Efficient Symbolic Execution of Concurrent Software

Shengjian Guo

Dissertation submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Engineering

Chao Wang, Co-chair

Michael Hsiao, Co-chair

Yaling Yang

Haibo Zeng

Dongyoon Lee

Na Meng

Sept 21, 2018

Blacksburg, Virginia

Keywords: Symbolic execution, Concurrent software, Predicate summary, Incremental execution, Programmable logic controller, Cache timing leak

Efficient Symbolic Execution of Concurrent Software

Shengjian Guo

ABSTRACT

Concurrent software has been widely utilizing in computer systems owing to the highly efficient computation. However, testing and verifying concurrent software remain challenging tasks. This matter is not only because of the non-deterministic thread interferences which are hard to reason about but also because of the large state space due to the simultaneous path and interleaving explosions. That is, the number of program paths in each thread may be exponential in the number of branch conditions, and also, the number of thread interleavings may be exponential in the number of concurrent operations. This dissertation presents a set of new methods, built upon symbolic execution, a program analysis technique that systematically explores program state space, for testing concurrent programs. By modeling both functional and non-functional properties of the programs as assertions, these new methods efficiently analyze the viable behaviors of the given concurrent programs. The first method is *assertion guided symbolic execution*, a state space reduction technique that identifies and eliminates redundant executions w.r.t the explored interleavings. The second method is *incremental symbolic execution*, which generates test inputs only for the influenced program behaviors by the small code changes between two program versions. The third method is SYMPLC, a technique with domain-specific reduction strategies for generating tests for the multitasking *Programmable Logic Controller* (PLC) programs written in languages specified by the IEC 61131-3 standard. The last method is *adversarial symbolic execution*, a technique for detecting concurrency related side-channel information leaks by analyzing the cache timing behaviors of a concurrent program in symbolic execution. This dissertation evaluates the proposed methods on a diverse set of both synthesized programs and real-world applications. The experimental results show that these techniques can significantly outperform state-of-the-art symbolic execution tools for concurrent software.

Efficient Symbolic Execution of Concurrent Software

Shengjian Guo

GENERAL AUDIENCE ABSTRACT

Software testing is a technique that runs software as a black-box on computer hardware multiple times, with different inputs per run, to test if the software behavior conforms to the designed functionality by developers. Nowadays, programmers have been increasingly developing multithreaded and multitasking software, e.g., web browser and web server, to utilize the highly efficient multiprocessor hardware. This approach significantly improves the software performance since a large computing job can now decompose to a set of small jobs which can then distribute to concurrently running threads (tasks). However, testing multithreaded (multitask) software is extremely challenging. The most critical problem is the inherent non-determinism. Typically, executing sequential software with the same input data always results in the same output. However, running a multithreaded (multitask) software multiple times, even under the same input data, may yield different output in each run. The root reason is that concurrent threads (tasks) may interleave their running progress at any time; thus the internal software execution order may be altered unexpectedly, causing runtime errors. Meanwhile, finding such faults is difficult, since the number of all possible interleavings can be exponentially growing in the number of concurrent thread (task) operations. This dissertation proposes four methods to test multithreaded/multitask software efficiently. The first method summarizes the already-tested program behaviors to avoid future testing runs that cannot lead to new faults. The second method only tests program behaviors that are impacted by program changes. The third method tests multitask *Programmable Logic Controller* (PLC) programs by excluding infeasible testing runs w.r.t the PLC semantics. The last method tests non-functional program properties by systematic concurrency analysis. This dissertation evaluates these methods upon a diverse set of benchmarks. The experimental results show that the proposed methods significantly outperform state-of-the-art techniques for concurrent software analysis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Reasoning about the runtime behaviors of concurrent software has always been a challenging task [52, 67, 83, 114]. It is not only because of the large state space within each thread but also because of the nondeterministic interactions among concurrent threads. In software testing, for example, testers have to carefully generate a set of test inputs that can cover as many as possible the paths in each thread and the interleavings of these concurrently running threads [35, 66, 131, 132]. However, direct construction of these data inputs is impractical because the number of the program paths may be exponential in the number of branch conditions. Furthermore, under the same data input, a program may exhibit different behaviors concerning how the concurrent threads interfere with each other. In the worst case, the number of interleavings can also be exponential in the number of concurrent operations. Hence, due to the simultaneous program path explosion and thread interleaving explosion, an exhaustive enumeration of all possible behaviors of concurrent software is always unrealistic.

Symbolic execution is a program analysis and testing method first proposed in the 1970s [39, 74]. It has attracted considerable attentions in recent years due to the rapid progress of modern SAT (Satisfiability) and SMT (Satisfiability Modulo Theory) Solving techniques [41]. Indeed, the past decade has witnessed exciting advances in symbolic execution related techniques for both sequential [22, 24, 58, 59, 98, 110, 111, 112, 118, 120] and concurrent software [14, 38, 48, 72, 104]. The main strengths of these techniques include precise modeling of the program semantics, systematic state space exploration, and automated test input generation. However, existing methods are still limited in their capabilities of mitigating *state space explosion*, especially for concurrent software.

Moreover, software updates from the patches or augment of new features may introduce new bugs to the existing code base. While existing regression testing tools attempt to address this problem, e.g., by leveraging the changes between two program versions to reduce the testing cost of the new version [70, 117, 133], they primarily focused on test selection or prioritization as opposed to generating new test cases. In this context, symbolic execution is a practical solution to the test generation problem. However, although recent works [100, 106, 127] have leveraged symbolic execution in regression testing to reduce the test generation cost for sequential software, they cannot directly apply to concurrent software.

Apart from multithreaded programs, there also exists multi-task concurrent software, e.g., the Programmable Logic Controller (PLC) software [71]. The interleaved execution of PLCs' multiple tasks engenders complex behaviors that are difficult to analyze in existing PLC verification methods [7, 15, 16, 40, 80, 95, 96]. Thus, an automated testing tool based on symbolic execution could greatly benefit PLC software testing. However, the difference between PLCs' priority-based scheduling scheme and the classic thread scheduling scheme pose significant challenges to traditional symbolic execution methods. Furthermore, such symbolic execution tool has to deal with the periodic and non-terminating semantics of each task in a PLC system.

Besides testing program *functional* properties, symbolic execution based methods can also model and analyze the *non-functional* properties, such as the existence of information leakage through side channels [19, 28, 29**?**]. For example, if symbolic execution technique can generate different values of sensitive inputs (e.g., cryptographic key or security token) that cause program execution timing variance, then the program is considered to have cache timing side-channel leaks. Recent works [12, 19, 28, 29, 99, 101, 123**?**] have applied symbolic execution to detecting side-channel leaks, but they primarily focused on self-leaks in sequential programs. However, a program that is leak-free when running alone may still have leaks when interleaved with other threads. And there lacks a symbolic execution method that targets analyzing such scenario.

1.2 Contributions

In this dissertation, I develop a set of new symbolic execution based methods for concurrent software, to address the challenging problems outlined in section 1.1.

First, I develop a new redundancy removal technique, named *Assertion Guided Symbolic Execution*, to reduce the overhead of symbolic execution by identifying the program paths and thread interleavings that have been explored previously and skipping them during the subsequent exploration if necessary.

Second, I develop a summary-based incremental symbolic execution method, named *Conc-iSE*, to leverages code changes between two closely related program versions to prune away redundant paths and interleavings during the execution of the new program version. *Conc-iSE* allows symbolic execution to explore only new program behaviors introduced by code changes.

Third, I develop a symbolic execution based method, named SYMPLC, for automatically testing

PLC software written in languages specified by the IEC 61131-3 standard. I also develop PLC-specific reductions for eliminating redundant interleavings thus effectively reducing the interleaving search space.

Fourth, I develop SYMSC, an *Adversarial Symbolic Execution* method for detecting cache timing leaks, or proving their absences, in concurrent software. SYMSC shows that *timing-leak-freedom* is not a compositional property: a program that is not leaky when running alone may become leaky when interleaved with other threads.

1.3 Organization

In Chapter 2, I establish the notations and provide the technical background.

In Chapter 3, I present *Assertion Guided Symbolic Execution* for multithreaded C programs.

In Chapter 4, I present *Incremental Symbolic Execution* for multithreaded software written in C.

In Chapter 5, I present SYMPLC, the symbolic execution based testing tool for PLC software code, together with three PLC-specific techniques.

In Chapter 6, I present SYMSC, the *Adversarial Symbolic Execution* method which targets detecting the cache timing side-channel leaks that manifest in concurrent program executions.

Finally, I conclude this dissertation in Chapter 7.

Chapter 2

Preliminaries

In this chapter, I establish the formal notations of a concurrent program and the baseline symbolic execution algorithm for it. In addition, I also define a *generalized interleaving graph* to present possible executions of a concurrent program. The concepts and annotations introduced in this chapter will be used throughout the following chapters.

2.1 Concurrent Programs

For ease of presentation, here I consider a simple imperative language with integer variables, assignments, and if-else statements only. This approach elides the details for handling of complex language features such as pointers, recursion, and system calls in symbolic execution since these are orthogonal issues addressed previously by many symbolic execution tools [22, 38]. A multi-threaded program P consists of a set of threads $\{T_1 \dots T_m\}$, where each thread, T_i, can be viewed as a sequential procedure. All threads share the same set of *global* variables $SVar$ while individual thread maintains a set of *thread-local* variables $LVar_i$.

Let st be an instruction in a thread with the thread index tid. Let *event* $e = \langle tid, l, st, l' \rangle$ be an execution instance of st, where l and l' represent the locations before and after executing the instance of st in the thread tid. If the same instruction is executed more than once, e.g., st is inside a recursive function call or in a loop, the execution makes copies of l, st, l' to make unique for each event. Conceptually, this corresponds to unrolling loops and recursive calls. A *global control state (GCS)* of the multithreaded program is a tuple $s = \langle l_1, \ldots, l_m \rangle$, where each l_i is a location in T_i. A global control state represents an *abstract* state implicitly containing all concrete states that have the same thread locations but potentially different values of the local and global variables.

Properties of interest are represented by assertions. Here I assume that every assertion in the form `assert(c)` is transformed to `if(!c)abort`. I also define an event named **abort** to represent the faulty program termination and another event named **halt** to represent the normal program termination. Let v_l denote a local variable, v_g denote a global variable, $cond_l$ denote a local condition, and exp_l denote an local expression. In addition to **abort** and **halt**, each instruction st in an event may correspond to one of the following three types:

- α-operation, which is a local assignment operation $v_l := exp_l$;
- β-operation, which is a local branch `assume(`$cond_l$`)`;
- γ-operation, which is a global operation defined as follows:
  - γ-I is a global write $v_g := exp_l$ or read $v_l := v_g$;
  - γ-II is a thread synchronization operations.

For each `if(c)-else` statement, the execution use `assume(`c`)` to denote the then-branch, and `assume(`$\neg c$`)` to denote the else-branch. Without loss of generality, the execution assumes that all if-else conditions use only local variables or local copies of global variables [55]. Hence global operations (γ) directly affect the thread interleaving order, while β-operations directly affect the path taken by each thread. α-operations, on the other hand, do not directly affect the selection of any program

path or thread interleaving.

For thread synchronizations, I focus on mutex locks and condition variables since they are frequently used in mainstream multithreaded programming environments such as C, C++, and Java. Specifically, I consider the following types of γ-II operations: thread creation, thread join, lock, unlock, signal, and wait. If other thread synchronizations or blocking operations are used they can be modeled similarly as γ-II events.

During the execution of the program, γ-operations are the thread interleaving points whereas β-operations are thread-local branching points. Both contribute to the path and interleaving explosion. In contrast, α-operations are local and thus invisible to other threads; they do not contribute directly to the path and interleaving explosion.

A *concrete* execution of the multithreaded program is characterized by $\pi = (in, sch)$, where in is the data input and sch is the thread schedule corresponding to the total order of events $e_1 \ldots e_n$. Its homologous *symbolic* execution can be denoted by $(*, sch)$, where the symbol $*$ indicates the data input is kept symbolic and thus may take any value. Each execution of the program P can be represented by a finite word $\{\alpha, \beta, \gamma\}^*\{\textbf{halt}, \textbf{abort}\}$. If the execution ends with **halt** it is a *normal* execution. If the execution ends with **abort** it is a *faulty* execution.

2.2 Generalized Interleaving Graph (GIG)

The set of all possible executions of a multithreaded program can be captured by a *generalized interleaving graph (GIG)*, in which the nodes denote global control states and the edges denote events. The root node corresponds to the initial symbolic state. The leaf nodes correspond to normal or faulty ends of the execution. Each internal node may have:

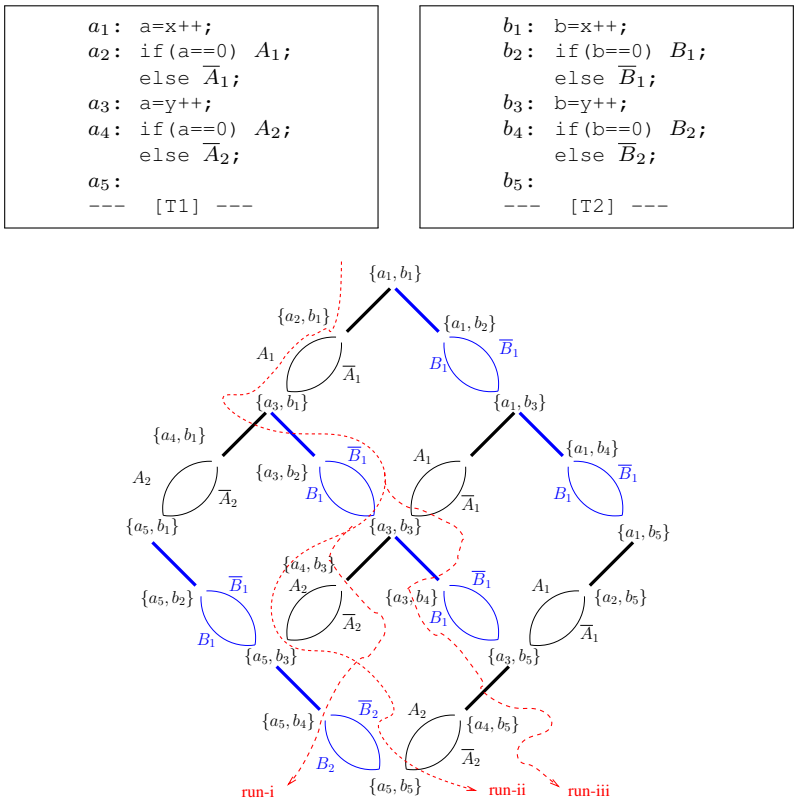


Figure 2.1: A two-threaded program and its generalized interleaving graph (GIG).

- one outgoing edge corresponding to an α-operation;
- two outgoing edges corresponding to a β-operation; or
- k outgoing edges where $k \geq 2$ is the number of enabled γ-operations from different threads.

A node with two or more outgoing edges is a *pivot point*.

- If the pivot point corresponds to β-operations then it is a *branching pivot point* (***b-PP***).
- If the pivot point corresponds to γ-operations then it is a thread *interleaving pivot point* (***i-PP***).

Figure 2.1 shows an example program and its GIG where black edges represent events from thread T_1 and blue edges represent events from thread T_2. For simplicity, here I assume that `a=x++` is atomic on the execution platform. The root node (a_1, b_1) corresponds to the starting points of the

two threads. The terminal node (a_5, b_5) corresponds to the end of the two threads. Nodes such as (a_1, b_1) are i-PP nodes, where the program execution can run either thread 1 which leads to (a_2, b_1), or thread 2 which leads to (a_1, b_2). In contrast, nodes such as (a_2, b_1) are b-PP nodes, where the execution can take either the `assume(a = 0)` branch, leading to the code segment A_1, or the `assume(a ≠ 0)` branch, leading to the code segment $\overline{A}_1$.

Note that the GIG does not have loop-back edges since the GIG paths represent unrolled executions. Furthermore, pointers, aliasing, and function calls have been resolved as well during execution. However, a GIG may have branches, which makes it significantly different from the typical thread interleaving graph used in the partial order reduction literature.

As is typical in symbolic execution algorithms, here I focus on only a finite set of executions and assume that each execution has a finite length. Typically, the user of a symbolic execution tool needs to construct a proper testing environment that satisfies the above assumption. In KLEE [22] and *Cloud9* [38], for example, the user may achieve this by bounding the size of the symbolic input thereby restricting the execution to a fixed number of paths of finite lengths.

2.3 Baseline Symbolic Execution

Following the majority of prior works on symbolic execution, I assume that the given program is terminating and each program execution has a finite length. I also assume the program is deterministic, i.e., the sequence of the instructions will be completely determined by (in, sch), where in is the data input and sch is the thread schedule. Therefore, (in, sch) represents a concrete execution. In contrast, $\pi = (*, sch)$ represents a symbolic execution where $*$ is the symbolic data input and $sch = e_1 \dots e_n$ is an order of the executed events. Algorithm 1 presents the baseline symbolic execution procedure for multithreaded programs following the prio works such as [38, 107, 108].

The recursive procedure $Explore$ is invoked with the symbolic initial state s_0. Inside the procedure, the algorithm differentiates among three scenarios based on whether s, the current state, is an *i-PP* node, a *b-PP* node, or a internal computation node.

If s is an *i-PP* node where multiple γ-operations are enabled, the algorithm recursively explores the next γ event from each thread. If s is a *b-PP* node where multiple sequential branches are feasible, the algorithm recursively explores each branch. If s is a non-branching node, the algorithm explores the next event. The current execution ends if s is a leaf node (normal_end_state, faulty_end_state) or an infeasible_state, at which point the algorithm returns from $Explore(s)$ by popping the state s from the stack S.

Each state $s \in S$ is a tuple $\langle pcon, \mathcal{M}, enabled, branch, done \rangle$, where $pcon$ is the path condition for the execution to reach s from s_0, $\mathcal{M}$ is the symbolic memory map, $s.enabled$ is the set of γ-events when s is an i-PP node, $s.branch$ is the set of β-events when s is a b-PP node, and $s.done$ is the set of α or β events already explored from s by the recursive procedure. Initially, s_0 is set to $\langle true, \mathcal{M}_{init} \rangle$, where $true$ means the state is always reachable and $\mathcal{M}_{init}$ represents the initial content of the memory. The execution of each instruction t is performed by the procedure $NextSymbolicState$ as follows:

- If t is **halt**, the execution ends normally.
- If t is **abort**, and $s.pcon$ is satisfiable under the current memory map $s.\mathcal{M}$, the execution has found an error.
- If t is **v:=exp**, the execution changes the current symbolic memory $\mathcal{M}$ by updating v's content to exp.
- If t is **assume(c)**, the execution updates the path condition to $(pcon \wedge c)$.

At each pivot point (i-PP or b-PP), the baseline algorithm tres to flip a decision made previously to compute a new execution. Let (in, sch) denote the current execution. By flipping the decision

Algorithm 1: Baseline Symbolic Execution Procedure.

```
1  Initially: State stack S = ∅; Start Explore(s_0) with the symbolic state s_0.
2  Explore(State s)
3  begin
4      S.push(s);
5      if s is a b-PP node then
6          while ∃t ∈ s.branch\s.done do
7              Explore(NextSymbolicState(s, t));   // β event
8              s.done ← s.done ∪ {t};
9          end
10     else if s is an i-PP node then
11         while ∃t ∈ s.enabled\s.done do
12             Explore(NextSymbolicState(s, t));   // γ event (enhanced)
13             s.done ← s.done ∪ {t};
14         end
15     else if s is other sequential computation node then
16         Explore(NextSymbolicState(s, s.crt));        // α event
17     else
18         // end of execution
19     end
20     S.pop();
21 end
22 NextSymbolicState(State s, Event t)
23 begin
24     let s = ⟨pcon, M, enabled, branch, done⟩;
25     if t is halt then
26         s' ← normal_end_state;
27     else if t is abort then
28         s' ← faulty_end_state;
29     else if t is assignment v := exp then
30         s' ← ⟨pcon, M[v ↦ exp]⟩;
31     else if t is assume(c) and M[pcon ∧ c] is satisfiable then
32         s' ← ⟨pcon ∧ c, M⟩;
33     else
34         s' ← infeasible_state;
35     end
36     return s';
37 end
```

made previously at an i-PP node, the algorithm computes a new execution (in, sch'), where sch' is a permutation of the original thread schedule. In contrast, by flipping the decision made previously at a b-PP node, the algorithm computes a new execution (in', sch), where in' is a new data input. Note that in both cases, the newly computed execution will be the same as the original execution up to the flipped pivot point. After the flipping, the rest of the execution will be a free run.

As an example, consider the GIG in Figure 2.1, where the current execution is represented by the

dotted line *run-i*. Flipping at the b-PP node (a_4, b_3) would lead to the new execution labeled *run-ii*, whereas flipping at the i-PP node (a_3, b_3) would lead to the new execution *run-iii*.

Chapter 3

Assertion Guided Symbolic Execution

New developments on symbolic execution [39, 74] have applied to both sequential [22, 24, 56, 58, 59, 98, 109, 118] and concurrent programs [14, 48, 104, 107, 108] in existing works. However, these methods are still limited on the problem of *state space explosion*. That is, the number of execution paths within each thread may be exponential in the number of branch conditions, and the number of thread interleavings may be exponential in the number of concurrent operationsi in different threads. Many techniques have been proposed to address this problem, including the use of function summaries [57], interpolation [69, 89, 130], static analysis [17], heuristic exploration [85], and coverage metrics [21, 48].

