_i$ to 0, set $Age\_addr_{k1}\_I_i$ to $(Age\_addr_{k1}\_I_j + 1)$, and set $Age\_addr_{k2}\_I_i$ to $Age\_addr_{k2}\_I_j$. Let the relation $UpdateRel(I_i, I_j)$ be the conjunction of the constraints defined above.

If a symbolic address (secret-dependent) is used by I_i, we encode it into the update relation as follows: for each concrete address that may be instantiated from the symbolic address, we construct an update relation $UpdateRel()$ under the assumption that it may be the actual address used by I_i.

Overall, the cache replacement constraint is defined as

$$\Phi_{rep} = \bigwedge_{0 \le i,j \le N \text{ and } i \ne j} (PC\_I_i = PC\_I_j + 1) \implies UpdateRel(I_i, I_j)$$

Dependency Constraint (Φ_{dep}) To ensure that out-of-order executions are feasible, we enforce the relative order of any two instructions if they have dependencies according to the DEP relation. Thus, the constraint is defined as

$$\Phi_{dep} = \bigwedge_{0 \le i,j \le N \text{ and } i \ne j \text{ and } DEP(I_i, I_j)} (PC\_I_i < PC\_I_j)$$

That is, if I_j depends on I_i, I_i must be executed before I_j.

Divergent Cache Constraint (Φ_{divc}) Let Var_s be a symbolic (secret) variable whose values include $v_1, v_2, \ldots$ and let I_i be a symbolic instruction whose actual addresses include $addr_{v_1}, addr_{v_2}, \ldots$ Here, the value v_1 corresponds to $addr_{v_1}$ and the value v_2 corresponds to $addr_{v_2}$. If accessing the memory block at $addr_{v_1}$ leads to a cache hit and accessing $addr_{v_2}$ leads to a cache miss (or vice versa), the target instruction I_i has divergent cache behaviors. Thus, the constraint is defined as

$$\Phi_{divc} = \bigvee_{\forall v_1, v_2} (0 \le Age\_addr_{v_1}\_I_i < MAX) \wedge (Age\_addr_{v_2}\_I_i \ge MAX)$$

Conjoining all of the subformulas defined above, we can construct the entire formula Φ which is satisfiable (SAT) if and only if there is a side-channel leak during one of the out-of-order executions.

5.2 The Overall Algorithm

The overall algorithm for predictive cache analysis is shown in Algorithm 1, which takes the in-order execution trace $\mathcal{T}_{ino} = \{I_1, \ldots, I_n\}$, the in-order cache state trace $\mathcal{T}_{cache} = \{S_0, \ldots, S_n\}$, and the sliding window size N as input. Internally, it uses a sliding window of N instructions, $\mathcal{T}_{window}$, to generate the SMT formula Φ. For this window, S_{init} is the initial cache state as computed by in-order simulation, and I_{target} is the target instruction. The formula Φ is satisfiable if and only if an out-of-order execution of the instructions within the window leads to divergent cache behaviors at the instruction I_{target}.

Algorithm 1 SymbolicCheck($\mathcal{T}_{ino}, \mathcal{T}_{cache}, N$) for predictive cache analysis.

1: **for** $pos \leftarrow 1$ to $(n - N)$ **do**
2: $first = (pos - N > 0)\ ?\ (pos - N) : 1$
3: $\mathcal{T}_{window} = \mathcal{T}_{ino}[first, pos]$
4: $I_{target} = \mathcal{T}_{ino}[pos]$
5: $S_{init} = \mathcal{T}_{cache}[first - 1]$
6: $\Phi =$ BuildFormula($\mathcal{T}_{window}, I_{target}, S_{init}$)
7: if ($SAT(\Phi) == true$) print LEAK_FOUND

Running Example We use the example code snippet in Fig. 2 to illustrate the symbolic encoding presented in this section. For this example, the in-order execution trace generated by our method is shown in the top half of Fig. 8. Note that `A` is marked as symbolic since `A[i]` is affected by the unknown variable `i`. The logical constraints are shown in the bottom half. Assume that the target instruction is $\mathtt{I}_4$, meaning that we want to construct a formula Φ to check if $\mathtt{I}_4$ has divergent cache behaviors.