Assertions can be leveraged to model various program properties, ranging from logic and numerical errors to memory safety and concurrency errors, and has been the focus of many software verification projects. When semantic errors of the program are modeled as code reachability, i.e., the reachability of a bad state guarded by the assertion condition, we can concentrate on exploring potentially failure-inducing executions as opposed to all feasible executions of the program. This approach is particularly attractive in the presence of *concurrency*, since it becomes possible to uni-

formly handle the exploration of both intra-thread execution paths and inter-thread interleavings leading to a simple but more powerful analysis algorithm.

In this chapter, I develop a new and complementary method, named *AGSE* (***A**ssertion **G**uided **S**ymbolic **E**xecution*), which is specifically for pruning redundant executions in multithreaded programs where the properties under verification are expressed as assertions. This method focuses on identifying and eliminating executions that are guaranteed to be redundant for checking assertions.

3.1 Introduction

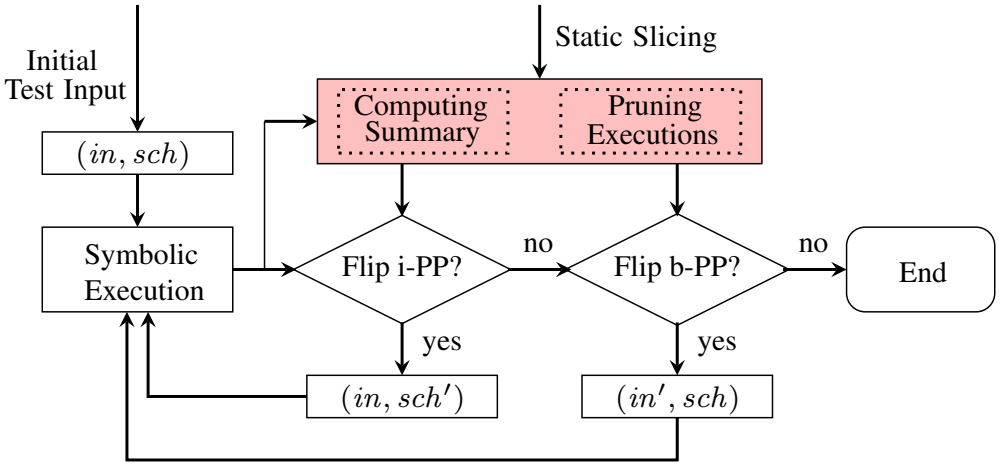


Figure 3.1: The *AGSE* (***A**ssertion **G**uided **S**ymbolic **E**xecution*) method.

Figure 3.1 illustrates the overall flow of *AGSE*. The shaded block represents the new addition, and the remainder illustrates the baseline symbolic execution procedure for multithreaded programs [107, 108]. Specifically, given a program P and some symbolic input variables, the procedure explores feasible executions of the program systematically, e.g., in depth-first search order.

Starting with an initial test (in, sch) consisting of program inputs and thread schedule, the baseline method first produces a concrete execution followed by a symbolic trace. Then, it tries to generate

a new test by flipping a prior decision at either a thread interleaving pivot point (i-PP) or a local branch pivot point (b-PP). The new test is denoted by either (in, sch') or (in', sch), depending on whether changes are made to the thread schedule (sch') or data input (in'), respectively. The iterative procedure terminates when no new test can be generated. State explosion occurs because it has to explore the combined space of data inputs and thread schedules where each execution may be unique, i.e., it leads to a different program state.

AGSE extends the baseline symbolic execution procedure by adding a new constraint-based pruning block shown in Figure 3.1. This new approach centers around the idea of summarizing the reasons why the bad state is unreachable via previously explored executions, and leveraging such information to avoid similarly futile executions. Specifically, at each global control location n, *AGSE* uses a predicate summary (PS) constraint to capture the *weakest preconditions* [44] of the assertion condition along all explored executions starting from n. Therefore, $\mathsf{PS}[n]$ captures the reason why prior executions are not able to violate the assertion. Whenever symbolic state n is reached again through another execution path, *AGSE* checks if the new *path condition* is subsumed by $\mathsf{PS}[n]$. If so, *AGSE* can safely backtrack from n since extending the execution beyond n would never lead to a bad state.

In both cases, the primary technical challenge is to ensure the overall algorithm remains sound in the presence of such optimizations. I have implemented *AGSE* in *Cloud9* [38], a state-of-the-art symbolic execution tool built upon LLVM and KLEE [22], to handle multithreaded C programs. I have also implemented heuristic based minimizations of predicate summary constraints during symbolic execution to reduce the computational overhead. Further, I have evaluated *AGSE* on a number of standard multithreaded C applications. The results show that *AGSE* can reduce the number of testing executions as well as the overall run time significantly.

AGSE's pruning of redundant executions can be viewed as a way of systematically exploring an

abstract search space defined by a set of predicates [11] which, in this case, are extracted from the assertion. Although the concrete search space may be arbitrarily large, the abstract search space can be significantly smaller. In this sense, our method is similar to *predicate abstraction* [60] in model checking except that the latter requires constructing *a priori* a finite-state model from the actual software code whereas our method directly works on the software code while leveraging the predicates to eliminate redundant executions.

AGSE is complementary to standard partial order reduction (POR) techniques in that it relies on property-specific information to reduce the state space. However, POR techniques typically do not target particular states. Experiments show that *AGSE* can indeed eliminate a different class of redundant executions from those eliminated by state-of-the-art dynamic partial order reduction (DPOR) [50] method. Toward this end, since DPOR is an elegant but delicate algorithm that can easily be made unsound without taking great care in the implementation [129], a main technical challenge in *AGSE* is to make sure the new pruning technique does not interfere with DPOR or make it less effective.

AGSE differs from the prior works by Wachter et al. [122], and Chu and Jaffar [36], which extended the well-known framework of lazy abstraction with interpolants by McMillan [89] to multithreaded programs. One main difference is that the computation of predicate summaries is significantly more general than existing methods, especially at the thread interleaving pivot points, where *AGSE* merge summaries from multiple execution paths to form a combined summary. Another main difference is in the integration of property specific pruning with partial order reduction. Kahlon et al. [73] implemented a variant of the symbolic partial order reduction algorithm whereas *AGSE* integrates the predicate summary-based pruning method with the more scalable DPOR algorithm.

To sum up, I make the following contributions:

- I propose an *Assertion Guided Symbolic Execution* method to identify and eliminate redun-

dant executions in multithreaded programs to reduce the overall computational cost.

- I implement the method in a state-of-the-art symbolic execution tool while ensuring it does not interfere with the popular DPOR algorithm or make it less effective.
- I demonstrate through experiments that the new method can indeed achieve a significant performance improvement on public benchmarks.

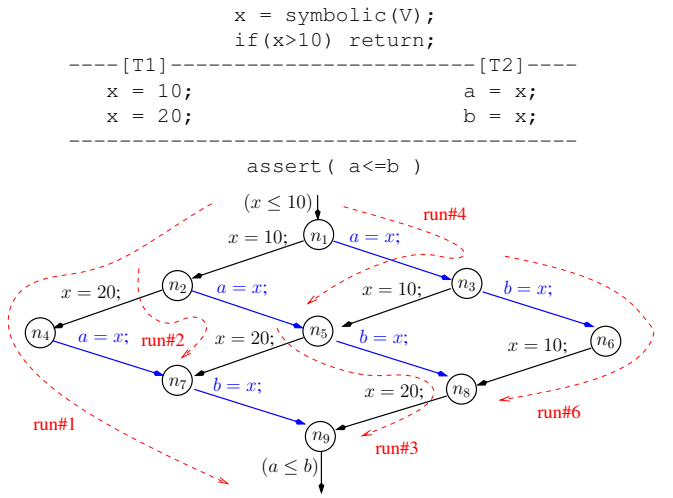
3.2 Motivation

In this section, I illustrate the high-level ideas in *AGSE* with examples.

Consider the example in Figure 3.2, which has two threads T_1 and T_2, two local variables a and b, and a global variable x. The initial value of x is a symbolic input which can be any integer value. The purpose is to check if the assertion fails and, if so, compute a failure-inducing test input.

The program has six distinct executions, each leading to a different final state defined by the values of a and b. According to the theory of partial order reduction [50], they belong to six different equivalence classes [88], as each has a different final state. However, exploring all six executions is not necessary for the purpose of checking the assertion, since some of these executions share the same reason why they cannot reach the bad state. *AGSE* can reduce the exploration from six executions to one full execution together with four partial executions, as illustrated by the red dotted lines in Figure 3.2.

AGSE first extracts a set of predicates by computing the weakest preconditions of the assertion condition along the explored executions. These predicates are then combined at the merge points (in the graph) to form a succinct summary that captures the reason why the bad state has not been reached via executions starting from these merge points. During subsequent symbolic execution it-



Run 1: `if(x<=10)x=10;x=20;a=x;b=x;` leads to `(a=20,b=20)`.
Run 2: `if(x<=10)x=10;a=x;x=20;b=x;` leads to `(a=10,b=20)`.
Run 3: `if(x<=10)x=10;a=x;b=x;x=20;` leads to `(a=10,b=10)`.
Run 4: `if(x<=10)a=x;x=10;x=20;b=x;` leads to `(a=V ,b=20)`.
Run 5: `if(x<=10)a=x;x=10;b=x;x=20;` leads to `(a=V ,b=10)`.
Run 6: `if(x<=10)a=x;b=x;x=10;x=20;` leads to `(a=V ,b=V )`.

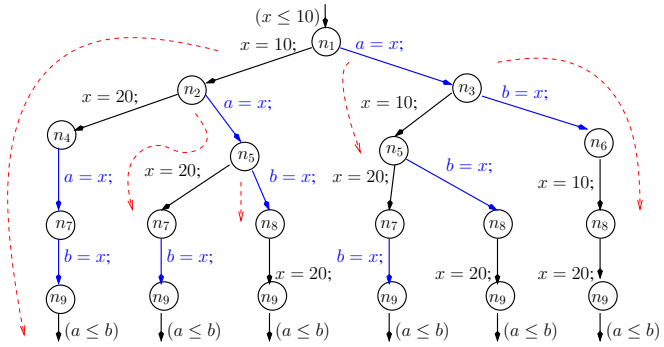


Figure 3.2: Our new method only explores one full run and four partial runs.

erations, *AGSE* needs to explore only those executions that have not be covered by these predicates, thereby leading to a sound reduction of the search space.

Now, I provide a step-by-step explanation of how *AGSE* works on this example:

- Run 1 is the first and only execution fully explored by *AGSE*, which goes through nodes n_1, n_2, n_4, n_7 in the graph in Figure 3.2 before executing `b=x;if(a<=b)`. Since it does not violation the assertion, *AGSE* summarizes the reason at n_9 and n_7, respectively, as follows:

$\mathsf{PS}[n_9] = (a \leq b)$ and $\mathsf{PS}[n_7] = (a \leq x)$. That is, as long as $(a \leq x)$ holds at node n_7, it would be impossible for the execution to reach the bad state.

- Run 2 goes through nodes n_1, n_2, n_5 before reaching n_7, where its new path condition is $pcon[n_7] = (V \leq 10)$ and symbolic memory is $\mathcal{M} =$`(a=10,x=20)`. Since $pcon[n_7] \rightarrow \mathsf{PS}[n_7]$ under $\mathcal{M}$, meaning the set of reachable states falls inside $\mathsf{PS}[n_7]$, continuing the current execution from n_7 would never lead to a bad state. Therefore, *AGSE* skips the remainder of this execution.
- Run 3 goes through nodes n_1, n_2, n_5, n_8 before reaching n_9, where its path condition again falls within $\mathsf{PS}[n_9]$. *AGSE* skips the remainder of this execution and updates the summary at n_8 and n_5 as follows: $\mathsf{PS}[n_8] = (a \leq b)$ and $\mathsf{PS}[n_5] = wp[n_7] \wedge wp[n_8] = (a \leq 20) \wedge (a \leq x)$. By conjoining the weakest preconditions along both interleavings $n_5 \rightarrow n_7$ and $n_5 \rightarrow n_8$, *AGSE* captures the summary common to both interleavings.
- Run 4 goes through nodes n_1, n_3 before reaching n_5, with the new path condition $pcon[n_5] = (V \leq 10)$ and symbolic memory $\mathcal{M} =$`(a=V,x=10)`. Since $pcon[n_5] \rightarrow \mathsf{PS}[n_5]$ under $\mathcal{M}$, *AGSE* skips the remainder of this execution, which would have led to Run 4 and Run 5 if it is allowed to continue.
- Run 6 goes through nodes n_1, n_3, n_6 before reaching n_8, where the new path condition falls within $\mathsf{PS}[n_8]$. Therefore, *AGSE* skips the remainder of this execution.
- At this moment, *AGSE* has completed the exploration.

Note that *AGSE conjoins* weakest preconditions from different interleavings at i-PP nodes such as n_5, but *unions* weakest preconditions from different thread-local paths at b-PP nodes (see Section 3.3.) Also, note that the amount of reduction achieved by *AGSE* depends on the program structure as well as the location of the assertion. For example, if `if(x>10)` changes to `if(x>11)`, *AGSE* would have to explore Run 5 instead of skipping it because $pcon[n_5] = (V \leq 11)$ would no longer be subsumed by $\mathsf{PS}[n_5] = (V \leq 10)$.

The running example demonstrates that *AGSE* differs from standard partial order reduction techniques such as DPOR [50] which could not prune away any of the six interleavings. Furthermore, *AGSE* also differs from the stateful state space exploration techniques commonly used in model checking [62, 65, 121], which record the forward reachable states explicitly during exploration to prevent visiting them again. Such methods would not be effective for the example in Figure 3.2 either because each of the six executions leads to a distinct state. In contrast, *AGSE* achieves a significant reduction due to the guidance of property specific information. In this sense, *AGSE* is a *property directed* reduction, whereas the aforementioned POR techniques are *property agnostic*.

However, it can be tricky to combine *AGSE*'s pruning method with the state-of-the-art DPOR algorithm. The main advantage of DPOR over static POR techniques lies in its dynamic update of backtrack sets, which uses runtime information to compute the dependency relation between shared variable accesses. Without taking any additional measure, pruning redundant executions may interfere with the dynamic update of backtrack sets in DPOR. Consider run 4 in Figure 3.2 as an example. If the execution is allowed to complete, when `b=x` is executed, thread T_2 will be added to the backtrack set of node n_3. However, if run 4 is terminated pre-maturely at node n_5 due to *AGSE*'s predicate summary-based pruning, thread T_2 would not be added to the backtrack set of node n_3 since `b=x` has been skipped. As a result, the DPOR algorithm would not explore run 6. Therefore, integrating DPOR with property specific pruning is a challenging task. *AGSE*'s solution to this problem is in Section 3.4.2.

AGSE's computation of predicate summaries at the thread interleaving merge point n_5 in Figure 3.2 shows that it is different from the prior work by Wachter et al. [122], and Chu and Jaffar [36]. Specifically, *AGSE* combines the summaries from all outgoing edges by conjoining them together, whereas existing methods do not merge interpolants at these i-PP nodes. Furthermore, *AGSE* differs from these existing methods in that they both implemented a symbolic POR whereas *AGSE* integrates the more scalable DPOR algorithm.

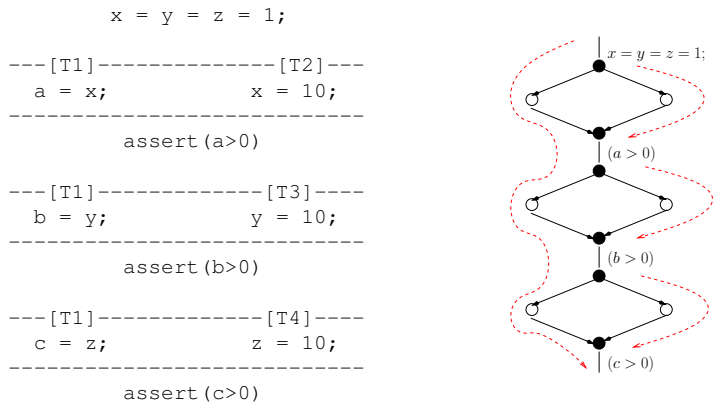


Figure 3.3: *AGSE* reduces the number of executions from 2^k down to $(k+1)$.

Next, I use the example in Figure 3.3 to demonstrate that *AGSE* has the potential to achieve an exponential reduction. In this contrived example, the interleaving of instructions in `{a=x, x=10}` is completely independent from `{b=y, y=10}` and `{c=z, z=10}`. Exploring all feasible executions results in 2^3 runs, each of which leads to a different final state. However, based on the abstract search space induced by the assertions, *AGSE* can reduce the exploration of eight runs down to one full run together with three partial runs, as marked by the '*' symbol in Figure 3.3. To further generalize the example, a program with k independent code segments would have 2^k distinct interleavings, which can be reduced by *AGSE* to $(k+1)$ executions.

3.3 Summarizing the Explored Executions

I first present how *AGSE* symbolically summarizes the reasons why explored executions cannot reach a bad state. In the next sections I will leverage the summary to prune away redundant

executions.

AGSE's summarization of the explored executions is based on the weakest precondition computation [44]. I differentiate the following two scenarios, depending on whether the execution encounters the assert statement or not.

- For each execution that encounters $assert(c)$ (and satisfies the condition c), *AGSE* computes the weakest precondition of the predicate c along this execution.
- For each execution that does not encounter the assert statement at all, *AGSE* computes the weakest precondition of the predicate `true` along this execution.

Since the weakest precondition is a form of Craig's interpolant [89], it provides a succinct explanation as to why the explored execution cannot reach the bad state guarded by $\neg c$.

Definition 3.1. Here I define the weakest precondition of the predicate ψ with respect to a sequence of instructions as follows:

- For the statement t: `v:=exp`, $WP(t, \psi) = \psi[exp/v]$;
- For the statement t: `assume(c)` , $WP(t, \psi) = \psi \wedge c$;
- For the sequence $t_1;t_2$, $WP(t_1; t_2, \psi) = WP(t_1, WP(t_2, \psi))$.

In the above definition, $\psi[exp/v]$ denotes the substitution of variable v in ψ with exp. As an example, consider the execution path in Table 3.1, which consists of three branch conditions and three assignments. Column 1 shows the control locations along the current path. Column 2 lists the sequence of executed instructions. Column 3 presents the weakest preconditions computed backwardly starting at l_6. Column 4 shows the rules applied during the computation.

Table 3.1: The weakeast precondicion computation along a program path.

Loc.	*Instruction*	*WP Computed*	*Rule Applied*
l_0	$if(a \leq 0)$	$(a \leq 0) \wedge (b \leq 0) \wedge (c \leq 0)$	$wp \wedge c$
l_1	$res := res + 1$	$(b \leq 0) \wedge (c \leq 0)$	$wp[exp/v]$
l_2	$if(b \leq 0)$	$(b \leq 0) \wedge (c \leq 0)$	$wp \wedge c$
l_3	$res := res + 2$	$(c \leq 0)$	$wp[exp/v]$
l_4	$if(c \leq 0)$	$(c \leq 0)$	$wp \wedge c$
l_5	$res := res + 3$	true	$wp[exp/v]$
l_6		true	*terminal*

3.3.1 Computing Predicate Summary at b-PP Nodes

Assume that the baseline symbolic execution procedure traverses the GIG in the depth-first search (DFS) order, meaning that it backtracks s, a branching pivot point (b-PP), only after exploring both outgoing edges $s \xrightarrow{assume(c)} s'$ and $s \xrightarrow{assume(\neg c)} s''$. This also includes the entire execution trees starting from these two edges. Let $wp[s']$ and $wp[s'']$ be the weakest preconditions computed from the two outgoing executions, respectively.

Following the classic definition of weakest precondition provided by Dijkstra [44], *AGSE* computes the weakest precondition at the b-PP node s as follows:

$$wp[s] := (c \wedge wp[s']) \vee (\neg c \wedge wp[s'']) \ .$$

Then, *AGSE* uses $wp[s]$ computed from these outgoing edges to update the global predicate summary $\mathsf{PS}[s]$.

For each global control state s *AGSE* defines a $\mathsf{PS}[s]$, which is the union of all weakest preconditions along the outgoing edges. Recall that the *Explore* procedure may visit each node s multiple times, presumably from different execution paths (from s_0 to s). Therefore, *AGSE* maintains a global map PS and updates each predicate summary entry $\mathsf{PS}[s]$ incrementally. Initially $\mathsf{PS}[s] = \mathsf{false}$ for every GIG node s. Then, *AGSE* merges the newly computed $wp[s]$ to $\mathsf{PS}[s]$ every

time the *Explore* procedure backtracks from s.

The detailed method for updating the predicate summary is highlighted in blue in Algorithm 2, which follows the overall flow of Algorithm 1, except for the following two additions:

- *AGSE* computes $wp[s]$ before the procedure backtracks from state s. At this moment, $wp[s]$ captures the set of all explored executions from s as a continuation of the current execution.
- *AGSE* updates the summary as follows: $\mathsf{PS}[s] = \mathsf{PS}[s] \vee wp[s]$. Here, $\mathsf{PS}[s]$ captures the set of execution trees as a continuation of all explored executions from s_0 to s, including $wp[s]$, which represents the newly explored execution tree.

3.3.2 Computing Predicate Summary at i-PP Nodes

In contrast to the straightforward computation of weakest precondition at the sequential merge point, the situation at the interleaving merge point is trickier. In fact, to the best of our knowledge, there does not exist a definition of weakest precondition in the literature for thread interleaving points. A naive extension of Dijkstra's original definition would be inefficient since it leads to the explicit enumeration of all possible interleavings. For example, assume that an i-PP node has two outgoing edges $s \xrightarrow{\gamma_1} s'$ and $s \xrightarrow{\gamma_2} s''$, one may attempt to define the weakest precondition at node s as follows:

$$((\gamma_1 <_{hb} \gamma_2) \wedge wp[s']) \vee ((\gamma_2 <_{hb} \gamma_1) \wedge wp[s'']) \ ,$$

where $(\gamma_1 <_{hb} \gamma_2)$ means that γ_1 executes before γ_2, $(\gamma_2 <_{hb} \gamma_1)$ means that we γ_2 executes before γ_1, and $wp[s']$ and $wp[s'']$ are the weakest preconditions along the two interleavings, respectively.

Although the above definition serves the purpose of summarizing the weakest preconditions along all explored executions from s, it has a drawback: the size of $wp[s]$ computed in this way can

Algorithm 2: Assertion Guided Symbolic Execution.

Initially: State stack $S = \emptyset$, summary $\mathsf{PS}[n] = \mathsf{false}$ for all node n; Start *Explore* with the symbolic state s_0.

Explore(State s)
begin
  S.push(s);
  if s *is a b-PP node* **then**
    $wp[s] := \mathsf{false}$;
    while $\exists t \in s.branch \backslash s.done$ **do**
      Explore($NextSymbolicState(s, t)$); // β event
      $s.done \leftarrow s.done \cup \{t\}$;
      $wp[s] \leftarrow wp[s] \vee computeWP\ (t, s')$;
    end
  else if s *is an i-PP node* **then**
    $wp[s] := \mathsf{true}$;
    while $\exists t \in s.enabled \backslash s.done$ **do**
      Explore($NextSymbolicState(s, t)$); // γ event (enhanced)
      $s.done \leftarrow s.done \cup \{t\}$;
      $wp[s] \leftarrow wp[s] \wedge computeWP\ (t, s')$;
    end
  else if s *is other sequential computation node* **then**
    Explore($NextSymbolicState(s, t)$); // α event
    $wp[s] \leftarrow computeWP\ (t, s')$;
  else
    // end of execution
    $wp[s] \leftarrow \mathsf{true}$;
  end
  $\mathsf{PS}[s] := \mathsf{PS}[s] \vee wp[s]$;
  S.pop();
end

$NextSymbolicState$(State s, Event t)
begin
  let $s = \langle pcon, \mathcal{M}, enabled, branch, done \rangle$;
  if t *is* ***halt*** **then**
    $s' \leftarrow$ normal_end_state;
  else if t *is* ***abort*** **then**
    $s' \leftarrow$ faulty_end_state;
  else if t *is* ***assignment*** $v := exp$ **then**
    $s' \leftarrow \langle pcon, \mathcal{M}[v \mapsto exp] \rangle$;
  else if t *is* $\mathbf{assume(c)}$ *and* $\mathcal{M}[pcon \wedge c]$ *is satisfiable* **then**
    $s' \leftarrow (\ pcon \rightarrow \mathsf{PS}[s]\ )$? early_termination_state : $\langle pcon \wedge c, \mathcal{M} \rangle$;
  else
    $s' \leftarrow$ infeasible_state;
  end
  return s';
end

$ComputeWP$(Event t, State s')
begin
  if t *is* $assume(c)$ **then**
    return $wp[s'] \wedge c$;
  else if t *is assignment* $v := exp$ **then**
    return $wp[s'][v \mapsto exp]$;
  else
    return $wp[s']$;
  end
end

quickly explode when there are a large number of threads. Recall that in a multithreaded program the number of outgoing edges at an i-PP node equals the number of enabled threads and the number of interleavings of k concurrent threads can be $k!$ in the worst case.

However, for the purpose of pruning redundant executions, the weakest precondition computation does not have to be precise to be effective. To mitigate the aforementioned interleaving explosion problem, *AGSE* uses the following definition, which can be viewed as an under-approximation of the naive definition:

$$wp[s] := \bigwedge_{1 \leq i \leq k} wp[s^k] \ ,$$

where each $wp[s^i]$ is the weakest precondition computed along one of the k outgoing edges of the form $s \xrightarrow{\gamma_i} s^i$, such that $1 \leq i \leq k$. Consider Figure 3.2 as an example. *AGSE* computes the weakest precondition at node n_5 by conjoining weakest preconditions at the two successor nodes n_7 and n_8. That is, $wp[n5] = wp[n_7] \wedge wp[n_8] = (a \leq 20) \wedge (a \leq x)$.

For the purpose of pruning redundant executions, conjoining weakest preconditions from different interleavings at i-PP nodes is a sound approximation. Although it may not capture all the explored executions and thus fail to prune certain redundant executions, all the pruned executions are guaranteed to be redundant.

3.4 Pruning the Redundant Executions

I present how *AGSE* leverages the predicate summaries to prune away redundant executions in this section.

3.4.1 Assertion Guided Pruning

To decide if *AGSE* can skip executions starting from a global control state s where s has been visited by $Explore$ previously through some executions from s_0 to s, but is reached again through a new execution, *AGSE* checks if the path condition of current path, $s.pcon$, can be subsumed by $\mathsf{PS}[s]$ under the current memory map $s.\mathcal{M}$. Intuitively, the $s.pcon$ represents the set of states reachable along the current execution from s_0 to s, whereas $\mathsf{PS}[s]$ represents the set of states from which it is impossible to reach the bad state.

Within the $NextSymbolicState$ procedure in Algorithm 2, *AGSE* checks for the pruning condition as follow:

- If $s.pcon \rightarrow \mathsf{PS}[s]$ holds under $s.\mathcal{M}$, extending the current execution beyond s would not lead to a bad state. Therefore, *AGSE* backtracks immediately by setting s' as an *early termination state*.
- Otherwise, there *may* exist an extension of the current execution beyond s to reach the bad state. In this case, *AGSE* needs to continue the forward symbolic execution as usual.

The validity of $s.pcon \rightarrow \mathsf{PS}[s]$ can be decided by checking the satisfiability of $(s.pcon \wedge \neg\mathsf{PS}[s])$ using an SMT solver. That is, $s.pcon \rightarrow \mathsf{PS}[s]$ holds if and only if $(s.pcon \wedge \neg\mathsf{PS}[s])$ is unsatisfiable.

AGSE's new pruning approach is complementary to partial order reduction techniques. POR is a generic reduction that relies solely on commutativity between concurrent operations. Therefore, two executions are considered equivalent as long as they result in the same program state. *AGSE*, in contrast, deploys assertions to guide the pruning. Therefore, even executions that result in different program states may still be regarded as equivalent.

Consider the GIG in Figure 2.1, which has 54 feasible executions (Assume that x++ is atomic in this example.). However, note that `a1:a= x++` and `b1:b= x++` do not commute, because from a state where x=0, for instance, executing `a1;b1` leads to a=0,b=1,x=2, but executing `b1;a1` leads to a=1,b=0,x=2.

Table 3.2: Applying various reduction techniques to Figure 2.1.

Reduction Technique	Number of Paths
None	54
Partial order reduction (POR)	34
Our predicate summary-based pruning method	18
Both POR and our new pruning method	13

As shown in Table 3.2, without applying any reduction technique, the program has a total of 54 distinct runs. Partial order reduction (POR) alone can reduce the 54 runs down to 34 runs. *AGSE*'s summary-based pruning method alone can reduce the 54 runs down to the 18 runs. Finally, applying both pruning method and POR can reduce the 54 runs down to 13 runs.

3.4.2 Interaction with DPOR

However, there is a caveat in combining our predicate summary-based pruning method with dynamic partial order reduction [50], because DPOR is a delicate algorithm that relies on the dynamic computation of the *backtrack sets*. Without taking precautions, naively pruning away redundant executions, even if they do not lead to the bad state, may deprive DPOR the opportunity to properly update its backtrack sets, thereby leading to unsound reduction.

As shown in Section 3.2, when the current execution is run 4 in Figure 3.2, by the time node n_5 is reached DPOR has not had the opportunity to update its backtrack set at n_3. Ideally, thread T_2 should be put into the backtrack set of n_3, that is, after *Explore* backtracks to n_3, it should proceed to explore run 6.

However, since $n_5.pcon \rightarrow \mathsf{PS}[n_5]$ along run 4, *AGSE*'s pruning method would force *Explore* to backtrack from n_5, thereby skipping the remainder of run 4 and run 5. Here, the technical challenge is how to properly update the backtrack set at node n_3 before *Explore* backtracks from n_5.

Fortunately, similar problems were encountered during the development of stateful DPOR algorithms [129]. In *AGSE* I follow the solution by Yang et al. [129]. Specifically, *AGSE* maintains two global tables, $RVar[s]$ and $WVar[s]$, for each global control state s. The $RVar$ table stores the set of global variables that have been read by some thread during previously explored executions starting from s. Similarly, the $WVar$ table stores the set of global variables that have been written to by some thread during previously explored executions starting from s. These two tables are updated at the same time the global PS table is updated.

For the example in Figure 3.2, after exploring run 1, run 2, and run 3, *AGSE* would have $WVar[n_5] = \{(x, T_1)\}$ representing that `x=20` has previously been executed by thread T_1 at some point after n_5. Similarly, *AGSE* has $RVar[n_5] = \{(x, T_2)\}$ representing that `b=x` has previously been executed by thread T_2 at some point after n_5.

Whenever procedure *Explore* decides to skip the execution tree from a node s, *AGSE* leverages the information stored in $WVar[s]$ and $RVar[s]$ to properly update the backtrack sets for DPOR. For example, the original DPOR algorithm waits until assignment `b=x` is executed by thread T_2 before it can update the backtrack set of n_3. Now, using the entry $(x, T_2) \in RVar[n_5]$, it can put thread T_2 into the backtrack set of n_3, as if `b=x` has been executed by thread T_2 at some point after n_5.

The correctness of this solution follows Yang et al. [129] in the context of stateful DPOR, which ensures that DPOR remains sound in the presence of assertion guided pruning. For more information on the dynamic update of backtrack sets, please refer to the original description of DPOR [50].

3.4.3 Proof of Correctness

Now, I state and prove the correctness of the overall algorithm. Let SE_{orig} be the baseline symbolic execution procedure described in Algorithm 1, and SE_{new} be the new symbolic execution procedure with predicate summary-based pruning, as described in Algorithm 2. Then SE_{new} is a sound reduction of SE_{orig} if it always reaches the same set of error states as SE_{orig}.

Theorem 3.2. *Given a program P and an error location E. The new symbolic execution procedure SE_{new} reaches E if and only if the original symbolic execution procedure SE_{orig} reaches E.*

Proof: I divide the proof into two steps. First, I prove that if SE_{new} reaches E, then SE_{orig} also reaches E. This is straightforward because SE_{new} explores a subset of the execution paths explored by SE_{orig}, as shown by a comparison of the two versions of $NextState$ in Algorithms 1 and 2.

Second, I prove that if SE_{orig} reaches E, then SE_{new} reaches E. I do this by contradiction. Assume that SE_{orig} can reach E along a path π but SE_{new} cannot. Since Lines 42–43 in Algorithm 2 are the only places where SE_{new} can skip a path, there must exist an event $\langle s, t, s' \rangle$ in path π such that $s.pcon \rightarrow \mathsf{PS}[s]$ holds under $s.\mathcal{M}$.

- Since path π is feasible, the subpath of π from s' to E must also be feasible. To skip π in SE_{new}, the subpath must have been explored and then summarized in $\mathsf{PS}[s']$, presumably when SE_{new} first explored the subpath.
- But if $\mathsf{PS}[s']$ already includes this common subpath from s' to E, by definition, SE_{new} must have reached the error block E. This contradicts our assumption that the new symbolic execution procedure SE_{new} cannot reach the error block E.

Therefore, the assumption is incorrect. The theorem holds.

3.5 Optimizations

In *AGSE*, the amount of entries in the summary table and the size of the logical formula in each entry may become an performance bottleneck. Since large logic formulas are expensive to compute and store, I would like to reduce the associated computational cost without affecting soundness of the overall procedure. Toward this end, I propose two optimizations.

3.5.1 Leveraging Static Program Slicing

The first optimization is to combine the assertion guided pruning with static program *slicing* to achieve a more significant state space reduction. Given an assertion statement st, *AGSE* defines the *slice* of st as the set of all statements in the program that may affect the result of st. The slice is computed based on two dependency relations: the control dependency relation and the data dependency relation. Intuitively, a statement st' is a control dependency of a statement st if the execution of st' determines whether st can be executed. Whereas a statement st'' is a data dependency of st if the execution of st'' may affect the data used in st.

```
1   if (p)
2   y = v;
3   z = w * 5;
4   if (q)
5   x = z * 2;
6   assert(x);
7
```

Figure 3.4: Example for static program slicing computation.

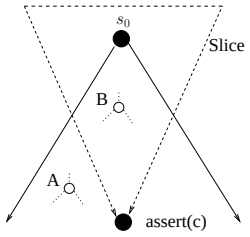


Figure 3.5: Using Type A and B nodes outside the slice.

Consider the example in Figure 3.4. The write to x at Line 5 has a *control dependency* at Line 4, and a *data dependency* at Line 3. The *slice* of Line 5 is defined as the transitive closure of its

control and data dependencies, which consists of Lines 3–5. In contrast, the branching statement at Line 1 and the write to `y` at Line 2 are irrelevant since their execution will *not affect* the value written to `x` at Line 5 nor the reachability of Line 5. Therefore, for the purpose of checking the assertion at Line 6, which is related to the value of x at Line 5, *AGSE* can simply ignore Lines 1–2. In other words, the slice of Line 5 (and Line 6) defines a sub-program producing an equivalent result as the full program as far as assertion checking is concerned.

AGSE combine static program slicing with symbolic execution as follows. First, *AGSE* computes the static program slice prior to the start of symbolic execution. Then, inside the symbolic execution procedure as described in Algorithm 2, for each *to-be-executed* b-PP or i-PP node s, *AGSE* checks if the corresponding branch condition or global operation belongs to the static slice of the assertion statement. If the answer is no, *AGSE* handles a pivot point s (which can be an i-PP or a b-PP) in one of the following ways depending on the node type as illustrated in Figure 3.5.

- **Type A:** If s is not on any path from s_0 to the assertion statement, *AGSE* treats each outgoing edge from s as if it is **halt**. In other words, *AGSE* stops the current execution and backtrack from s immediately. Note that backtracking will automatically trigger the computation of weakest precondition.
- **Type B:** If s is on some GIG path from s_0 to the assertion statement, *AGSE* cannot simply treat s as the end of the program since outgoing paths from s may still lead to the assertion failure. As shown in Figure 3.5, *AGSE* has to symbolically execute at least one of the outgoing edges from the Type B node, while skipping the other outgoing edges.

The correctness of this approach directly follows from the definition of slicing. For both Type A and Type B nodes outside the program slice, *which outgoing edge to execute* does not affect the reachability of the bad state. Due to the relative efficiency of the static slicing algorithm, the overhead of computing the slice is minimal compared to the subsequent symbolic execution procedure.

However, experiments show that, by leveraging static program slicing results, *AGSE* can significantly decrease of the number of executions to be explored, thus decreasing the complexity of the overall analysis.

3.5.2 Approximating the Summary Constraints

In general, any kind of underapproximation of $\mathsf{PS}[s]$ may be applied to Algorithm 2 to replace $\mathsf{PS}[s]$, while the soundness of the proposed pruning method still maintains. The optimization here is to heuristically reduce the computational cost associated with predicate summaries. Toward this end, I develop the following two underapproximations in *AGSE*.

First, *AGSE* uses a fixed-size global hash table to control the overall storage cost for PS. Note that two different global control locations s and s' may possibly map to the same entry in this hash table. Whenever such collision happens, instead of storing both summaries in a linked list for that entry, *AGSE* limits the overall cost by dropping one of them. That is, when $key(s) = key(s')$, *AGSE* heuristically removes one entry, effectively setting the corresponding predicate summary false.

Second, *AGSE* uses a fixed threshold to bound the size of each individual logical constraint for $\mathsf{PS}[s]$. In other words, when the predicate summary becomes too large, *AGSE* will stop adding new weakest-preconditions to it, thereby dropping all subsequently explored subpaths. That is,

$$\textbf{if } (\textbf{size}(\mathsf{PS}[s]) < \text{bnd}) \quad \mathsf{PS}[s] := \mathsf{PS}[s] \vee wp[s] \ .$$

This is again an underapproximation of $\mathsf{PS}[s]$.

A main advantage of this on-demand constraint minimization framework is that it allows various forms of underapproximations to be plugged into it without affecting the soundness proof of the overall algorithm. With underapproximations, it is possible that we may no longer be able to

prune away all redundant executions, however, *AGSE* can guarantee that all pruned executions are truly redundant. In particular, the baseline symbolic execution in Algorithm 1 (no pruning) can be viewed as an extreme form of underapproximation, where $\mathsf{PS}[s]$ is underapproximated to false for all global control locations.

3.6 Evaluation

I have implemented *AGSE* in *Cloud9* [38], which in turn builds upon the LLVM compiler [8] and the KLEE symbolic virtual machine [22]. Note that KLEE does not by itself support multi-threading, and although *Cloud9* has extended KLEE to support a limited number of POSIX thread routines, it does not attempt to cover all feasible thread interleavings. Indeed, *Cloud9* allows for context switches only before certain POSIX thread synchronizations but not before shared variable reads/writes. Furthermore, *Cloud9* does not support partial order reduction. Instead, it forks new executions every time specific POSIX synchronizations are encountered.

I have extended *Cloud9* to implement the baseline symbolic execution in Algorithm 1, which systematically explores both intra-thread paths and thread interleavings. Then, I implemented the DPOR algorithm [50]. Based on these extensions, I have implemented the new assertion guided pruning (Algorithm 2) with the optimizations presented in Section 3.5.

I have conducted experiments on two sets of benchmarks. The first set consists of multithreaded C programs from the 2014 Software Verification Competition (SV-COMP) benchmark [115] and programs from [48, 79]. The second set consists of two real multithreaded applications: *nbds* [93], a collection of lock-free data structures, and *nedmalloc* [94], a thread-safe malloc implementation. Each of these programs has between 40 to 6,500 lines of code, with a combined total of 40,291 lines of code. Each benchmark program is first compiled into LLVM bitcode using Clang, before

given to the symbolic execution tool with a set of user annotated variables as symbolic input.

Table 3.3: Summary of the experimental results.

			Cloud9		+DPOR		+DPOR+AG	
Name	LOC	Threads	Runs	Time (s)	Runs	Time (s)	Runs	Time (s)
fibbenchfalse1	44	2	924	61.4	48	2.0	15	1.8
fibbenchfalse2	44	2	–	>1800	628	36.2	34	3.9
fibbenchfalse3	44	2	–	>1800	8704	503.8	378	13.7
indexertrue	85	2	–	>1800	81	2.8	24	6.0
lazy01false	51	3	11	0.5	3	0.3	3	1.1
reorder2false1a	85	2	7	0.3	3	0.3	3	1.2
reorder2false1b	85	3	91	1.4	26	0.6	9	1.2
reorder2false1c	85	4	2421	89.1	205	3.2	39	1.6
reorder2false2a	85	2	23	0.6	14	0.5	14	1.5
reorder2false2b	85	3	479	8.9	233	5.0	64	2.2
sigmafalse1	49	2	12	0.4	6	0.3	2	1.2
sigmafalse2	49	3	180	3.2	50	1.0	2	1.2
sigmafalse3	49	4	4830	222.4	862	18.6	2	1.2
singletonfalse	57	4	60	1.1	24	0.6	19	1.1
stackfalse	120	2	527	8.6	236	3.9	49	2.8
stateful01true	55	2	6	0.4	6	0.4	5	1.2
twostage3false	129	3	4862	302.1	88	1.1	34	2.2
dekkertrue	55	2	–	>1800	280	3.6	6	1.5
petersontrue1	43	2	–	>1800	1052	22.7	64	2.7
petersontrue2	43	2	–	>1800	2566	86.6	85	8.1
readwritelktrue1	52	2	24	0.6	4	0.3	4	1.1
readwritelktrue2	52	4	–	>1800	–	>1800	436	14.9
timevarmutextrue	55	2	41	0.8	4	0.3	2	1.0
szymanskitrue	55	2	–	>1800	–	>1800	6	1.8
unveriftrue	40	2	–	>1800	221	2.9	27	1.7
bluetoothbad	88	2	–	>1800	1789	25.1	95	4.0
art-example	71	2	450	11.5	146	3.1	9	1.5
fsbenchbad	86	8	–	>1800	256	9.2	9	20.9
tickettrue	76	2	1062	19.6	274	4.8	44	1.9
accountbad	60	3	8	0.4	8	0.4	8	1.0
circularbufbad1	109	2	118	1.7	118	1.9	58	3.8
circularbufbad2	109	2	358	5.5	358	5.5	132	6.4
readreadwrite	50	3	96	1.4	19	0.5	3	1.1
queuefalse	167	2	252	3.9	252	3.8	26	3.9
nbds-slU1a	1942	2	–	>1800	133	8.9	5	7.8
nbds-slU1b	1942	2	–	>1800	–	>1800	76	16.2
nbds-slU1c	1942	2	–	>1800	–	>1800	202	35.2
nbds-slU2a	1942	2	–	>1800	241	25.3	29	12.8
nbds-slU2b	1942	2	–	>1800	–	>1800	118	24.5
nbds-slU2c	1942	2	–	>1800	–	>1800	717	164.8
nbds-skiplist	1994	3	–	>1800	–	>1800	1	25.1
nbds-hashw1a	2322	2	–	>1800	1339	167.4	123	177.8
nbds-hashw1b	2322	2	–	>1800	6501	1568.9	675	222.8
nbds-hashw1c	2322	2	–	>1800	–	>1800	2399	476.9
nbds-hashw2a	2234	2	–	>1800	5852	674.1	369	155.3
nbds-hashw2b	2234	2	–	>1800	–	>1800	1735	257.4
nbds-hashw2c	2234	2	–	>1800	–	>1800	4017	528.4
nbds-hash	2375	2	–	>1800	–	>1800	2283	333.8
nbds-list	1887	3	–	>1800	10274	1130.7	1	5.9
nedmalloc	6303	4	–	>1800	–	>1800	1	12.0
Average				**986.9**		**518.5**		**51.6**

Table 3.3 summarizes the results of the experimental evaluation. Columns 1–3 present the benchmark name, lines of C code, and the number of threads in each program. Columns 4–9 compare the performance of three different methods in terms of the number of explored runs and the total execution time. *Cloud9* denotes the baseline symbolic execution in Algorithm 1, +DPOR denotes the baseline algorithm with dynamic partial order reduction, and +DPOR+AG denotes *AGSE*, which augments the baseline algorithm with DPOR and assertion guided pruning. The runtime of *AGSE* includes the time to compute the slice. All tests used a maximum time of 30 minutes.