The program counter and cache state constraints are shown in Lines 10-12; recall that each program counter variable must have a unique value. The dependency constraints are shown in Line 13. Then, in Line 14, we show the two symbolic variables used to check divergent cache behaviors; their values are in the range of the symbolic store in Line 5.

The update function for Instruction $\mathtt{I}_4$ starts from Line 15. If $\mathtt{v}_1$`==0x77ef5bd0`, which means `0x77ef5bd0` is used, the age after executing $\mathtt{I}_4$ is set to 0. The dependency relations indicate that $\mathtt{I}_5$ is allowed to execute before $\mathtt{I}_4$. From Line 16 to 18, we show an example update age constraints with program counter constraint and the condition which `Age_0x77ef5bd4_`$\mathtt{I}_4$ would increase by 1 from its predecessor $\mathtt{I}_5$ according to Section 5.1. Similarly, we encode other predecessors of $\mathtt{I}_4$ for the update function in Line 19. Finally, we encode the divergent cache constraint in Line 20.

```
In-Order Execution  Trace                                          Dependency relations:
I1: load  0x77ef5bd0 /*A[0]*/                                             |
I2: load  0x77ef5bd4 /*A[1]*/                                             V
I3: load  0x77ef5bd8 /*A[2]*/
I4: symbolic store 0x77ef5bd0, 0x77ef5bd4, 0x77ef5bd8 /*A[i]*/ <I1,I4> <I2,I4> <I3,I4>
I5: load  0x77ef5bdc /*B   */
Initialize Ages: Age_0x77ef5bd0_init == 0, Age_0x77ef5bd4_init == 0
                 Age_0x77ef5bd8_init == 0, Age_0x77ef5bdc_init == 0
PC Constraints:  1 ≤ PC_I1:5 ≤5, PC_I0 == 0, distinct(PC_Ii)
Age Constraints: -1 ≤ Age_0x77ef5bd0_Ii ≤ 3, -1 ≤ Age_0x77ef5bd4_Ii ≤ 3
                 -1 ≤ Age_0x77ef5bd8_Ii ≤ 3, -1 ≤ Age_0x77ef5bdc_Ii ≤ 3
DEP Constraints:  PC_I1 < PC_I4, PC_I2 < PC_I4, PC_I3 < PC_I4, PC_I0 < PC_I1:5
Symbolic Var:    v1/v2 ∈ {0x77ef5bd0, 0x77ef5bd4, 0x77ef5bd8}, v1 ≠ v2
Update Function: v1 == 0x77ef5bd0  ⟹  Age_0x77ef5bd0_I4 == 0
- I4.Pred is I5: (PC_I5 + 1 == PC_I4 ∧ Age_0x77ef5bd4_I5 > Age_0x77ef5bd0_I5
∧ Age_0x77ef5bd0_I5 ≠ -1 ∧ Age_0x77ef5bd4_I5 ≠ -1)
 ⟹   Age_0x77ef5bd4_I4 = Age_0x77ef5bd4_I5 + 1; ......
- I4.Pred is I1, I2, I3: ......
DivC Constraint:  Age_v1_I4 ≥ 3 ∧ Age_v2_I4 < 3 ∧ Age_v2_I4 ≠ -1
```

Fig. 8. An example encoding where the register variable i holds a secret value .

5.3 Optimizations of the Symbolic Encoding

Without optimization, the size of the formula Φ may be as large as $O(N^2M)$ in the worst case, where N is the number of instructions in the sliding window and M is the number of memory addresses used inside the window. In practice, however, many of the logical constraints can be skipped. Here, we propose two optimization techniques.

Skipping the Infeasible Cache Update Relations While constructing the constraints that update the cache states of the instructions, the default approach is to assume that, for any instruction I_i, any other instruction I_j in the same window may be executed immediately before I_i. This means it must construct N^2 update relations. However, due to the dependencies among instructions captured by the DEP relation, there may be many instruction pairs (I_j, I_i) such that I_j is not allowed to execute before I_i. By leveraging the information, we can skip many of these update relations.

Skipping the Unnecessary Φ_{divc} Constraints In many cases, by checking the initial cache state with respect to the sliding window of N instructions, we may be able to know that divergent cache behaviors are impossible during any of the out-of-order executions. In other words, Φ_{divc} is guaranteed to be unsatisfiable (UNSAT). Thus, we can avoid generating Φ. Toward this end, we check for the following two conditions, each of which is sufficient for Φ_{divc} to be UNSAT:

- *All ages are too young*: Inside the initial cache state (with respect to the window), if all cache line ages are less than $(MAX - M)$, where M is the number of unique addresses used in this window, we skip checking any of the instructions in this window for divergent cache behaviors. This is because the cache is large enough that, regardless of the execution order, none of the cache lines will be evicted.