In the remainder of this section, I analyze the experimental results in more details for the following two questions:

1. How effective is *AGSE*'s pruning technique? Is it more effective than DPOR alone?
2. How scalable is *AGSE*? Is it practical in handling realistic C programs?

First, I show the comparison of Cloud9 and +DPOR+AG (*AGSE*) in two scatter plots in Figure 3.6, where the x-axis in each scatter plot represents the number of runs (or time) of the baseline algorithm (Cloud9), and the y-axis represents the number of runs (or time) of +DPOR+AG. Each benchmark program is represented by a dot in the scatter plots; dots below the diagonal lines are winning cases for our method. The results show that *AGSE* can significantly reduce the number of runs explored by symbolic execution as well as the overall execution time. In many cases, the baseline algorithm timed out after 30 minutes while *AGSE* finished in a few seconds.

Next, I show the comparison of +DPOR and +DPOR+AG in the scatter plots in Figure 3.7. The goal is to quantify how much of the performance improvement comes from *AGSE*'s new assertion guided pruning as opposed to DPOR. Again, dots below the diagonal lines are winning cases for +DPOR+AG over DPOR. For most of the benchmark programs, *AGSE* demonstrated a significant performance improvement over DPOR. For some benchmark programs, however, *AGSE* was slightly slower than +DPOR despite that it executed the same, or a smaller, number of runs. This

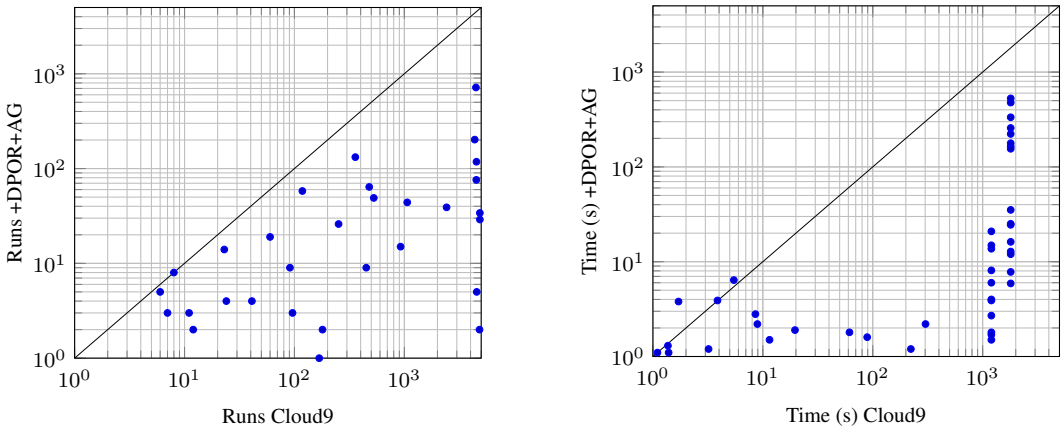


Figure 3.6: Scatter plots comparing *AGSE* with Cloud9.

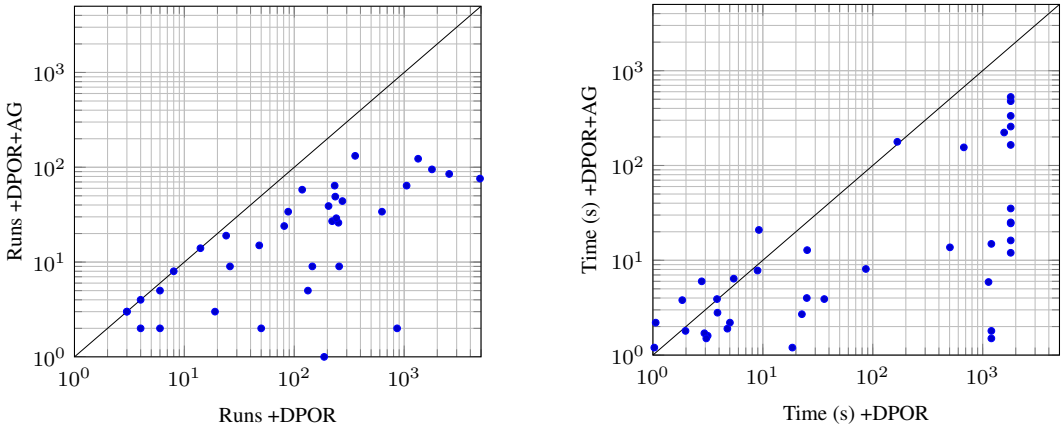


Figure 3.7: Scatter plots comparing *AGSE* with DPOR.

is due to the additional overhead of running the supplementary static slicing algorithm, as well as predicate summary-based pruning, which did not provide sufficient performance boost to offset their overhead.

However, it is worth noting that, where the combined optimization of slicing and pruning is able to bring a performance improvement, it often leads to a drastic reduction in the execution time compared to DPOR alone. For example, in *nedmalloc* (Table 3.3), *AGSE* was able to identify that the property does not depend on any shared variables. In such cases, it can safely skip exploring the entire interleaved state space and finish in just one run.

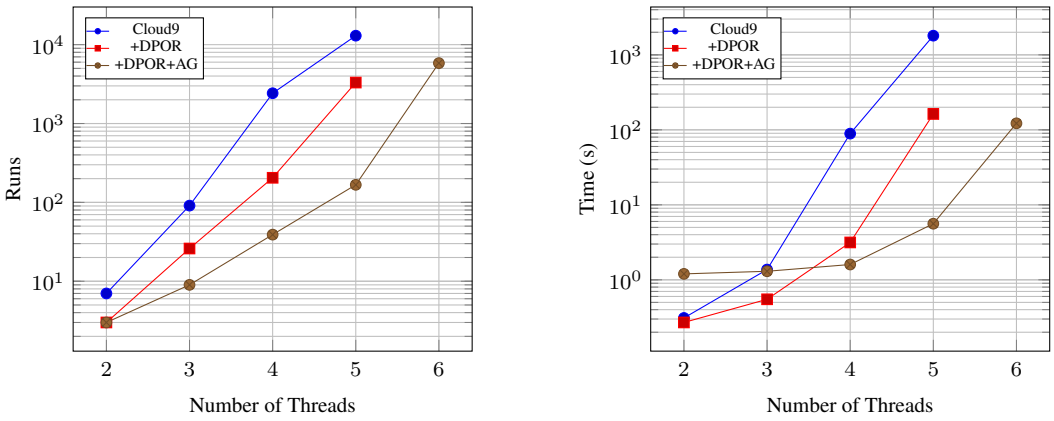


Figure 3.8: Parameterized results for *reorder2false* benchmark.

I also evaluated the growth trends of the three methods when the complexity of the benchmark program increase. Figure 3.8 shows the results of comparing the three methods on a parameterized program named *reorder2false*. In the two figures, the x-axis denotes the number of threads created in this parameterized benchmark while the y-axis presents, in logarithmic scale, the number of runs explored and the execution time in seconds. As shown by these two figures, the computational overhead of all three methods increases as the complexity of the program increases. However, *AGSE* increases at a significantly reduced rate compared to the two existing methods.

3.7 Conclusion

I have presented a predicate summary-based pruning method for improving symbolic execution of multithreaded program. The method is designed to work with the DPOR algorithm, and has the potential of achieving exponential reduction. I have implemented the method in emphCloud9 and demonstrated its effectiveness through experiments on multithreaded C/C++ benchmarks.

Chapter 4

Incremental Symbolic Execution

As software evolves, updates made from the addition of new features or patches may introduce new bugs. While some existing regression testing tools can leverage code changes between two software versions to reduce the testing cost, they focus primarily on test case selection or test case prioritization as opposed to the creation of new test cases. In contrast, symbolic execution is a technique for automatically generating new tests, and, in recent works [100, 106, 127], has been used in regression testing to reduce the overall cost for sequential software testing.

Change-impact analysis [125] has been widely applied in software testing and verification. The existing incremental symbolic execution tool, DiSE [100], uses an intra-procedural static change-impact analysis and then leverages it to reduce the cost of symbolic execution. The extension of DiSE, named iDiSE [106], improves it in two ways: by making the change-impact analysis inter-procedural, and by using dynamic calling-context information to increase accuracy. Yang et al.extend DiSE [127] to a property-guided symbolic execution procedure for checking assertions in evolving programs.

Change-impact analysis has been used in the context of program verification as well. Backes et al. [10] use a change impact analysis to improve the functional equivalence checking in regression verification. Specifically, the change-impact is used to focus on the equivalence checking of affected portions of the code. Similarly, SymDiff [81] focuses on proving assertions in the context of regression verification.

However, none of these previous techniques were designed for concurrent programs: they all target sequential software, and their extensions to concurrent programs remains non-trivial due to the inherent difficulties in analyzing thread interferences.

Specifically, prior works use conservative static analysis methods to estimate the impact of the code changes and then leverage the information to avoid re-executing program paths that are not affected by these code changes. However, these existing methods only handle code changes in sequential software. Furthermore, they rely on the overly conservative and intra-procedural analysis to estimate the change impacts, without making use the more accurate information available from previous runs.

In this chapter, I develop *Conc-iSE*: the first summary based incremental symbolic execution algorithm for concurrent programs.

Conc-iSE also differs from the prior works on regression testing of multithreaded programs [70, 117, 133]. In Jagannath et al. [70] and Yu et al. [133], the primary focus was on test case selection and test case prioritization, i.e., to detect certain concurrency bugs quicker by heuristically selecting test cases and scheduling them in certain orders, as opposed to generating new test cases. In contrast, *Conc-iSE* focuses on making symbolic execution incremental, which will benefit test case generation. Our method also differs from the work by Terragni et al. [117], which symbolically analyzes the alternative interleavings of some concrete executions based on the trace logs. Unlike our method, it does not perform symbolic execution based test input generation to explore both

intra-thread program paths and inter-thread interleavings.

4.1 Introduction

Figure 4.1 shows the overall flow of *Conc-iSE*. It takes both old (P) and new (P') program versions, together with a set of execution summaries of P as input, and iteratively explores new execution paths through P'. During the iterative exploration *Conc-iSE* uses supplementary information from both P (the execution summaries) and P' to perform the incremental analysis.

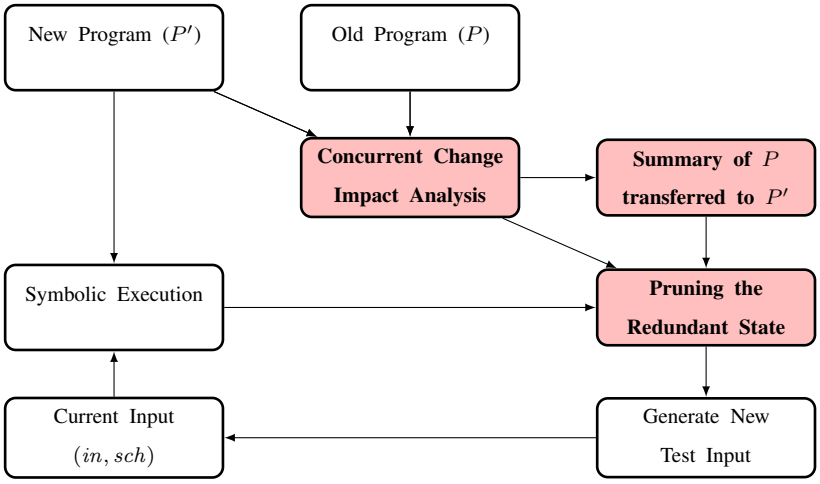


Figure 4.1: The summary based incremental symbolic execution.

The standard and non-incremental symbolic execution procedure is shown in the lower half of Figure 4.1, which starts from an arbitrary initial test (in, sch) of P' and repeatedly generates new tests for the new program P'. Here, in denotes the data input and sch denotes the thread schedule. I also assume the new program P' has a deterministic execution which is decided by the pair (in, sch). During symbolic execution, new states are generated to explore alternate branches and alternate thread schedules. For each new state, the symbolic execution engine generates a new pair (in', sch') containing the data input and thread schedule to reach the new state. In the non-

incremental approach, no information about previously explored executions in the old program version P as well as changes made to the new program version P' are used to decide if a new program state is redundant: program executions equivalent to behavior in P are re-explored in P'.

Incremental Symbolic Execution considers two program versions P and P' while assuming P is a prior version which has already been explored symbolically. The goal is to only explore the *new* behavior in P'. Prior work [100, 106] used a forward change-impact analysis, built on the idea of program slicing [124], to determine if a statement in P' was affected by a modification; only affected portions of the code in P' need to be explored again during symbolic execution.

However, performing a change-impact analysis using a conservative static analysis alone often results in the testing of redundant executions. This is because a conservative static analysis, such as program slicing, ignores the actual values of variables in the program. As shown in Section 4.2, even if a statement is modified (from P to P'), the paths affected by this modification may still be equivalent to some paths in the previous version. To define a more accurate equivalence class of execution paths, I make use of the predicate summaries (introduced in Section 3.3) from P while testing P', as opposed to performing only a conservative change-impact analysis. At a high level, the predicate summaries, defined at each global control state, capture the set of all explored executions starting from s (suffixes). The summaries are computed using a backward weakest-precondition computation. Notice that the *AGSE* method developed in Chapter 3 applies predicate summaries to one program only – it is a non-incremental symbolic execution algorithm.

Conc-iSE also has a more aggressive change-impact analysis that is inter-procedural and inter-thread, and therefore may be used to handle both sequential and concurrent programs. It consists of a forward analysis and a backward analysis. The forward change-impact analysis computes the program statements in P' that *may be affected* by the code changes to P. Similar to prior works on incremental symbolic execution [100, 106, 127], this is used to avoid portions of P' unaffected by

the code changes. Intuitively, if a code modification in P' only affects a small number of statements then much of P' is the same as P. In contrast, the backward change-impact analysis computes the set of statements that *may impact* some statements that are changed from P to P'. The backward analysis is used to determine if an predicate summary from the old version P can be carried over to the new version P'.

The combination of predicate summaries and forward and backward change-impact analyses, as well as their interaction with the baseline symbolic execution procedure, is shown in Figure 4.1. Prior incremental symbolic execution techniques [100, 106, 127] only handled sequential programs and only used a simpler version of the forward change-impact analysis. In contrast, *Conc-iSE* is the first incremental symbolic execution algorithm capable of handling concurrent programs. Specifically, when a new state in P' is generated, *Conc-iSE* checks both the change-impact information and the predicate summaries to see if the state is either in the unmodified section of the program, or if it is equivalent to some previously explored execution in P. If either condition is true, then the new state is redundant and can be skipped.

To sum up, I make the following contributions:

- I propose the first incremental symbolic execution algorithm capable of handling code changes in concurrent programs.
- I develop a new summary based algorithm for pruning away redundant paths and interleavings during incremental symbolic execution.
- I develop a new static analysis algorithm that leverages both forward and backward change-impact analysis to more aggressively estimate the impact of code changes.
- I implement the new method in *Conc-iSE* and evaluate it on a set of multithreaded programs to show the effectiveness at decreasing testing time.

4.2 Motivation

In this section, I use two examples to state the main ideas behind the new method.

4.2.1 Pruning with Change-Impact Analysis

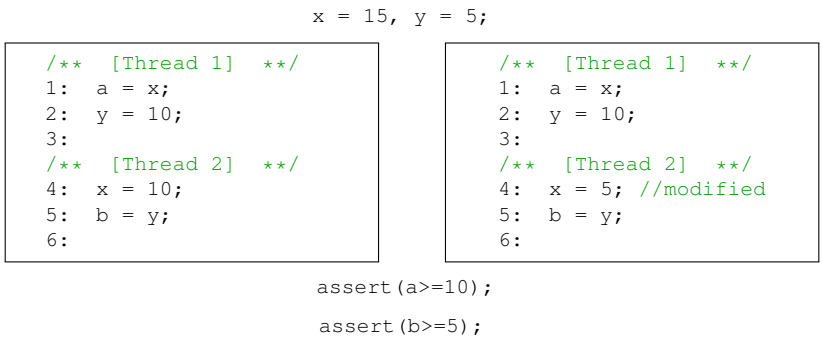


Figure 4.2: Example: Old version(left) and new version (right).

Consider the example in Figure 4.2; the old program, on the left-hand side, has two threads accessing shared variables x and y. The two variables are initialized to 15 and 5, respectively. After running both threads, the two assertions are checked. The new program is shown on the right-hand side; the only modification is on Line 4.

First, due to the sharing of variable x, Line 1 in the first thread is also affected though the modification of Line 4 is in the second thread. However, such impacted instructions cannot be identified by existing algorithms [100, 106, 127] since they are not always safe for concurrent programs. In contrast, *Conc-iSE* solves this problem.

Second, I want to stress that there are six possible executions of the program, as shown in the abstract state transition graphs in Figure 4.3. State-of-the-art partial-order reduction (POR) techniques can reduce the number of executions down to four. *Conc-iSE* does even better by reducing

the number of executions down to two.

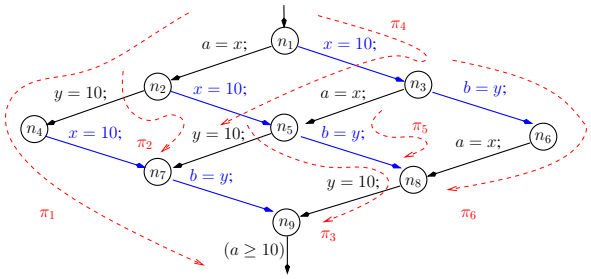


Figure 4.3: Interleaved executions of the program: $\pi_1, \ldots, \pi_6$.

Specifically, in Figure 4.3 each node denotes a global control state, e.g., $n_1 = (1, 4)$ means Thread 1 is at Line 1 and Thread 2 is at Line 4, while $n_2 = (2, 4)$ means they are at Lines 2 and 4, respectively. After applying POR, only four executions remain (marked by red dotted arrows), as shown on the left in Figure 4.4. The reason why π_2 and π_6 are skipped is because they are equivalent to π_1 and π_5, respectively. That is, executing the two independent instructions $y = 10$ and $x = 10$ in different orders always lead to the same result. (More specifically, these two executions are removed by a POR method with *sleep-sets*; refer to [51, 55] for more information.)

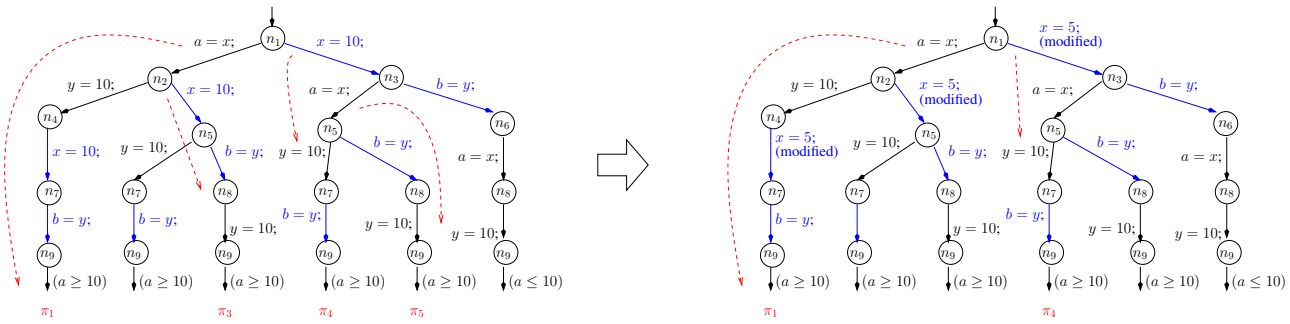


Figure 4.4: Executions explored by incremental symbolic execution (with POR).

By leveraging the concurrent change-impact analysis, *Conc-iSE* can identify even more redundant executions than POR. Specifically, since the code change in Line 4 does not impact Line 2 or Line 5 or *assert(b>=5)*, *Conc-iSE* does not re-explore the different execution orders of $y = 10$ and $b = y$. Because of this reason, as shown in the right-hand-side tree in Figure 4.4, *Conc-iSE* can reduce the

four executions further down to two (π_1 and π_4). Here I assume that assertions are embedded in the individual threads. As such, the assertion conditions always refer to local variables, or local copies of global variables. This is consistent with many prior works on POR [51]. It is also worth pointing out the assertions are important in this example: if the assertion is *assert(a>=b)*, then π_3 and π_5 may not be skipped. Details of the new change-impact analysis algorithm and its application to incremental symbolic execution are presented in Sections 4.3.1 and 4.3.2.

4.2.2 Pruning with Execution Summary

In addition to extending the prior change-impact analysis algorithm [100, 106, 127] from sequential programs to concurrent programs, I also develop an orthogonal pruning technique based on a backward change-impact analysis. That is, instead of computing the set of instructions that may be affected by the changes, *Conc-iSE* compute the set of instructions that may affect the changed instructions. Details of the backward change-impact analysis and its application are presented in Sections 4.3.1 and 4.3.3. Here I illustrate the main ideas using the example in Figure 4.5.

Consider the two versions of the sequential program; the old version is on the left, and the new version is on the right. The only modification is on Line 1; the condition is changed from $(x > 0)$ to $(x \geq 0)$. The forward change-impact analysis computes that the change has impacted all the other lines in the new program. Therefore, applying existing algorithms such as [100, 106, 127] would not help since they would re-explore all four paths.

Notice, however, that if dividing initial program states into three sets, denoted by $(x > 0)$, $(x = 0)$, and $(x < 0)$, that only when $(x = 0)$ does the modified program behave differently from the original program. In the old version, such case was handled by paths π_3 and π_4, but in the new version, it is handled by paths π_1 and π_2. Therefore, instead of re-exploring all paths, the new execution

```
1:  if (x>0)
2:      a = x+2;
3:  else
4:      a = x-2;
5:  if (y>0)
6:      b = a+1;
7:  else
8:      b = a-1;
9:  assert(b!=0);
```

```
1:  if (x>=0) //(modified)
2:      a = x+2;
3:  else
4:      a = x-2;
5:  if (y>0)
6:      b = a+1;
7:  else
8:      b = a-1;
9:  assert(b!=0);
```

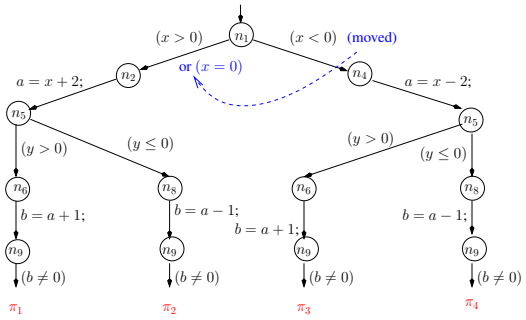


Figure 4.5: Not all four paths need to be re-explored though all instructions are impacted.

only needs to re-explore π_1 and π_2. The question then is how to figure out, algorithmically, that paths π_3 and π_4 are indeed redundant.

The solution in *Conc-iSE* is to compute, for each global control state s, a summary of all the explored executions starting from s. For example, the summary at n_4, with respect to *assert(b≠0)*, would be $\mathsf{PS}[n_4] = (y > 0) \wedge (x \neq 1) \vee (y \leq 0) \wedge (a \neq 3)$. This summary is created from the union of the weakest precondition of $(b \neq 0)$ along the two outgoing paths.

Since the code change in Line 1 does not affect the aforementioned weakest precondition computation, the summary can be carried over to the new program. During the analysis of the new program, an execution can stop as soon as the path condition, denoted, e.g., as $pcon[n_4] = (x < 0)$, falls within the set $\mathsf{PS}[n_4]$ of explored executions. This early termination can be done because if $pcon[n_4] \wedge \neg\mathsf{PS}[n_4]$ is unsatisfiable, re-exploring the executions starting from n_4 would not lead to any new error.

In Section 4.3.3, I will introduce another example used to showcase the power of an latest incremen-

tal symbolic execution technique, DiSE [100]. The example shows that the new summary-based technique in *Conc-iSE* can offer an even greater reduction.

4.3 The Incremental Approach

4.3.1 The Overall Algorithm

The incremental symbolic execution algorithm for concurrent programs, shown in Algorithm 3, has two significant changes compared to the baseline procedure in Algorithm 1. However, since the bulk of the recursive procedure remains the same, in Algorithm 3 I only highlight the parts that are significantly different.

First, the input of the symbolic execution procedure has changed. Instead of taking one program as input, the new procedure in *Conc-iSE* takes both old program version P and new version P' as the input. Prior to the symbolic execution of the new program P', *Conc-iSE* computes the forward impacted set $\mathsf{IS}_{\mathsf{fwd}}$ and the backward impacted set $\mathsf{IS}_{\mathsf{bwd}}$. In addition, *Conc-iSE* carries over the table PS of predicate summaries computed in P. For each state s, the set of explored executions starting from s is denoted $\mathsf{PS}[s]$.

Second, new Lines 27–29 and 32–34 are added inside *NextSymbolicState*. They leverage $\mathsf{IS}_{\mathsf{fwd}}$, $\mathsf{IS}_{\mathsf{bwd}}$, and $\mathsf{PS}[s]$ to decide, at each symbolic execution step ($s \xrightarrow{t} s'$), if all executions starting at the next state s' are redundant. Specifically, if $t.inst \notin \mathsf{IS}_{\mathsf{fwd}}$, the current branching statement is not in the impacted set. Since which branch to execute at s is immaterial, if one of the branches has previously been explored, *Conc-iSE* can force an early termination of the current execution.

Similarly, if $t.inst \notin \mathsf{IS}_{\mathsf{bwd}}$, the weakest precondition computation, upon which the execution

Algorithm 3: Incremental Symbolic Execution.

```
1  IS_fwd ← ComputeForwardImpactedSet(P, P′);
2  IS_bwd ← ComputeBackwardImpactedSet(P, P′);
3  PS[s] ← the summary at s computed in previous program P;
4  ...
5  NextSymbolicState(State s, Event t)
6  begin
7      let s = ⟨pcon, M, enabled, branch, done⟩;
8      if t is halt then
9          s′ ← normal_end_state;
10     else if t is abort then
11         s′ ← faulty_end_state;
12     else if t is assignment v := exp then
13         if t.inst ∉ IS_bwd and pcon ⟹ PS[s] then
14             s′ ← early_termination_state;
15         else
16             s′ ← ⟨pcon, M[v ↦ exp]⟩;
17         end
18     else if t is assume(c) and M[pcon ∧ c] is satisfiable then
19         if t.inst ∉ IS_fwd and the other branch was explored then
20             s′ ← early_termination_state;
21         else
22             s′ ← ⟨pcon ∧ c, M⟩;
23         end
24     else
25         s′ ← infeasible_state;
26     end
27     return s′;
28 end
```

summary is calculated, will not be affected by the code changes. Therefore, *Conc-iSE* can carry the summary $\mathsf{PS}[s]$ from P to P'. If the current path condition $pcon$, in the modified program, is subsumed by $\mathsf{PS}[s]$ then continuing the execution from s would lead to no new errors. In such case, *Conc-iSE* can enforce an early termination of the current execution.

For the example in Figure 4.5, the code change in Line 1 would only invalidate the summary $\mathsf{PS}[n_1]$. Therefore, although *Conc-iSE* cannot force an early termination at n_1, it can leverage the summaries at other nodes to prune away redundant executions. In particular, when the execution reaches either n_2 or n_4, *Conc-iSE* terminates the execution immediately. This is because both $pcon[n_2] \wedge \neg\mathsf{PS}[n_2]$ and $pcon[n_4] \wedge \neg\mathsf{PS}[n_4]$ are unsatisfiable. Specifically, *Conc-iSE* maintains

the following predicate summaries:

$$\mathsf{PS}[n_2] = (y > 0) \wedge (x \neq -3) \vee (y \leq 0) \wedge (x \neq -1)$$
$$\mathsf{PS}[n_4] = (y > 0) \wedge (x \neq 1) \vee (y \leq 0) \wedge (x \neq 3)$$

Furthermore, the path conditions are $pcon[n_2] = (x \geq 0)$, and $pcon[n_4] = (x < 0)$, respectively. Therefore, *Conc-iSE* can check $pcon[n_2] \wedge \neg\mathsf{PS}[n_2]$ as follows:

$$= (x \geq 0) \wedge ((y \leq 0) \vee (x = -3)) \wedge ((y > 0) \vee (x = -1))$$
$$= (x \geq 0) \wedge (y \leq 0) \wedge (y > 0)$$
$$= \mathsf{false}$$

It can also check $pcon[n_4] \wedge \neg\mathsf{PS}[n_4]$ as follows:

$$= (x < 0) \wedge ((y \leq 0) \vee (x = 1)) \wedge ((y > 0) \vee (x = 3))$$
$$= (x < 0) \wedge (y \leq 0) \wedge (y > 0)$$
$$= \mathsf{false}$$

The above checks indicate that no new errors can be detected by continuing the symbolic execution from n_2 and n_4. Therefore, *Conc-iSE* terminates the execution immediately without exploring any remaining paths. In the remainder of this section, I will present the algorithms for conducting the forward and backward change-impact analysis, as well as the construction and reuse of predicate summaries.

4.3.2 Change-Impact Analysis

The first important component of *Conc-iSE* is the detection and characterization of code changes, called the change-impact analysis (CIA) [84]. The identification of code changes requires com-

parison of two program versions by matching their representations, often in the form of flow graphs [103], tree representations [128], or locations in source files. *Conc-iSE* follows Person et al. [100] to define three types of changes: deleted, added, and modified.

Computing the Impacted Sets

Algorithm 4 presents the new change-impact analysis for multithreaded programs in *Conc-iSE*. The analysis takes two program versions P and P' as input and returns two impacted sets as output. One impacted set is $\mathsf{IS}_{\mathsf{fwd}}$, the forwardly impacted set, while the other impacted set is $\mathsf{IS}_{\mathsf{bwd}}$, the backwardly impacted set. The computation of the two sets consists of several steps.

First, *Conc-iSE* compares P and P' using a lightweight *diff* tool that computes the set Δ_{diff} of changed instructions (added, deleted, or modified). Since the remaining instructions exist in both programs, *Conc-iSE* constructs a map Δ_{map} that maps every unchanged instruction $inst \in P$ to its counterpart $inst' \in P'$.

Second, for each added instruction, denoted $inst_{add} \in \Delta_{diff}$, *Conc-iSE* performs a forward control- and data-dependency analysis in P' to identify all instructions depending on $inst_{add}$. Details of this analysis are presented in the next subsection. *Conc-iSE* also performs a backward control- and data-dependency analysis in P' to identify all instructions that $inst_{add}$ depends on. Here I denote the set of instructions as AI.

Third, for each modified instruction, denoted $inst_{mod} \in \Delta_{diff}$, *Conc-iSE* performs a forward control- and data-dependency analysis in P' to identify the instructions depending on $inst_{mod}$. *Conc-iSE* also performs a backward control- and data-dependency analysis to identify all instructions that $inst_{mod}$ depends on. Let's denote the set of instructions as MI.

Fourth, for each deleted instruction, denoted $inst_{del} \in \Delta_{diff}$, *Conc-iSE* performs both the forward

Algorithm 4: Forward and backward change-impact analysis.

```
1  Δ_diff ← Diff (P, P′);
2  Δ_map ← Map (P, P′, Δ_diff);
3  ComputeForwardImpactedSet(P, P′)
4  begin
5      IS_fwd ← { };
6      foreach inst ∈ Δ_diff do
7          if inst is added then
8              IS_fwd ← IS_fwd ∪ FwdDependencyAnalysis(P′, inst);
9          else if inst is modified then
10             IS_fwd ← IS_fwd ∪ FwdDependencyAnalysis(P′, inst);
11         else
12             // inst is deleted
13             impacted ← FwdDependencyAnalysis(P, inst);
14             foreach st ∈ impacted do
15                 st′ ← QueryMap (Δ_map, st);
16                 IS_fwd ← IS_fwd ∪ FwdDependencyAnalysis(P′, st′);
17             end
18         end
19     end
20     return IS_fwd;
21 end
22 ComputeBackwardImpactedSet(P, P′)
23 begin
24     IS_bwd ← { };
25     foreach inst ∈ Δ_diff do
26         if inst is added then
27             IS_fwd ← IS_fwd ∪ BwdDependencyAnalysis(P′, inst);
28         else if inst is modified then
29             IS_fwd ← IS_fwd ∪ BwdDependencyAnalysis(P′, inst);
30         else
31             // inst is deleted
32             impacted ← BwdDependencyAnalysis(P, inst);
33             foreach st ∈ impacted do
34                 st′ ← QueryMap (Δ_map, st);
35                 IS_fwd ← IS_fwd ∪ BwdDependencyAnalysis(P′, st′);
36             end
37         end
38     end
39     return IS_bwd;
40 end
```

and backward control- and data-dependency analysis to compute the set of instructions depending on $inst_{del}$, and the set of instructions that $inst_{del}$ depends on. Let's denote this set as DI.

Fifth, *Conc-iSE* iteratively computes AI, MI, and DI, using the results from the previous step as input, until the computation reaches a fixed point. Furthermore, for each deleted instruction in DI,

Conc-iSE retrieves its counterpart in P' by querying the Δ_{map}; the results form a new set DI'.

Finally, the union of AI, MI, and DI' forms the complete set of impacted instructions IS.

Algorithm 4 formalizes the above descriptions by dividing the computation of $\mathsf{IS}_{\mathsf{fwd}}$ and $\mathsf{IS}_{\mathsf{bwd}}$ into two separate routines. These routines, in turn, rely on two subroutines (described in Section 40) to perform the control- and data-dependency analysis for concurrent programs.

Inter-Thread Dependency Analysis

The control- and data dependency relations are computed by a constraint-based, flow- and context-insensitive analysis. Largely, the analysis follows traditional approaches [49, 68]: It calculates control-dependencies using post-dominance, and data-dependencies by the transitive closure of use-def chains. The main contributions here are reasoning about the soundness of this analysis in a concurrent setting, and applying it in a change-impact analysis.

First, an informal intuition behind control- and data-dependencies is: a statement s_2 is control dependent on a statement s_1 if the computation of s_1 determines if s_2 is executed. For example, in `if (c) x++;` the statement `x++` is control-dependent on `if (c)` (specifically, on the value of the predicate `c`). On the other hand, s_4 is data dependent on s_3 if the computation of s_3 influences the computation of s_4. For example, in `a = x; b = a + y;` the statement `b = a + y` is data-dependent on the statement `a = x` since the value of `a` determines the value of `b`.

In terms of soundness, *Conc-iSE* considers all ordering of statements since its data-dependency analysis is flow-insensitive. As a result, any statement from any thread can, effectively, execute at any time. This over-approximates the potentially more restrictive scheduling constraints, which ensures the soundness of the analysis for multithreaded programs.

```
1 int x = 0;
2 void thread1() {
3   int t1 = x;
4   create(thread2);
5 }
6 void thread2() {
7   x = 5;
8 }
```

Figure 4.6: Example for false data-dependencies across threads.

However, using a flow-insensitive analysis, while sound, results in false dependencies across threads. Consider the program in Figure 4.6: thread one reads the value of `x` and then creates thread two which writes to `x`. In a flow-insensitive analysis, the read in thread one is data-dependent on the write in thread two. But, the write can never be visible to thread one: thread two does not exist until after the read.

To capture this situation, *Conc-iSE* uses the *happen-before* relation. A statement s_1 happens-before a statement s_2 if on all program executions s_1 executes before s_2, e.g., the statement `create(thread2)` happens-before `x = 5`.

Conc-iSE refines the flow-insensitive data-dependency analysis as follows: if s_1 happens-before s_2 then s_1 must not be data-dependent on s_2. This is sound because the happens-before relation ensures there does not exist a program path from s_2 to s_1 so, s_1 cannot witness the effect of s_2. Currently, *Conc-iSE* deduces happens-before constraints from thread creation. This approach is similar to recent work using happens-before to refine data race detection [91, 92].

In the implementation *Conc-iSE* first builds the DATADEP relation, where $(a, b) \in$ DATADEP means the variable a is data-dependent on the variable b. To compute the relation, *Conc-iSE* first scan the program to generate a set of input items for the DATADEP relation.

To generate the input items, *Conc-iSE* use the structure of each instruction to determine the data-dependency, similar to prior constraint-based analyses for Java [18, 90, 105]. Here I use the binary

operation to showcase how *Conc-iSE* handles it; other instructions are handled similarly. A binary operation $r = op\ v_1\ v_2$, where r, v_1, and v_2 are variables and op is an operator, such as add applies op to v_1 and v_2 to produce r. So, the input items to DATADEP are (r, v_1) and (r, v_2).

Similarly, *Conc-iSE* provides input items to the happens-before relation from thread creation sites. Within a thread, *Conc-iSE* determines the happens-before relation using dominance and reachability on the threads control-flow graph. If s_1 dominates s_2 and s_1 is not reachable from s_2, then s_1 happens-before s_2. Dominance ensures that on all paths to s_2 contain s_1; reachability ensures that there is no path from s_2 to s_1. All in all, this ensures that s_1 always occurs before s_2.

Conc-iSE then computes the transitive closure of the happens-before and DATADEP relations and use happens-before to filter false dependencies from DATADEP. Finally, the forward (resp. backward) dependency analysis on some statement s is the forward closure from s of the combination of the control- and data-dependency relations, often called the slice [124].

4.3.3 Redundant Path Pruning

The second important component of the new incremental analysis is the pruning of executions that have already been explored. This section explains how to reuse the predicate summaries in the new program P' to prune away redundant executions whereas the computation of the predicate summaries in P is in Section 3.3.

First, the symbolic execution of P stores the predicate summaries [61] of all the explored executions in a table denoted PS, where each entry $\mathsf{PS}[s]$ stores a logical formula that represents all explored executions (suffixes) starting from s. Then, during the symbolic execution of P', *Conc-iSE* leverages the backward analysis result to decide if the summaries can be carried over to P'.

Reusing the Execution Summaries

Prior to using the summary table computed in P in the new program P', *Conc-iSE* checks if recent code changes have invalidated some of these summaries. If the answer is no, *Conc-iSE* can safely reuse them to prune away redundant executions in P'. For example, in Figure 4.5, since the only change is at Line 1, i.e., from *if (x>0)* to *if (x>=0)*, the weakest precondition computation is not affected at all other nodes except for n_1. In other words, *Conc-iSE* can reuse the previously computed summaries at all the other nodes.

The method for leveraging the summaries to prune away redundant executions lays at Lines 13–14 and Lines 19–20 in Algorithm 3. Here, $pcon$ represents the set of forwardly reachable states along the current execution, while $\neg\mathsf{PS}[s]$ represents the set of states that may lead to some previously unexplored errors. If the intersection is empty, however, there is no need to continue the current execution beyond s. In the actual implementation, the validity of $(pcon \implies \mathsf{PS}[s])$ is decided by checking the satisfiability of its negation, $(pcon \wedge \neg\mathsf{PS}[s])$, which can be solved efficiently by existing SMT solvers.

To demonstrate the advantages of *Conc-iSE* over existing techniques, I apply it a motivating example used in the DiSE [100] paper, called WBS, which has been used to showcase the pruning power of the DiSE tool. Since DiSE was designed for sequential programs, the WBS example, shown in Figure 4.7, is also a sequential program. Nevertheless, *Conc-iSE* can handle it as a concurrent program with a single thread.

In the WBS example, there is only one code change at Line 2, where the guard is changed from (*PedalPos* == 0) to (*PedalPos* <= 0). The red rounded rectangles represent the impacted *Control Flow Graph* (CFG) nodes in P', while the white rounded rectangles (except n_{begin} and n_{end}) represent nodes that are not impacted by the change. The baseline symbolic execution procedure needs

```
1:  int AltPress:=0; Meter:=2
    procedure UPDATE(int PedalPos, int BSwitch, int
    PedalCmd)
2:      if PedalPos<=0 then //(modified)
3:          PedalCmd += 1
4:      else if PedelPos == 1 then
5:          PedalCmd += 2
6:      else PedalCmd = PedalPos
7:
8:      PedalCmd = PedalCmd + 1
9:
10:     if BSwitch == 0 then
11:         Meter = 1
12:     else if BSwitch == 1 then
13:         Meter = 2
14:
15:     if PedalCmd == 2 then
16:         AltPress = 0
17:     else if PedalCmd == 3 then
18:         AltPress = 1/4
19:     else AltPress == 1/2
```

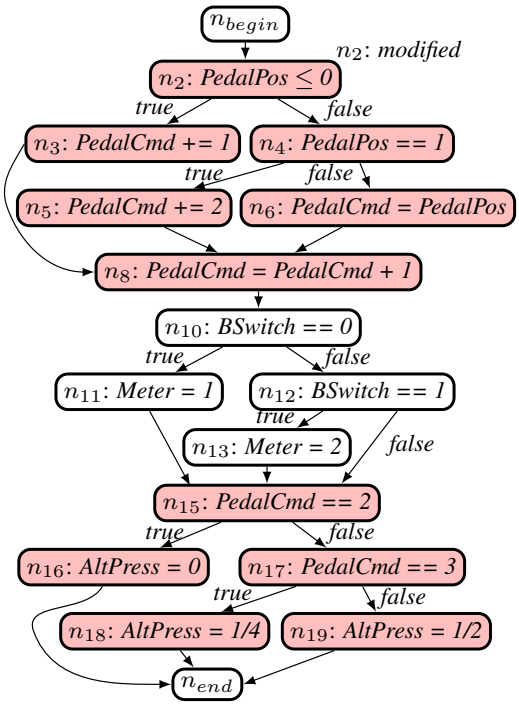


Figure 4.7: The WBS example taken from DiSE [100].

to explore all twenty-one paths whereas DiSE only needs to explore seven paths (Table 4.1), due to the reduction based on its forward impact analysis. That is, the change at node n_2 does not impact the nodes n_{10}, n_{11}, n_{12} and n_{13}.

Table 4.1: Comparing the paths explored by DiSE and *Conc-iSE*.