Table 1. Statistics of the benchmark programs and the execution traces.

Name	Description	SLOC	Logged execution trace				
			Length	# Store	# Load	# Addr	#Cache-line
AES	Advanced Encryption Standard	2,077	32,069	8,753	23,316	3,126	139
DES	Archetypal block cipher	1,090	10,162	3,994	6,168	946	148
SEED	Symmetric key block cipher	720	20,820	6,999	13,821	2,044	130
Camellia	Symmetric key block cipher	555	14,595	5,487	9,108	1,63	130
Chacha20	Pseudorandom function based stream cipher	263	15,739	3,668	12,071	687	134
IDEA	International Data Encryption Algorithm	288	2,920	884	2,036	318	140
ARIA	Symmetric key block cipher	1,265	15,672	5,237	10,435	1,642	128
SM4	Symmetric key block cipher	301	11,362	3,410	7,952	1,412	131
MD5	MD5 message-digest algorithm	312	3,134	878	2,256	361	156
Blake2	Hash based on ChaCha stream cipher	512	4,832	1,363	3,469	309	163
SHA256	Secure Hash Algorithm standard	825	5,900	1,302	4,598	435	164
Whirlpool	Hash designed after Square block cipher	1,100	6,941	1,915	5,026	1,257	172

– *The age of addr accessed by the target instruction is too young*: Inside the initial cache state, if the age of *addr* is less than $(MAX - M)$, we skip checking this particular target instruction for divergent cache behaviors. This is because, regardless of the value of the secret variable, this particular cache line will never be evicted out of the cache.

6 Experiments

We have implemented our method in a tool named Spreca, which builds upon the LLVM compiler [17] and the Z3 SMT solver [19]. Specifically, it uses LLVM to implement the static analysis component, which takes a C program as input and computes the dependencies of memory-related instructions before instrumenting the LLVM bit-code; the instrumented bit-code, after compilation, is used to generate the execution trace at run time. We use Z3 to implement our symbolic analysis component, which takes the logged execution trace as input and generates SMT formulas of the cache states for leakage detection. Overall, our implementation includes 3.6K lines of C++ code inside LLVM for trace generation, SMT encoding and leakage detection, as well as 0.5K lines of Python/Bash script code for processing the trace files and automation. The archive is available at: https://doi.org/10.5281/zenodo.6117196.

6.1 Benchmarks

The benchmarks used to evaluate our tool are a set of C programs from `OpenSSL 1.1.1k` that implement well-known block-ciphers such as AES and DES and cryptographic hashing functions such as SHA256 and Whirlpool. The statistics of these benchmark programs are shown in Table 1, including the name of the program, a short description, the number of lines of C code, and statistics of the logged execution trace, which serves as input of our symbolic analysis method. For each execution trace, we show the trace length, the number of Store (ST) operations, the number of Load (LD) operations, the number of distinct memory locations touched by the execution, and the number of corresponding cache lines.

Our experiments were designed to answer the following questions:

Table 2. Results of our symbolic predictive analysis method for 8K fully associative cache, with LRU replacement policy, and window size set to 10.

Name	Trace length	SMT solver calls made			Leaking sites	Analysis time (s)
		total instances	SAT instances	UNSAT instances		
AES	32,069	0	0	0	0	246.0
DES	10,162	1	0	1	0	620.4
SEED	20,820	593	14	579	5	4,922.1
Camellia	14,595	366	15	351	6	2,475.1
Chacha20	15,739	0	0	0	0	4.9
IDEA	2,920	0	0	0	0	1.0
ARIA	15,672	1,060	0	1,060	0	8,760.2
SM4	11,362	27	0	27	0	788.1
MD5	3,134	0	0	0	0	1.2
Blake2	4,832	0	0	0	0	1.8
SHA256	5,900	0	0	0	0	2.4
Whirlpool	6,941	0	0	0	0	2.8

- Is our method effective in detecting OOE-related cache side-channel leaks?
- Is our method, based on symbolic analysis, more scalable than explicit analysis?