π	Explored by DiSE	Explored by *Conc-iSE*
1	$\{n_2, n_3, n_8, n_{10}, n_{11}, n_{15}, n_{16}\}$	partial (up to n_3)
2	$\{n_2, n_3, n_8, n_{10}, n_{11}, n_{15}, n_{17}, n_{18}\}$	skipped
3	$\{n_2, n_3, n_8, n_{10}, n_{11}, n_{15}, n_{17}, n_{19}\}$	skipped
4	$\{n_2, n_4, n_5, n_8, n_{10}, n_{11}, n_{15}, n_{16}\}$	partial (up to n_4)
5	$\{n_2, n_4, n_5, n_8, n_{10}, n_{11}, n_{15}, n_{17}, n_{18}\}$	skipped
6	$\{n_2, n_4, n_5, n_8, n_{10}, n_{11}, n_{15}, n_{17}, n_{19}\}$	skipped
7	$\{n_2, n_4, n_6, n_8, n_{10}, n_{11}, n_{15}, n_{16}\}$	skipped

However, there is still redundancy among the seven executions explored by DiSE. As shown in the third column of Table 4.1, certain common subpaths are explored repeatedly. For example, $\{n_8$, n_{10}, n_{11}, n_{15}, $n_{16}\}$ is an already-explored subpath in π_1 but it is re-explored in π_4 and π_7, also, $\{n_{10}$,

n_{11}, n_{15}, n_{17}, n_{18}} is an already-explored subpath in π_2 but it is re-explored in π_5, and {n_{10}, n_{11}, n_{15}, n_{17}, n_{19}} is an already-explored subpath in π_3 but it is re-explored in π_6. In contrast, *Conc-iSE* can reduce the seven executions further down to two executions (Column 3 in Table 4.1).

Table 4.2: Execution summaries computed for P in *Conc-iSE*.

Entry	Summary
$\mathsf{PS}[n_{19}]$	true
$\mathsf{PS}[n_{18}]$	true
$\mathsf{PS}[n_{17}]$	$((PedalCmd{==}3)\wedge\mathsf{PS}[n_{18}])\vee((PedalCmd\neq 3)\wedge\mathsf{PS}[n_{19}])$ = true
$\mathsf{PS}[n_{16}]$	true
$\mathsf{PS}[n_{15}]$	$((PedalCmd{==}2)\wedge\mathsf{PS}[n_{17}])\vee((PedalCmd\neq 2)\wedge\mathsf{PS}[n_{17}])$ = true
$\mathsf{PS}[n_{13}]$	$= \mathsf{PS}[n_{15}][2/Meter]$ = true
$\mathsf{PS}[n_{12}]$	$((BSwitch{==}1)\wedge\mathsf{PS}[n_{15}])\vee((BSwitch\neq 1)\wedge\mathsf{PS}[n_{15}])$ = true
$\mathsf{PS}[n_{11}]$	$= \mathsf{PS}[n_{15}][1/Meter]$ = true
$\mathsf{PS}[n_{10}]$	$((BSwitch{==}0)\wedge\mathsf{PS}[n_{11}])\vee((BSwitch\neq 0)\wedge\mathsf{PS}[n_{12}])$ = true
$\mathsf{PS}[n_8]$	$= \mathsf{PS}[n_{10}][(PedalCmd+1)/PedalCmd]$ = true
$\mathsf{PS}[n_6]$	$= \mathsf{PS}[n_8][PedalPos/PedalCmd]$ = true
$\mathsf{PS}[n_5]$	$= \mathsf{PS}[n_8][(PedalCmd+2)/PedalCmd]$ = true
$\mathsf{PS}[n_4]$	$((PedalPos{==}1)\wedge\mathsf{PS}[n_5])\vee((PedalPos\neq 1)\wedge\mathsf{PS}[n_6])$ = true
$\mathsf{PS}[n_3]$	$= \mathsf{PS}[n_8][(PedalCmd+1)/PedalCmd]$ = true
$\mathsf{PS}[n_2]$	$((PedalPos\leq 0)\wedge\mathsf{PS}[n_3])\vee((PedalPos{>}0)\wedge\mathsf{PS}[n_4])$ = true

Specifically, during the symbolic execution of the old program P, the predicate summaries at n_{17}, n_{15}, n_{10}, n_4 and n_2 are incrementally constructed, and Table 4.2 shows the summary table of P in terms of these locations.

Then, in the symbolic execution of the new program P', *Conc-iSE* first applies the forward change-impact analysis for the modification in Line 2 (which is the same as in DiSE), and then apply the backward change-impact analysis, which indicates that the summary is invalid only at node n_2 (immediately before Line 2); for all other nodes, *Conc-iSE* can safely reuse the summaries since these nodes are not in the backward slice of n_2.

By checking the validity of $pcon[s] \implies \mathsf{PS}[s]$ for the nodes $s \neq n_2$ during the execution, *Conc-iSE* reduces the seven runs further down to two partial runs.

The execution on P' starts by visiting n_2. As the summary at n_2 is invalid, since it is in the backwards impacted-set, the execution continues exploring without checking the summary. Con-

sider that the *true* branch of n_2 is first selected; execution proceeds until reaching the next assignment statement at n_3. Noticing that the summary at n_3 is still valid and $(pcon[n_3] \wedge \neg \mathsf{PS}[n_3])$ = (*PedalPos*$\leq$*0*)$\wedge$ $\neg$true = false, the execution stops here, generates the first partial run $\{n_2, n_3\}$, and backtracks to n_2.

Next, the *false* branch of n_2 is selected and the execution runs until the following branch statement at n_4. As $(pcon[n_4] \wedge \neg \mathsf{PS}[n_4])$ = (*PedalPos*>*0*)$\wedge$ $\neg$true = false, then the execution also stops and generates the second partial run $\{n_2, n_4\}$ before backtracking to n_2. Since the two outgoing edges of n_2 have been explored and n_2 is the entry of the program, the whole execution on P' terminates.

Therefore, the two runs explored by *Conc-iSE* are $\pi_1 = \{n_2, n_3\}$ and $\pi_4 = \{n_2, n_4\}$, as shown in Column 3 of Table 4.1.

4.4 Evaluation

Conc-iSE builds upon the LLVM compiler [82] and the *Cloud9* [38] symbolic execution engine. *Cloud9* relies on KLEE [23] as the backend. I extended *Cloud9* to soundly support POSIX threads; the original implementation only coarsly considered different interleavings at synchronization operations. In contrast, the baseline symbolic execution procedure in *Conc-iSE* schedules threads at a finer granularity (e.g., the memory accesses) and ensures that all interleavings are systematically explored. Furthermore, I have implemented the dynamic partial-order reduction (DPOR) algorithm [51], which *Cloud9* does not originally support.

In addition, I have implemented the forward and backward change impact analysis to provide guidance to the incremental symbolic execution algorithm. The implementation of the change-impact analysis follows on the construction of program dependence graphs similar to [49] and [68].

The change-impact analysis also contains a flow-insensitive pointer analysis capable of handling multithreaded programs. The analysis is constraint-based and works on LLVM bitcode. The Z3's μZ [64] fix-point solver is used to compute the fix-point of the constraints.

To share the summary information between the symbolic execution of different program versions, *Conc-iSE* uses the Memcached Distributed Cache as an external persistent storage. The summaries are computed and encoded in KLEE KQuery formula format during the symbolic execution. After executing the original program, summaries are serialized as binary sequences for Memcached storage. Before running the new program, they are loaded into main memory and mapped to the corresponding global control locations. *Conc-iSE* also has a *summary renewal* mechanism to determine if the code changes have invalidated the summaries of some global control locations, and reset these summaries to false if such situation happens.

4.4.1 Subjects and Methodology

Experiments are conducted on two sets of benchmarks. The first set consists of multi-threaded C programs from the 2015 Software Verification Competition (SV-COMP) benchmarks [116] and C programs from [48, 79]. The second set includes a collection of lock-free data structure applications (*nbds-list*, *nbds-skiplist* and *nbds-hashtable*) from [93]. Each of these benchmark programs has between 50 to 2,500 lines of code, with a total of 34,219 lines. Each benchmark program is first compiled to LLVM bitcode by Clang, before given to the symbolic execution engine.

For the C programs from [48, 79, 116], since there are no different versions of these programs available online, I manually made three types of mutants to the programs, acting as modified, deleted and added statements. For benchmark programs from [93], I studied the evolution history of them from the code repository, and selected some real updates from the developer team as the

changes to those programs.

4.4.2 Experimental Results

Table 4.3 summarizes the experimental results of the evaluation. The program name, version, amount of changes, lines of code, and the thread number for each program are shown in columns 1–5. Columns 6–11 compare the experimental performance of three different methods in terms of the number of explored program runs as well as the total execution time. *Full exploration* denotes the baseline symbolic execution procedure introduced in Algorithm 1, while *DPOR* denotes the baseline symbolic execution augmented with dynamic partial order reduction, and *Incremental* denotes our new method, which augments the baseline procedure with dynamic partial order reduction, change-impact analysis, and summary-based pruning. The static analysis time and summary computation time are also included in the execution time of *Incremental* method. All tests used a maximum time threshold of 30 minutes.

The remainder of this section discusses the experimental results in more details, for the two research questions:

1. How effective is the proposed *Incremental* technique?
2. How scalable is *Conc-iSE*? Can it handle realistic C programs?

First, two scatter plots in Figure 4.8 compares the experimental performance between *Full Exploration* and *Incremental*. The x-axis of each scatter plot denotes the execution time (or number of runs) of the baseline symbolic execution (Full Exploration), while the y-axis denotes the execution time (or number of runs) of *Conc-iSE* (Incremental). In the scatter plots, each dot represents a benchmark program, and the dots below the diagonal lines are the winning cases of *Incremental*. Figure 4.8 shows that *Incremental* can significantly reduce the number of runs explored by sym-

Table 4.3: Experimental results of concurrent benchmarks.

						Full		DPOR		CIA		SCIA (new)	
Name	Version	LOC	# Changes	Impacted (%)	Threads	# Runs	Time (s)	# Runs	Time (s)	# Runs	Time (s)	# Runs	Time (s)
fibbench	v1	65	1	0.0	2	924	16.6	48	0.8	1	0.3	1	0.3
	v2	66	2	10.6		–	>1800	142	2.0	142	2.2	22	0.9
	v3	67	2	13.6		–	>1800	628	12.1	628	12.0	34	2.4
	v4	67	2	17.9		–	>1800	3943	160.3	3943	161.4	39	2.5
	v5	68	3	2.9		–	>1800	1420	30.3	10	0.4	7	0.3
account	v1	68	1	8.8	3	749	14.2	106	1.8	106	1.7	38	0.9
	v2	69	1	10.1		5838	370.1	208	3.5	208	3.4	81	1.6
	v3	70	3	14.3		1773	50.5	168	2.8	168	2.7	55	1.2
	v4	70	1	6.6		1773	47.8	168	2.7	14	0.5	11	0.4
	v5	71	2	6.6		13407	1642.4	325	5.3	11	0.4	9	0.3
lazy01	v1	58	1	10.3	3	156	2.4	12	0.4	12	0.4	9	0.3
	v2	59	2	11.9		1399	36.1	43	0.8	43	0.8	18	0.5
	v3	61	4	11.5		8313	624.1	71	1.2	71	1.2	18	0.5
	v4	62	2	1.6		8313	625.3	71	1.1	2	0.3	2	0.1
	v5	61	4	13.1		–	>1800	211	3.1	179	2.5	26	0.6
indexer	v1	85	1	22.4	2	–	>1800	729	29.6	729	30.5	33	20.3
	v2	85	1	16.5		–	>1800	81	2.5	5	0.4	5	0.4
	v3	86	2	23.3		–	>1800	90	2.5	90	2.6	30	5.2
	v4	87	2	2.3		–	>1800	90	2.5	1	0.3	1	0.3
	v5	88	2	22.7		–	>1800	1314	41.2	1314	43.7	563	53.9
readreadwrite	v1	59	1	5.1	3	191	2.7	36	0.7	1	0.2	1	0.3
	v2	60	3	13.3		105	1.6	10	0.4	4	0.3	4	0.3
	v3	63	3	12.7		728	13.7	34	0.7	34	0.8	20	0.5
	v4	63	1	12.7		728	14.0	34	0.7	9	0.3	8	0.3
	v5	67	5	19.1		5444	175.1	101	1.6	22	0.6	18	0.5
stateful01	v1	65	2	9.2	2	88	1.4	37	0.7	37	0.7	12	0.4
	v2	67	1	9.0		296	4.3	117	1.7	46	0.8	15	0.4
	v3	68	2	10.3		3267	120.8	675	11.4	327	4.8	22	0.5
	v4	68	1	16.2		3267	119.6	675	10.1	675	9.9	71	1.0
	v5	68	1	16.2		3267	121.3	675	8.9	675	8.9	42	0.7
reorder	v1	94	1	6.4	2	1190	17.8	38	0.7	34	0.6	30	0.5
	v2	92	2	6.5		222	2.7	15	0.5	11	0.4	9	1.2
	v3	94	2	7.4		2903	72.1	61	1.2	38	0.7	28	0.5
	v4	96	2	7.4		4698	176.1	125	1.9	125	1.9	32	0.7
	v5	97	3	7.2		9557	273.1	68	1.2	53	0.9	39	0.8
twostage3	v1	141	1	2.8	3	4862	286.8	101	1.6	7	0.3	7	0.3
	v2	142	1	5.6		5878	298.4	148	2.2	148	2.4	79	1.3
	v3	141	2	5.7		2636	97.8	101	1.6	60	1.1	27	0.6
	v4	141	1	7.1		2636	96.0	69	1.4	37	0.8	29	0.6
	v5	141	1	5.1		2568	94.7	188	3.2	123	2.3	38	0.7
szymanski	v1	73	1	20.5	2	–	>1800	12473	171.3	2	0.3	2	0.3
	v2	74	2	27.4		–	>1800	13434	197.6	150	2.2	67	1.4
	v3	73	1	20.5		–	>1800	10180	136.7	73	1.3	61	1.1
	v4	73	1	26.7		–	>1800	14365	207.7	591	8.9	79	2.2
	v5	73	1	20.0		–	>1800	–	>1800	73	1.3	61	1.2
bluetooth	v1	128	2	25.8	2	–	>1800	2112	31.7	1739	25.1	287	8.7
	v2	130	2	22.1		–	>1800	2292	34.6	1133	16.4	223	6.4
	v3	130	1	22.3		–	>1800	2324	35.3	1154	16.5	276	5.3
	v4	131	5	38.2		–	>1800	2617	40.6	2617	41.5	532	13.8
	v5	133	3	39.1		–	>1800	2437	38.5	2437	36.1	417	11.9
circularbuf	v1	115	1	26.9	2	52	0.9	52	0.9	52	0.9	32	0.8
	v2	116	2	15.5		1077	19.7	277	4.1	277	4.1	68	3.3
	v3	116	1	6.9		3794	171.8	770	14.3	126	1.9	21	0.5
	v4	118	2	15.3		3794	173.1	2916	105.6	462	7.4	46	1.5
	v5	117	1	28.2		–	>1800	924	17.8	924	18.1	102	3.3
nbds-list	v1	1168	5	9.2	2	–	>1800	1724	433.9	501	223.3	422	136.1
	v2	1624	3	1.9		–	>1800	898	117.3	10	141.6	10	141.6
	v3	1626	4	5.2		–	>1800	4660	701.6	503	102.9	503	103.2
	v4	1887	5	3.5		–	>1800	6007	698.9	35	90.7	14	80.4
	v5	1885	3	5.0		–	>1800	1304	160.7	198	73.2	175	53.1
nbds-skiplist	v1	1734	2	10.3	2	–	>1800	–	>1800	1874	263.6	1266	202.7
	v2	2095	2	3.0		–	>1800	4645	228.0	284	61.6	180	56.5
	v3	2095	2	3.2		–	>1800	–	>1800	299	61.9	223	59.9
	v4	2100	3	0.4		–	>1800	7508	266.3	5	48.3	5	48.2
	v5	2101	1	2.5		–	>1800	–	>1800	550	65.6	417	56.3
nbds-hashtable	v1	2234	1	0.3	2	–	>1800	4818	218.6	9	170.1	9	169.5
	v2	2322	2	8.6		–	>1800	–	>1800	2686	650.8	2686	632.6
	v3	2375	2	7.3		–	>1800	–	>1800	1684	440.5	1453	416.1
	v4	2418	2	2.7		–	>1800	9474	730.8	612	258.8	431	190.3
	v5	2422	2	4.6		–	>1800	17556	1396.2	849	337.1	763	303.5
Total		**34926**					**70585.0**		**17149.7**		**3478.4**		**2816.7**

bolic execution as well as the total execution time. And in many cases, *Incremental* can finish the execution in seconds while the baseline algorithm does not stop after 30 minutes.

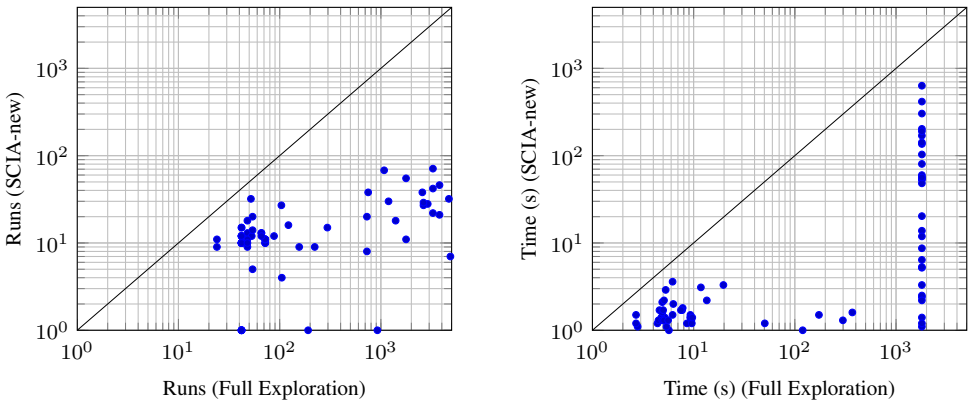


Figure 4.8: SCIA (new) versus standard symbolic execution.

Second, the two scatter plots in Figure 4.9 compares between *DPOR* and *Incremental*. The goal is to depict how much performance improvement can be achieved by *Incremental* over *DPOR* alone. Similarly, the dots below the diagonal lines are the winning cases of *Incremental*.

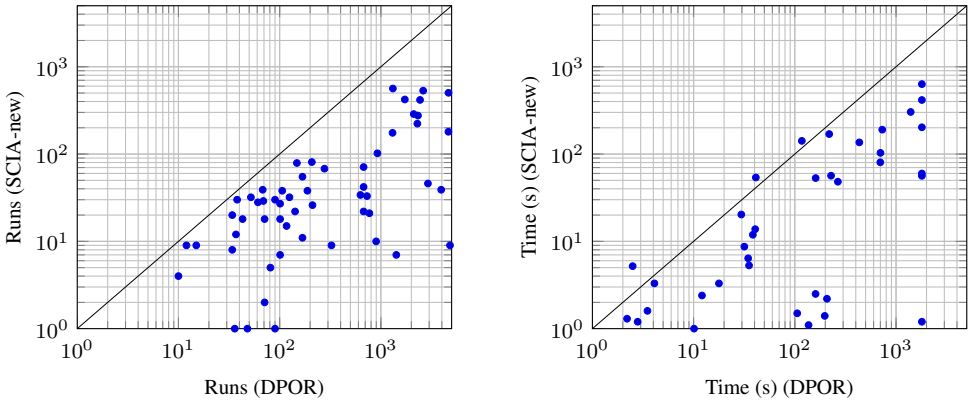


Figure 4.9: SCIA (new) versus DPOR-only symbolic execution.

Again, the *Incremental* method brings significant performance improvement compared to *DPOR*. However, there are also some dots slightly over the diagonal lines, despite that they have the same or a smaller number of runs. This situation comes from the additional computation overhead of the

static slicing, summary computation, as well as the summary-based pruning, that makes the total execution slower than *DPOR*. But, overall, the runtime using SCIA versus DPOR alone is 83% smaller.

It is worth pointing out that, while the *Incremental* method in *Conc-iSE* is able to bring a performance improvement, it could also bring a drastic reduction in the execution time compared to *DPOR* alone. For instance, in $nbds\text{-}list$ v2 (Table 4.3), *Conc-iSE* detects that the changes does not impact any shared variable access. Then the execution process can quickly skip exploring the whole state space and terminates after just one run.

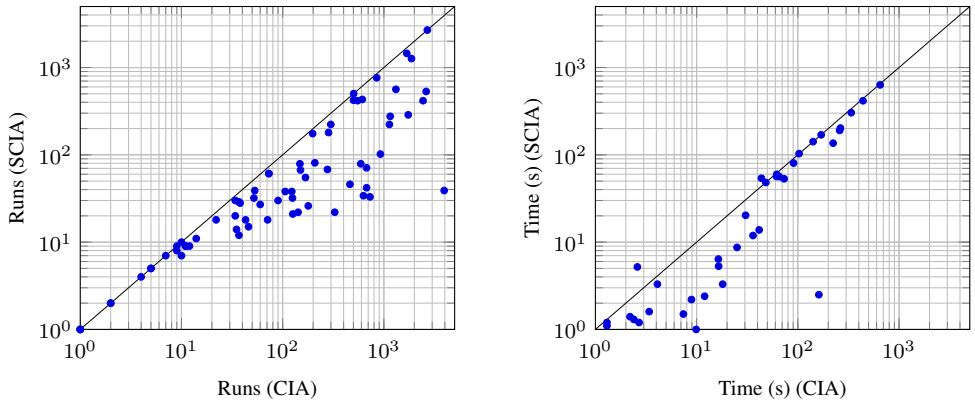


Figure 4.10: Comparing CIA versus SCIA.

Next, Figure 4.10 compares the two incremental approaches (*CIA* and *SCIA*). This comparison shows the effect of reusing the predicate summaries. Similar to previous cases, sometimes the summary based technique is not able to provide a significant reduction causing the runtime to be slightly higher; this usually occurs when the backward impact analysis causes many useful summaries to be removed. Nonetheless, on most cases the summary based technique is able to have a significant reduction in the number of runs leading to a reduction in runtime.

4.4.3 Threats to Validity

Fundamentally, an incremental analysis is only applicable when the code modification affects a subset of the entire program: if the entire program is modified then the incremental analysis degrades to the non-incremental one. *Conc-iSE* is more suitable in a software development environment where numerous small changes are made rather than monolithic changes. So, the manual modifications made during the experiments affected a subset of the entire program. The modifications from *nbds* are developer made modifications. Overall, the modifications affected around 0.3% to 10.3% of the program. It remains to be shown if this reflects the majority of software development scenarios.

Since *Conc-iSE* is expected to run frequently on small modifications there may be a large time burden on developers. The problem of when to schedule tests is considered as an orthogonal issue (e.g., Herzig et al. [63]). The tests on sequential programs are limited to one small example which may not reflect typical sequential C programs. Additional replication over more sequential programs would be required to see if the summary based technique is a good fit. Similarly, further repetitions on more concurrent programs would be required to make stronger generalizations.

4.5 Conclusion

I have presented *Conc-iSE*, the first summary-based incremental symbolic execution algorithm for concurrent programs. The new change-impact analysis in *Conc-iSE* is inter-procedural and thread-safe, capable of more accurately identifying instructions affected from the code changes between two program versions. I have also showed how execution summaries computed from the previous program can be used to prune redundant runs during the symbolic execution of the new program.

I have implemented *Conc-iSE* and evaluated it on a large set of multithreaded C programs. The experimental results show that *Conc-iSE* can significantly reduce the runtime cost of symbolic execution compared to state-of-the-art techniques.

Chapter 5

Symbolic Execution of PLC Code

Programmable logic controllers (PLCs) are specialized computers for automating electro-mechanical processes in a wide variety of industrial applications, including factory assembly lines, transportation systems, and smart power grids. PLCs are often equipped with domain-specific operating systems and virtual machines for executing software code written in programming languages such as Structured Text (ST), Ladder Diagram (LAD), and Sequential Function Chart (SFC). Since PLC software control critical infrastructures (e.g., the SCADA systems), design defects or implementation bugs may lead to catastrophes. However, despite the already widespread use of PLCs, automated testing tools are still lacking. In this work, we fill the gap by developing the first *symbolic execution based* tool for automatically testing PLC software.

Although symbolic execution has been applied to many programming languages, prior to this work, it has never been applied to multitask PLC software code. One reason is that PLC software are written in specialized and somewhat archaic languages that differ from mainstream programming languages, thus lacking open-source development tools. Another reason is that PLC software are periodic programs that often do not terminate, and they involve multiple tasks running concurrently

with respect to each other. Tasks may own different levels of priorites, where tasks with higer priorities could preempt low-priority tasks. Thus, precise modeling of this *non-conventional* execution semantics is difficult.

In this work, I solve the problems by leveraging an open-source PLC compiler named *Matiec* [87] and the symbolic execution tool *Cloud9* [38].

5.1 Introduction

First, I leverage *Matiec* to translate each PLC task from the original language (e.g., ST) to C. The C code is functionally-equivalent in that each of its program paths has a corresponding path in the original PLC task, which ensures that tests generated from the C code can be mapped back to the PLC. Second, I synthesize a test harness (i.e., the *main()* function in C) to invoke PLC tasks as threads. Threads are further constrained to precisely model the priority-based preemptive scheduling as defined in the PLC program semantics. Finally, I extend *Cloud9* to symbolically execute the multithreaded C model. The new symbolic execution procedure systematically generates test cases to cover both paths of each periodic task and interleavings of these tasks.

Figure 5.1 shows the overall flow of the new method SYMPLC, where P denotes the PLC program, and the translation from P to C is implemented in the *Matiec* PLC compiler. The new symbolic execution procedure based on *Cloud9* produces test cases of the form (in, sch), where in denotes the input data and sch denotes the interleaving schedule. Since *Cloud9* only supports coarse-grained thread scheduling, I extended it to execute multithreaded C code at a finer granularity. Furthermore, I develop several PLC-specific reduction techniques that leverage the periods and priorities of tasks as well as visited states to efficiently pruning redundant interleavings. Since these redundant interleavings are due to PLC-specific program semantics, they cannot be removed

by state-of-the-art partial order reduction (POR) techniques [51].

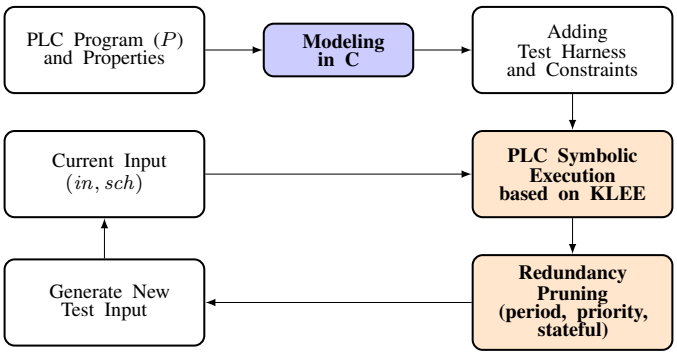


Figure 5.1: The overall flow of SYMPLC.

One advantage of SYMPLC as a software tool is the flexibility resulted from its separation of the *modeling* and *analysis* phases. In the modeling phase, it focuses on capturing the precise semantics of PLC programs written in various languages by constructing the functionally-equivalent C model. Each PLC language may be handled by a dedicated front-end; multiple front-ends may be developed without interfering each other. Eventually, PLC tasks, regardless of which languages they were written in, are merged into the same C model that simulates the preemptive scheduling using threads. In the analysis phase, it focuses on executing the multithreaded C model as efficiently as possible, without worrying about complications of the PLC languages. Thus, the overall architecture allows SYMPLC to easily support new languages and new PLC execution platforms.

Another advantage of SYMPLC is the efficiency resulted from the PLC-specific interleaving reduction techniques. Since these new techniques are designed specifically for the PLC task scheduling, they are more effective than generic partial order reduction (POR) techniques. The experiments show POR is often ineffective for removing redundant executions in PLC programs due to their semantic differences from thread interleavings. For example, in standard multithreaded programs, two threads with the same priority level are allowed to preempt each other, whereas in PLC programs, they are not allowed to preempt each other. Furthermore, PLC tasks are executed periodi-

cally, which means they never terminate. The new reduction techniques in SYMPLC are designed to take advantage of these unique characteristics.

SYMPLC is a test input generation tool. As such, it differs from existing tools for simulating, verifying, or synthesizing PLC software. Specifically, simulators [25, 77, 97] can execute PLC code in controlled environments, but they require the users to handcraft test inputs. In contrast, SYMPLC automatically generates these inputs. Verification tools [47, 80, 96] are designed to formally prove the correctness of properties in *models* of PLC software, but these formal models are at a much higher level of abstraction than the actual software code. In contrast, SYMPLC directly executes the actual PLC code. Synthesis tools [33, 34] have the ambitious goal of generating PLC code directly from formal specifications, thus bypassing the programmers completely. However, these tools only synthesize small programs with single tasks due to scalability problems. In comparison, SYMPLC is more scalable and can uniformly handle both single- and multi-task PLC programs.

In conclusion, I make the following contributions:

- I propose the first symbolic execution based method for automatically testing PLC software written in languages of the IEC 61131-3 standard, by first translating the original PLC tasks to C code and then applying symbolic execution to generate the test inputs.
- I propose a number of PLC-specific reduction techniques for eliminating redundant interleavings, which are more effective in reducing the interleaving search space than state-of-the-art POR techniques.
- I implement the new methods and experimentally evaluate them on a set of PLC programs to show SYMPLC's efficiency and effectiveness in detecting property violations.

5.2 Motivation

In this section, I use examples to illustrate bugs in PLC programs and explain why SYMPLC is necessary to detect them.

5.2.1 Single-task PLC Programs

Figure 5.2 shows three PLC programs that implement a two-player game named *Responder* [31], where $I0.0$, $I0.1$ and $I0.2$ are inputs from the game host and two players, while $Q0.0$ and $Q0.1$ are outputs for the players. The program consists of two sections: `CONFIGURATION` and `PROGRAM`. The `CONFIGURATION` section declares global variables and allocates resource (CPU) to a task. For example, Task `T1` is started every 10 milliseconds and each time it executes an instance named `Game` of the program `ProgA`. The actual code of `ProgA`, provided in the `PROGRAM` section, has two statements. The first statement at Line 12 reads from $I0.0$, $I0.1$, $Q0.0$, and $Q0.1$ and then computes the new value for $Q0.0$, while the second statement computes the new value for $Q0.1$.

Initially, all inputs, outputs, and global variables are set to *false*. The host starts the game by setting $I0.0$ to *true*. Then, the players try to respond as quickly as possible by setting their inputs to *true*. If the first player is faster, its output $Q0.0$ becomes *true*, indicating she has won. But if the first player is slower, the second player's output $Q0.1$ becomes *true*. After a player's output becomes *true*, it should remain true until the host sets $I0.0$ back to *false*.

The program in Figure 5.2 (a) is buggy because, when both players respond at the same time, the program is not able to set both outputs to *true* (indicating a tie). Instead, it is biased toward the first player – since the PLC program is executed sequentially, i.e., one line after another, $Q0.0$ will be set to *true* first, which prevents $Q0.1$ from being set to *true* subsequently.

```
1   CONFIGURATION PLC_Cell1
2     VAR_GLOBAL
3       I0.0: BOOL; I0.1: BOOL; I0.2: BOOL;
4       Q0.0: BOOL; Q0.1: BOOL;
5     END_VAR
6     RESOURCE CPU_Responder ON CPU001
7       TASK T1 (INTERVAL := t#10ms, PRIORITY := 1);
8       PROGRAM Game WITH T1 : ProgA;
9     END_RESOURCE
10  END_CONFIGURATION
11  PROGRAM ProgA
12    Q0.0 := (I0.1 OR Q0.0) AND (NOT Q0.1) AND I0.0 ;
13    Q0.1 := (I0.2 OR Q0.1) AND (NOT Q0.0) AND I0.0 ;
14  END_PROGRAM
15
```

(a) The initial (buggy) implementation

```
1   VAR_GLOBAL
2   ...; M0.0: BOOL; M0.1: BOOL;
3   END_VAR
4   ...
5   PROGRAM ProgA
6     M0.0 := (I0.1 OR Q0.0) AND (NOT Q0.1) AND I0.0 ;
7     M0.1 := (I0.2 OR Q0.1) AND (NOT Q0.0) AND I0.0 ;
8     Q0.0 := M0.0;
9     Q0.1 := M0.1;
10  END_PROGRAM
11
```

(b) Revised but still buggy implementation

```
1   PROGRAM ProgA
2     M0.0 := (I0.1  AND (NOT Q0.1) OR Q0.0) AND I0.0 ;
3     M0.1 := (I0.2  AND (NOT Q0.0) OR Q0.1) AND I0.0 ;
4     Q0.0 := M0.0;
5     Q0.1 := M0.1;
6   END_PROGRAM
7
```

(c) The correct implementation

Figure 5.2: Three implementations of the PLC *Responder* in ST.

To fix this bug, developers could introduce two auxiliary global variables $M0.0$ and $M0.1$ as shown in Figure 5.2 (b), to buffer the temporary outputs before assigning them to $Q0.0$ and $Q0.1$, respectively. Thus, setting $M0.0$ to *true* does not prevent $M0.1$ from becoming *true*. Indeed, when the two players respond at the same time, both outputs will be set to *true*.

Unfortunately, the revised program is still faulty. Assume that both outputs have been set to *true* at the end of the first task execution because two players responded concurrently. Since task `T1` executes periodically, during the next task execution, $Q0.1$ being *true* will force $Q0.0$ to become *false*, and $Q0.0$ being *true* will force $Q0.1$ to become *false*. Thus, both outputs become *false* at the

end of the second execution, which is not expected. Recall that the expected behavior is that both outputs remain true, until the host ends the game.

To fix the second bug, developers may have to revise the code to what is shown in Figure 5.2 (c). Compared with the program in Figure 5.2 (b), the modification is actually minor – by simply enlarging the scope of the two logical-OR operators to include $Q0.0$ and $Q0.1$. Because of this modification, after $Q0.0$ and $Q0.1$ become *true*, they will remain *true* during all subsequent executions of `T1` regardless of the new input data, until the host ends the game by setting $I0.0$ to *false*.

These three examples show that even a simple PLC program with a single task may have subtle bugs in its implementation due to the non-conventional program semantics. Thus, automated testing tools such as SYMPLC would be invaluable.

5.2.2 Multi-task PLC Programs

Figure 5.3 shows a two-task PLC program which implements a simplified version of the robotic controller from [26]. The `RESOURCE` section contains the two tasks, both of which are assigned to the device `CPU001`. Task `T1` has a shorter period (`100ms`) and a higher priority, while task `T2` has a longer period (`200ms`) and a lower priority. In PLCs, tasks with higher priorities may preempt low-priority tasks, but not vice versa. Assume tasks never miss their corresponding deadlines, then implicitly, the timing constraint is that `T1` finishes its execution within 100ms and `T2` finishes within 200ms. Furthermore, the two tasks are associated with programs `ProgA` and `ProgB` defined below.

The `PROGRAM` sections provide the source code of the tasks, which share two global variables. In addition, `ProgA` reads from the input variable `Sensor_input`, whereas `ProgB` does not read

```
1   CONFIGURATION PLC_Cell2
2     VAR_GLOBAL
3       Obstacle : BOOL := 0; Forward : INT := 50;
4     END_VAR
5     RESOURCE CPU_main ON CPU001
6       TASK T1 (INTERVAL := t#100ms, PRIORITY := 1); //High
7       TASK T2 (INTERVAL := t#200ms, PRIORITY := 2); //Low
8       PROGRAM Fast WITH T1 : ProgA;
9       PROGRAM Slow WITH T2 : ProgB;
10    END_RESOURCE
11  END_CONFIGURATION
12
13  PROGRAM ProgA
14    VAR_INPUT
15      Sensor_input : INT;
16    END_VAR
17    Obstacle := 0;
18    IF (Sensor_input <= 10) THEN
19      Obstacle := 1;
20      Forward := -100;
21    END_IF;
22  END_PROGRAM
23
24  PROGRAM ProgB
25    IF (Obstacle = 0) THEN
26      Forward = 100;
27    END_IF;
28  END_PROGRAM
29
```

Figure 5.3: A Multi-task PLC Program in Structured Text.

from any primary input.

`ProgA` is responsible for obstacle detection, e.g., by setting `Forward` to the reverse speed -100 when the value of the input `Sensor_input` indicates an obstacle ahead. `ProgB` computes the forward speed of the robot if no obstacle is detected. Thus, both tasks may write to the variable `Forward` (Lines 20 and 26). The race condition would cause a problem in the following scenario:

- `T1` runs first and `Sensor_input` is greater than 10;
- `T1` finishes its first execution of `ProgA`;
- `T2` starts and proceeds to the statement at Line 26, then it is preempted by `T1` before writing to `Forward`;
- `T1` detects an obstacle and sets `Forward` to -100, and finishes its second execution of

`ProgA`;

- `T2` continues the execution of `ProgB`.

At this moment, the value of `Forward` is -100, and should have remained -100, but `ProgB` overwrites it to 100 as illustrated by Figure 5.4. The erroneous value is not expected, and may result in the robot hitting the obstacle.

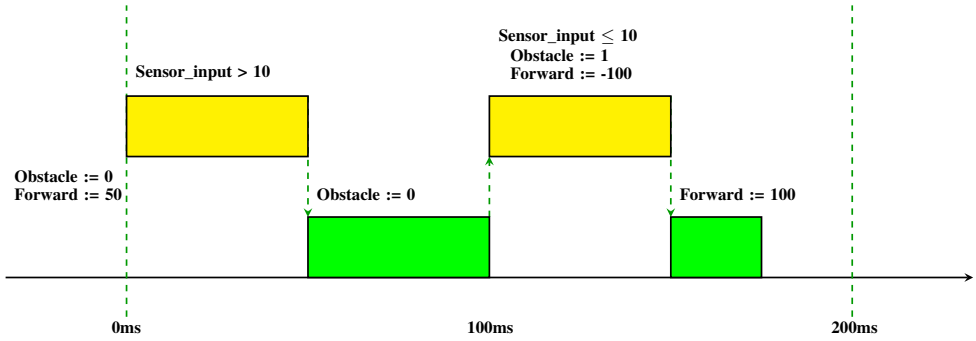


Figure 5.4: The task interleaving that fails the assertion.

Note that detecting the kind of bug shown in Figure 5.4 is not easy, since it requires a combination of the right input data (`Sensor_input` being > 10 in the first execution of `ProgA` and ≤ 10 in the second execution of `ProgA`) and task interleaving (`ProgB` is preempted by the second execution of `ProgA` right before the write to `Forward`). Although in practice, simulators may be used to reproduce this bug after it is detected, the users are required to handcraft the error-triggering input data in the first place, which is difficult. Furthermore, simulators do not have the capability of systematically exploring the space of task interleavings. In contrast, SYMPLC solves the problem by automatically exploring the combined input and interleaving space. Thus, given the source code of this PLC program, SYMPLC will generate not only the failure-triggering test data but also the corresponding task schedule.

5.3 Modeling PLC Program Semantics

SYMPLC first translates PLC tasks to equivalent C code, and then models their execution semantics using threads.

5.3.1 Translating PLC Tasks to C

Variables. PLC programs have different variable types. For example, the keyword VAR_INPUT defines read-only input variables, VAR_OUTPUT defines output-only variables, and the type VAR_EXTERNAL defines the global variables. There are eight such usage types in IEC 61131-3 standard, all of which are mapped by SYMPLC to proper variables in the C program. The translation is mostly straightforward except for inputs, which require special handling.

Inputs. Variables such as `sensor_i1` and `sensor_i2` at Line 15 in Figure 5.5 are primary inputs. They need to be fed a symbolic value every time the corresponding task is activated. This is accomplished by calling the API function `symplc_mk_symbolic`, which returns a symbolic value for the variable. SYMPLC applies value-range constraints over these symbolic values to ensure that they always concretize to values allowed by their types. The use of symbolic values simulates the fact that input data may be arbitrary.

Timers. The behavior of PLC timers is abstracted by treating the output of each timer invocation as a symbolic variable: it is either *true* or *false* since both values are possible at run time. It ensures that actions depending on different timer outputs are always covered. Although this modeling approach may introduce potentially redundant test cases, it has the advantage of not missing any valid test input. Furthermore, the redundant test cases may be eliminated by the new PLC-specific reduction techniques implemented inside SYMPLC.

```
1   bool Obstacle = 0; int  Forward = 50;
2   void ProgA (int Sensor_input){
3     Obstacle = 0;
4     if (Sensor_input <= 10){
5       Obstacle = 1;
6       Forward = -100;
7     }
8   }
9   void ProgB (){
10    if (!Obstacle){
11      Forward = 100;
12    }
13  }
14  void thread1 () {
15    int sensor_i1, sensor_i2;
16    symplc_mk_symbolic(&sensor_i1, ...);
17    symplc_mk_symbolic(&sensor_i2, ...);
18    ProgA(sensor_i1);
19    //symplc_task_boundary();
20    ProgA(sensor_i2);
21  }
22  void thread2 () {
23    ProgB();
24  }
25  int main( void ){
26    pthread_t t1, t2;
27    for (i=0; i<MAX_ITER; i++) {
28      //symplc_hyperperiod_begin();
29      pthread_create(&t1, 0, thread1, 0);
30      pthread_create(&t2, 0, thread2, 0);
31      //symplc_set_priority_n_period(t1, 1, 100);
32      //symplc_set_priority_n_period(t2, 2, 200);
33      pthread_join(&t1);
34      pthread_join(&t2);
35      //symplc_hyperperiod_end();
36      assert(Obstacle == (Forward == -100));  // property
37    }
38  }
39
```

Figure 5.5: The Multithreaded C Model of the ST Program.

Statements. The translation of PLC program statements from the ST language to C is straightforward because as a programming language, C is strictly more expressive than ST. Thus, any ST statement in the original program can be expressed by a corresponding C statement. Furthermore, since the number of built-in functions in ST (library functions) is fairly small, each of these functions may be replaced by a corresponding C function. In our implementation, the translation from ST code to C code is carried out by the *Matiec* PLC compiler, which has been designed to conforme to the popular IEC 61131-3 standard. In Figure 5.5, for example, the program statements of the PLC robotic controller are translated into the C code at Lines 1-13.

5.3.2 Constructing the Test Harness

The test harness is the *main()* function that treats PLC tasks as threads and incorporates them to a complete C program. In Figure 5.5, for example, the test harness consists of Lines 14-38. There are two separate issues in simulating PLC tasks using threads. The first one is constructing a thread for potentially multiple invocations of each task (Lines 14-24). The second one is using these threads to simulate the periodic execution of PLC tasks (Lines 25-38).