Toward this end, for each benchmark program, we applied our symbolic analysis method to check if it can find OOE-related cache side-channel leaks, i.e., leaks that otherwise would not show up unless out-of-order execution is considered. To evaluate the scalability of our method, we also compared it with a baseline explicit analysis method. Due to space limit, we omit the detailed algorithm of the explicit analysis method, which systematically enumerates the same set of out-of-order executions of instructions considered by our symbolic analysis method. Thus, both our symbolic method and the explicit method examine the same type of secret-dependent divergent cache behaviors, but they differ in efficiency and scalability.

6.2 Leakage Detection Results

Table 2 shows the results of our symbolic analysis method. These results were obtained using the following parameters: the cache has a total of 8K bytes, divided into 128 cache lines, with 64 bytes per cache line. The cache is fully associative, with the LRU replacement policy. The OOE window size is set to 10, meaning the number of Load/Store instructions that will be executed out of order is bounded to 10. Recall that inside the reorder buffer, there can be many non-memory instructions (e.g., arithmetic operations); thus, setting the window size to 10 is a reasonable choice. In this table, Columns 1-2 show the program name and the trace length. Columns 3-5 show the number of SMT solver calls, the number of satisfiable (SAT) instances, and the number of unsatisfiable (UNSAT) instances. Column 6 shows the number of leaking sites detected by our method and Column 7 shows the total analysis time in seconds.

Note that the number of SMT solver calls may be smaller than the number of instructions in the trace and, in many cases, is 0 because of the optimizations implemented during our symbolic encoding: for any instruction, if our simple

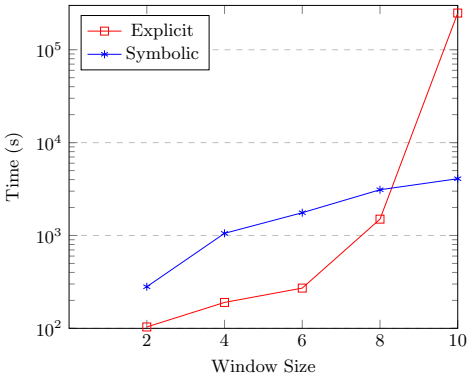

Fig. 9. Comparison of the analysis time: symbolic method versus explicit method.

checks reveal that no OOE-related divergent cache behavior is possible, we skip the more time-consuming SMT solver call. Also note that the number of leaking sites in Column 6, which are locations in the original C program, may be smaller than the number of UNSAT instances in Column 4; this is because multiple UNSAT results may be mapped to the same source code location.

To confirm that the leaking sites reported in Table 2 are indeed feasible (5 for SEED and 6 for Camellia), we manually inspected the source code and the LLVM bit-code of both SEED and Camellia. Our manual inspection shows that the reordered sequences provided by the SMT solver are indeed feasible as we check them against the source code. We also find that the divergent cache behaviors are real in that the two concrete values computed for each symbolic (sensitive) variable can indeed lead to a cache hit in one case but a cache miss in the other case.

6.3 Scalability Results

To evaluate the scalability of our symbolic analysis method, we compared its analysis time to that of the baseline explicit enumeration method. This experiment was conducted on SEED, with the OOE window size set to 2, 4, 6, 8 and 10, respectively. This is because the computational complexity of the problem increases exponentially as the OOE window size increases. The results are shown in Fig. 9, where the x-axis is the OOE window size and the y-axis is the analysis time in seconds. The blue line represents our symbolic method while the red line represents the explicit method.

The results in Fig. 9 show that, while our symbolic method has a higher fixed cost (associated with generating SMT formulas, calling the Z3 solver, and interpreting the results), and thus is slower than the explicit method when the OOE window size is smaller, it becomes significantly more efficient when the window size is larger. The figure also show that, as expected, the explicit method has an exponential blowup – its analysis time is actually worse than exponential

(factorial in the window size) – whereas the scalability of our symbolic method is significantly better.

7 Related Work

As we have mentioned earlier, the most closely related work is that of Guo et al. [10, 11] which relies on KLEE to detect cache side channels. However, their method only treats program input as symbolic, while still explicitly enumerating the out-of-order executions. Unlike their method, we analyze the set of all possible out-of-order executions symbolically by encoding them in a single logical formula to avoid the exponential blowup. In this sense, our method is the only predictive analysis method that can symbolically analyze the cache behaviors of out-of-order executions.

Besides our method and the method of Guo et al. [10, 11], there are many other techniques for analyzing cache side channels. Some of them use symbolic execution as well, e.g., to detect concurrency-related leaks [12] as well as leaks in sequential programs [3, 21, 29, 32]. Others use static analysis techniques including those based on abstract interpretation [6, 28, 30, 31]. In addition to leakage detection, there are techniques for leakage quantification [1, 2, 5, 7, 16] as well. However, none of these prior works considers out-of-order execution.

Beyond side-channel leakage detection and leakage quantification, cache analysis has been used in other applications such as estimating the worst-case execution time (WCET) of real-time software [9, 13, 25]. Beyond cache analysis, the idea of trace-based predictive analysis has been applied to multithreaded programs to detect concurrency bugs [8, 14, 22–24, 26, 27]. However, a crucial difference is that while concurrency bugs are violations of functional properties of a program, our method for side-channel analysis focuses exclusively on non-functional properties.

8 Conclusions

We have presented a symbolic method for analyzing the cache behaviors of out-of-order executions associated with an in-order execution trace. The method uses static analysis to compute dependencies before instrumenting the program to generate the in-order execution trace. Then, it uses an SMT solver based symbolic analysis to analyze the cache behaviors of all out-of-order executions. Our experiments on cryptographic software code show that the symbolic analysis method is effective in detecting OOE-related cache side-channel leaks and is significantly more scalable than explicit analysis. For future work, we plan to extend our method to detect side-channel leaks caused by other CPU-level optimizations.

Acknowledgements This work was partially funded by the U.S. National Science Foundation grants CNS-1722710 and CNS-1702824.

References

1. Backes, M., Köpf, B., Rybalchenko, A.: Automatic discovery and quantification of information leaks. In: 30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA. pp. 141–153 (2009)
2. Bao, Q., Wang, Z., Li, X., Larus, J.R., Wu, D.: Abacus: Precise side-channel analysis. In: 43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021. pp. 797–809 (2021)
3. Brotzman, R., Liu, S., Zhang, D., Tan, G., Kandemir, M.T.: 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. pp. 505–521 (2019)
4. Bulck, J.V., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T.F., Yarom, Y., Strackx, R.: Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In: Enck, W., Felt, A.P. (eds.) 27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018. pp. 991–1008 (2018)
5. Chattopadhyay, S., Beck, M., Rezine, A., Zeller, A.: Quantifying the information leak in cache attacks via symbolic execution. In: Talpin, J., Derler, P., Schneider, K. (eds.) 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. pp. 25–35 (2017)
6. Doychev, G., Feld, D., Köpf, B., Mauborgne, L., Reineke, J.: CacheAudit: A tool for the static analysis of cache side channels. IACR Cryptol. ePrint Arch. **2013**, 253 (2013)
7. Eldib, H., Wang, C., Taha, M.M.I., Schaumont, P.: QMS: evaluating the side-channel resistance of masked software from source code. In: The 51st Annual Design Automation Conference 2014, DAC '14, San Francisco, CA, USA, June 1-5, 2014. pp. 209:1–209:6 (2014)
8. Ganai, M.K., Arora, N., Wang, C., Gupta, A., Balakrishnan, G.: BEST: A symbolic testing tool for predicting multi-threaded program failures. In: Alexander, P., Pasareanu, C.S., Hosking, J.G. (eds.) 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011. pp. 596–599 (2011)
9. Guan, N., Yang, X., Lv, M., Yi, W.: FIFO cache analysis for WCET estimation: a quantitative approach. In: Macii, E. (ed.) Design, Automation and Test in Europe, DATE 13, Grenoble, France, March 18-22, 2013. pp. 296–301 (2013)
10. Guo, S., Chen, Y., Li, P., Cheng, Y., Wang, H., Wu, M., Zuo, Z.: SpecuSym: speculative symbolic execution for cache timing leak detection. In: Rothermel, G., Bae, D. (eds.) ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020. pp. 1235–1247 (2020)
11. Guo, S., Chen, Y., Yu, J., Wu, M., Zuo, Z., Li, P., Cheng, Y., Wang, H.: Exposing cache timing side-channel leaks through out-of-order symbolic execution. Proc. ACM Program. Lang. **4**(OOPSLA), 147:1–147:32 (2020)
12. Guo, S., Wu, M., Wang, C.: Adversarial symbolic execution for detecting concurrency-related cache timing leaks. In: Leavens, G.T., Garcia, A., Pasareanu, C.S. (eds.) 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. pp. 377–388 (2018)

13. Huynh, B.K., Ju, L., Roychoudhury, A.: Scope-aware data cache analysis for WCET estimation. In: 17th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2011, Chicago, Illinois, USA, 11-14 April 2011. pp. 203–212 (2011)
14. Kahlon, V., Wang, C., Gupta, A.: Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In: Bouajjani, A., Maler, O. (eds.) Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5643, pp. 398–413 (2009)
15. Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y.: Spectre attacks: Exploiting speculative execution. In: 2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019. pp. 1–19 (2019)
16. Köpf, B., Mauborgne, L., Ochoa, M.: Automatic quantification of cache side-channels. In: Madhusudan, P., Seshia, S.A. (eds.) Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings. Lecture Notes in Computer Science, vol. 7358, pp. 564–580 (2012)
17. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis and transformation. In: International Symposium on Code Generation and Optimization. pp. 75–88. San Jose, CA, USA (2004)
18. Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., Hamburg, M.: Meltdown: Reading kernel memory from user space. In: Enck, W., Felt, A.P. (eds.) 27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018. pp. 973–990 (2018)
19. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings. Lecture Notes in Computer Science, vol. 4963, pp. 337–340 (2008)
20. Novillo, D.: Memory SSA - a unified approach for sparsely representing memory operations (2007)
21. Pasareanu, C.S., Phan, Q., Malacaria, P.: Multi-run side-channel analysis using symbolic execution and Max-SMT. In: IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016. pp. 387–400 (2016)
22. Roemer, J., Genç, K., Bond, M.D.: Smarttrack: efficient predictive race detection. In: Donaldson, A.F., Torlak, E. (eds.) Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020. pp. 747–762 (2020)
23. Said, M., Wang, C., Yang, Z., Sakallah, K.A.: Generating data race witnesses by an smt-based analysis. In: Bobaru, M.G., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6617, pp. 313–327 (2011)
24. Sinha, A., Malik, S., Wang, C., Gupta, A.: Predicting serializability violations: SMT-based search vs. DPOR-based search. In: Eder, K., Lourenço, J., Shehory, O. (eds.) Hardware and Software: Verification and Testing - 7th International Haifa Verification Conference, HVC 2011, Haifa, Israel, December 6-8, 2011, Revised Selected Papers. Lecture Notes in Computer Science, vol. 7261, pp. 95–114 (2011)

25. Theiling, H., Ferdinand, C., Wilhelm, R.: Fast and precise WCET prediction by separated cache and path analyses. Real Time Syst. **18**(2/3), 157–179 (2000)
26. Wang, C., Kundu, S., Ganai, M.K., Gupta, A.: Symbolic predictive analysis for concurrent programs. In: Cavalcanti, A., Dams, D. (eds.) FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5850, pp. 256–272 (2009)
27. Wang, C., Limaye, R., Ganai, M.K., Gupta, A.: Trace-based symbolic analysis for atomicity violations. In: Esparza, J., Majumdar, R. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6015, pp. 328–342 (2010)
28. Wang, S., Bao, Y., Liu, X., Wang, P., Zhang, D., Wu, D.: Identifying cache-based side channels through secret-augmented abstract interpretation. In: Heninger, N., Traynor, P. (eds.) 28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019. pp. 657–674 (2019)
29. Wang, S., Wang, P., Liu, X., Zhang, D., Wu, D.: Cached: Identifying cache-based timing channels in production software. In: Kirda, E., Ristenpart, T. (eds.) 26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017. pp. 235–252 (2017)
30. Wu, M., Guo, S., Schaumont, P., Wang, C.: Eliminating timing side-channel leaks using program repair. In: Tip, F., Bodden, E. (eds.) Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018. pp. 15–26 (2018)
31. Wu, M., Wang, C.: Abstract interpretation under speculative execution. In: McKinley, K.S., Fisher, K. (eds.) Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019. pp. 802–815 (2019)
32. Yarom, Y., Genkin, D., Heninger, N.: CacheBleed: a timing attack on openssl constant-time RSA. J. Cryptogr. Eng. **7**(2), 99–112 (2017)