It is always feasible to simulate PLC task interleaving semantics using threads because threads have strictly more *permissive* interleaving semantics. That is, all possible interleavings allowed by PLC tasks are included in the set of interleavings allowed by threads. However, threads may allow certain interleavings that are not possible in PLCs. Thus, SYMPLC has to constrain the threads in its C model to make the modeling of PLC tasks precise. Toward this end, the first step is to construct all threads for a hyper-period.

Hyper-period. PLC tasks in the same program may have different periods. For instance, in the running example, `T1` has a period of 100ms and `T2` has a period of 200ms. In this context, a hyper-period is the least common multiplier of the periods of all tasks. Thus, the hyper-period of the running example is 200ms. Clearly, within a hyper-period, `T1` will be executed twice and `T2` will be executed once. The reason why using the hyper-period is because timing-related program behaviors repeat themselves after each hyper-period. Thus, focusing on analyzing the tasks within each hyper-period is important. Furthermore, the hyper-period will be used to reduce the symbolic execution cost. In the C model, I construct one thread for all the execution instances of each task in a hyper-period. That is why in Figure 5.5, `thread1()` invokes `ProgA` twice, but `thread2()` invokes `ProgB` only once.

Periodic execution. Next, SYMPLC construct a for-loop in the *main()* function to execute all

threads concurrently. Each iteration of the for-loop corresponds to a hyper-period. The total number of iterations is bounded by a user-defined parameter MAX_ITER, since PLC programs in general are non-terminating programs. Within each hyper-period, SYMPLC first creates the threads and then set their parameters (period and priority). These parameters will be passed to the symbolic execution engine to avoid exploring interleavings that are not allowed by the PLC program semantics. As shown in Figure 5.5, SYMPLC uses special API functions to signal the boundary of the hyper-period and boundaries of tasks within each thread.

Assertions. The assertion at the end of the hyper-period represents the property to be checked. In PLC programs, developers may use the ASSERTION(...) keyword to specify a property. Such assertions are translated into assertions in the C program straightforwardly. SYMPLC also allows users to specify additional assertions, which are inserted at the end of the hyper-period (e.g., at Line 36 in Figure 5.5). Assertions are reachability properties because each *assert(c)* may be modeled as *if(!c) ERROR*, where *ERROR* is an error location. During symbolic execution, if any error location is reached, the symbolic execution tool produces an error-triggering test case.

5.4 Symbolic Execution Phase

5.4.1 Multithreaded C Model for PLC

The multithreaded C model of a PLC program consists of a set of periodic tasks $\mathsf{T} = \{T_1, \ldots, T_n\}$. Each task $T_i \in \mathsf{T}$, where $1 \leq i \leq n$, denotes an instance of a PLC program within a hyper-period. Consider the program named `ProgA` in Figure 5.5, which has two instances in a hyper-period (Lines 18 and 20). In our C model, these two instances are considered as different tasks in T.

Tasks share a set GV of global variables. Each T_i also has a set LV_i of local variables. In addition,

each T_i may read from a set PI of primary inputs. Thus, T_i can be viewed as a sequential program that reads from primary inputs as well as global variables, updates the global variables, and computes the outputs. Since tasks are executed periodically, in addition to being a sequential program, each T_i has the following attributes:

- T_i.*tid* denotes the unique identifier of the task;
- T_i.*priority* denotes the priority level of the task;
- T_i.*period* denotes the execution period of the task within a hyper-period;
- T_i.*startT* denotes the starting time of task T_i's current period;
- T_i.*endT* denotes the ending time of the task T_i's current period.

Due to PLC's non-conventional interleaving semantics, for any two tasks T_i and T_j, where $i \neq j$,

- if $T_i.priority < T_j.priority$, then T_j may preempt the execution of T_i at any time between $T_i.startT$ and $T_i.endT$, but T_i cannot preempt T_j;
- if $T_i.priority = T_j.priority$, neither task may preempt the other task.

This is different from the standard interleaving semantics of a multithreaded program, where threads with the same priority are allowed to preempt each other.

The execution of task T_i leads to a sequence of events $t_1, \ldots, t_k$. For ease of presentation, we assume each event $t \in T_i$ inherits all attributes of the task T_i including *tid*, *priority*, *period*, *startT*, and *endT*. In other words, $t.startT$ and $t.endT$ are the expected start time and end time of the period of the task T_i. In addition, SYMPLC introduces $t.task$ to denote the task T_i that generates the event t.

Some events in a PLC program are reads and writes of global variables, while others are computations over local variables. Local operations are further divided into branching statements e.g., $if(c)$, and assignments $lv = exp$, where exp may be arithmetic computations, bit-string opera-

tions, boolean operations, etc. Without loss of generality, we assume $if(c)$ involves only local variables, because $if(exp(gv))$, where $gv \in \mathsf{GV}$, can always be replaced by $lv = gv; if(exp(lv))$, where $lv \in \mathsf{LV}_i$ is a newly added local variable and $if(exp(lv))$ involves only local variables. Thus, during symbolic execution, SYMPLC only need to consider two types of events:

- *interleaving schedule* events, which perform context switches right before global reads and writes;
- *sequential computation* events, which are either $if(c)$ or assignments over local variables.

Only *interleaving schedule* events may affect the execution order of different tasks. Thus, we will focus on analyzing them to identify redundant interleavings. In contrast, *sequential computation* events are handled in the same way as in standard symbolic execution tools.

5.4.2 Overall Algorithm

Algorithm 5 shows the overall execution procedure of a multi-task PLC program, which builds upon the baseline symblic execution Algorithm 1. For bravity, I omit the same part of the recursive procedures and highlight the main differences.

Specifically, if s is an interleaving schedule node (right before a global read or write), SYMPLC is invoked recursively to explore each possible schedule together with the subsequent events (Lines 7-11). If s is a sequential computation node (local statement within a task), SYMPLC is invoked recursively to explore each branch and assignment (Lines 5-6 and Lines 12-13, respectively). Upon reaching the end of an execution (Lines 14-15), SYMPLC generates the corresponding test case and backtracks from the current state.

Subroutine $NextSymbolicState$ takes the current state s and the event t as input, and returns the

Algorithm 5: Symbolic execution of a multi-task PLC program.

```
1  Initially: State stack S = ∅; Start Explore(s_0) with the symbolic state s_0.
2  Explore(State s)
3  begin
4      S.push(s);
5      if s is a b-PP node then
6          ...
7      else if s is an i-PP node then
8          while ∃t ∈ s.enabled\s.done and ¬IsRedundant(s, t) do
9              Explore(NextSymbolicState(s, t));   // γ event (enhanced)
10             s.done ← s.done ∪ {t};
11         end
12     else if s is other sequential computation node then
13         ...
14     else
15         ...
16     end
17     S.pop();
18 end
19 NextSymbolicState(State s, Event t)
20 begin
21     ...
22 end
23 IsRedundant(State s, Event t )
24 begin
25     if t is redundant regarding the POR theory then
26         return true;
27     end
28     return false;
29 end
```

newly computed symbolic state s' as output. Here we omit the details of this process since it remains the same as in standard symbolic execution procedures in the literature.

The challenge is mitigating the combinatorial blowup associated with the interleaving schedule events (Lines 8-11) because in the worst situation, the number of different interleaved executions can be exponential in the number of global operations. Traditional techniques for mitigating the interleaving explosion problem are based on *partial order reduction (POR)* [51], the main idea of which is to group interleavings into a set of equivalence classes, and then pick a representative interleaving of each class while skipping the other (redundant) interleavings. In Algorithm 5, this is implemented inside Subroutine $IsRedundant$. However, POR does not consider the addi-

tional interleaving constraints imposed by PLC tasks. As such, it is not effective in mitigating the interleaving explosion problem in PLC programs.

5.5 PLC-specific Reductions

This section presents three new reduction techniques designed to take advantage of the unique characteristics of PLC programs. Specifically, they are related to leveraging information from (1) the priorities of tasks, (2) periods of tasks, and (3) previously visited program states during symbolic execution.

Algorithm 6 shows the details of the first two reductions. The third reduction will be presented in Section 5.5.3. Here the subroutine $IsRedundant$ returns *true* if executing t from the state s is redundant, either it is due to DPOR or infeasibility regarding the PLC interleaving semantics. Within the current hyper-period, t' is the last event chosen before reaching s (Line 7). The subsequent two sections illustrate how these two types of reductions make use of t' in more details.

5.5.1 Priority-based Reduction

In this new reduction, SYMPLC imposes three rules following the way PLCs schedule their tasks:

1. The active task with the highest priority must be scheduled to run before all other active tasks whenever a hyper-period starts.
2. A running task can only be preempted by another running task with a strictly higher priority;
3. If a high-priority task starts before the period beginning of a low-priority task, there must be no interleavings between these two tasks.

Algorithm 6: Deciding if event t chosen at s is redundant.

```
1  IsRedundant(State s, Event t )
2  begin
3      if t is redundant regarding the POR theory then
4          return true;
5      end
6      // Priority-based reduction.
7      let t' be the last event in S before reaching t;
8      if t' is NULL then
9          if t.priority is not the highest in s.enabled then
10             return true;
11         end
12     else
13         if t is about to preempt t' then
14             if t'.priority ≥ t.priority or t'.startT ≥ t.startT then
15                 return true;
16             end
17         end
18         // Period-based reduction.
19         if t'.tid ≠ t.tid then
20             if t'.endT < t.startT or t'.startT ≥ t.endT then
21                 return true;
22             end
23         end
24         if t is the last event in t.task then
25             if ∃t_h, t_l ∈ S that t_h.tid==t.tid and t_h preempted t_l then
26                 if ∃t'' ∈ S that t''.task interleaved with t.task and t''.startT ≥ t_l.endT then
27                     return true;
28                 end
29             end
30         end
31     end
32     return false;
33 end
```

Algorithm 6 formalizes these rules in Lines 8-17. First, when both high-priority task and low-priority task are enabled and ready to run, the high-priority task should always run first. This corresponds to the conditions at Lines 8-11: if t' does not exist, it means t is the first event in the current hyper-period. At this moment, the PLC must choose the highest-priority task to execute. Thus, if t is not the highest-priority task, $IsRedundant$ returns *true*.

On the other hand, if t' exists and t is about to preempt t', we first leverage the task priorities to perform a reduction, and then leverage both the priorities and the periods to perform another

reduction. Specifically, SYMPLC checks two the following conditions at Line 14.

The first condition at Line 14 ensures that t has a strictly higher priority than t', because PLCs only allow high-priority tasks to preempt low-priority tasks. And tasks having the same level of priority are not allowed to preempt each other. The second condition makes use of periods of the tasks. Note that at this point, t's priority is higher than that of t'. The condition checks if the (expected) start time of the period of t is before the (expected) start time of the period of t'. If this is the case, the interleaving is infeasible because the low-priority event t' should not have occurred before t (it should only be executed after the end of t's period).

Consider the PLC program in Figure 5.3 again as an example, but with an important modification—setting the `INTERVAL` of `T1` to `t#200ms` instead of `t#100ms`. Since both tasks now need `t#200ms`, the hyper-period becomes `200ms`, meaning `ProgA` and `ProgB` are invoked once each in the new threads `thread1` and `thread2`, respectively. The control flow of these two new threads are shown in Figure 5.6, where nodes are the global reads or writes and solid lines are the control flows. Recall that the primary input `Sensor_input` is modeled as a *symbolic* variable, thus allowing both branches immediately after the node `1` to be taken. In contrast, the branches immediately after the node `4` depend only on the value of the global variable `Obstacle`.

If it were a standard multithreaded program, each thread would be allowed to preempt the other one at the control flow nodes, thus leading to a total of 12 interleavings, as shown in the second and fifth columns of Table 5.1, labeled *All-Interleavings*. Among them, the two interleavings marked with $\star$ would violate the assertion. After applying the DPOR algorithm, for example, eight interleavings would remain while the other four would be removed. Specifically, `1-2-4-3` is removed because it is equivalent to `1-2-3-4`; `1-4-5-2-3` is equivalent to `1-4-2-5-3`; `4-1-5` is equivalent to `4-5-1`; and `4-1-5-2-3` is equivalent to `4-5-1-2-3`.

However, applying SYMPLC's new priority-based reduction would lead to significantly fewer in-

```
1     RESOURCE CPU_main ON CPU001
2       TASK T1 (INTERVAL := t#200ms, PRIORITY := 1); //High
3       TASK T2 (INTERVAL := t#200ms, PRIORITY := 2); //Low
4       PROGRAM Fast WITH T1 : ProgA;
5       PROGRAM Slow WITH T2 : ProgB;
6     END_RESOURCE
7
```

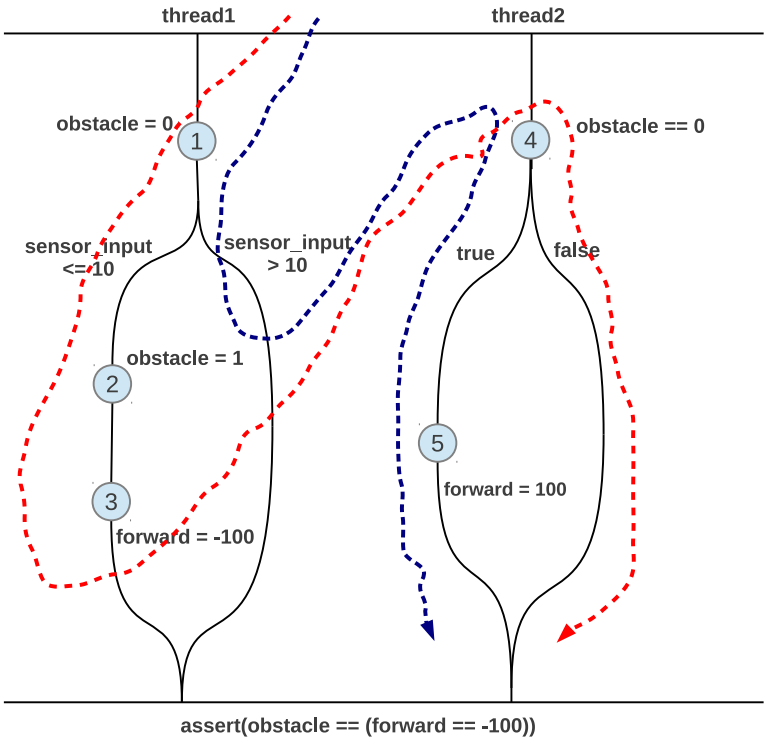


Figure 5.6: The control flow graph of the modified program.

terleavings. In fact, only two interleavings would remain, which are shown by the red and blue dotted lines in Figure 5.6. This is because, according to our second rule, all six interleavings in Column 2 except `1-4-5` and `1-2-3-4` are infeasible, because the low-priority task (`T2`) preempts the high-priority task. Similarly, according to our first rule, all six interleavings in Column 6 are infeasible, because when both `T1` and `T2` are active and ready to run at the beginning, the PLCs would always choose to execute the high-priority task (`T1`).

Since the erroneous interleavings (4 and 10) are not explored by SYMPLC, and SYMPLC termi-

Table 5.1: Explored interleavings with priority-based reduction.

ID	All-Interleavings	DPOR	SymPLC	ID	All-Interleavings	DPOR	SymPLC
1	`1-4-5`	✓	✓	7	`4-5-1`	✓	
2	`1-2-3-4`	✓	✓	8	`4-1-5`		
3	`1-2-4-3`			9	`4-1-2-5-3`	✓	
4 ⋆	`1-4-2-3-5`	✓		10 ⋆	`4-1-2-3-5`	✓	
5	`1-4-2-5-3`	✓		11	`4-1-5-2-3`		
6	`1-4-5-2-3`	✓		12	`4-5-1-2-3`	✓	

nates after two hyper-periods (due to the termination condition to be presented in Section 5.5.3), Then the validity of this assertion condition proves.

SYMPLC's implementation uses an *on-the-fly* computation to decide whether the current interleaving is feasible. Take the second rule as an example. Whenever an instruction accessing global variables is interpreted in the symbolic execution engine, SYMPLC checks the priority of its task against the operation history of current execution. If a preceding operation is from an active task whose priority is higher than the current one, then the interleaving resulted from executing t at s should be skipped. The first and the third rule are developed in a similar fashion.

In Figure 5.6, for instance, interleaving `4-5-1` is determined to be infeasible immediately after the first node `4` is reached by SYMPLC, since the first rule is violated. Therefore, SYMPLC backtracks from node `4` while skipping the interleavings numbered 8-12 entirely.

5.5.2 Period-based Reduction

SYMPLC develops two rules over task interleaving in this new reduction:

1. Two tasks are allowed to interleave when their expected execution periods overlap in time;

2. If a high-priority task T_h preempts a low-priority task T_l, T_h must not interleave with any task whose period begin time is not earlier than the period end time of T_l.

Algorithm 6 formalizes these rules at Lines 19-31. Recall that $t.startT$ and $t.endT$ are the expected logical time when the period of t begins and ends (we are not concerned with the actual start time and end time of t, except that they must fall within the period). Without these rules, any two operations from different threads would have been allowed to execute concurrently in the same hyper-period. However, since each task must meet its own deadline, some of them can never run concurrently.

```
1   CONFIGURATION PLC_Cell1
2     RESOURCE CPU_main ON CPU001
3       TASK T1 (INTERVAL := t#100ms, PRIORITY := 1); //H-priority
4       TASK T2 (INTERVAL := t#200ms, PRIORITY := 2); //M-priority
5       TASK T2 (INTERVAL := t#300ms, PRIORITY := 3); //L-priority
6       PROGRAM Fast WITH T1 : ProgA;
7       PROGRAM Const WITH T2 : ProgB;
8       PROGRAM Slow WITH T2 : ProgC;
9     END_RESOURCE
10  END_CONFIGURATION
```

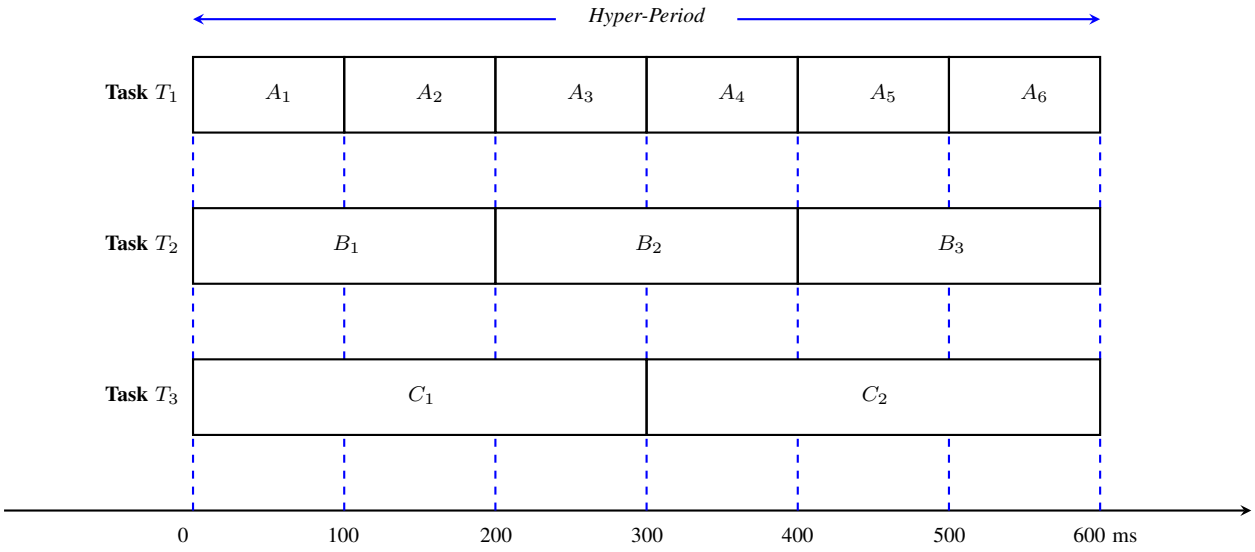


Figure 5.7: Three periodic tasks with a hyper-period of 600ms.

Consider the program in Figure 5.7 as an example, which has three tasks `T1`, `T2` and `T3` with periods 100ms, 200ms and 300ms, respectively. Thus, the hyper-period is 600ms, allowing `T1` to execute six times, `T2` to execute three times, and `T3` to execute twice. For ease of presentation, let the six instances of `T1` be denoted from A_1 to A_6, the three instances of `T2` be denoted from B_1 to

B_3, and the two instances of `T3` be denoted C_1 and C_2.

Without the timing-related information, symbolic execution would have to explore all possible interleavings of these tasks, including the obviously infeasible ones between A_1 and B_2, for example, which do not overlap in time. These infeasible interleavings will be removed by applying our reduction rules.

SYMPLC first compare the task IDs of t' and t in Algorithm 6 – different IDs means they belong to different tasks. The next rule at Line 19 is straightforward, since interleavings cannot occur if the two tasks do not overlap in time. In the running example, the period of A_1 is [0ms, 100ms] while the period of B_2 is [200ms, 400ms]. Obviously, events in A_1 do not occur concurrently with events in B_2. Similarly, the periods of B_3 and C_1 do not overlap. Both of these two cases are handled by the conditions at Lines 20 of Algorithm 6.

The second rule (Lines 24-30) is more subtle because the infeasible interleavings are deduced via a preceding interleaving, based on both periods and priorities of involved tasks. As shown in Figure 5.7, the period B_2 is expected to start before A_4. Thus, it appears that A_4 may interleave with B_2. However, if B_2 preempts C_1 in a particular execution, then B_2 must end before the end of the period of C_1, to allow C_1 to meet its deadline. Since B_2 would have ended before the start of the period of A_4, it cannot run concurrently with A_4. Thus, in this particular example, A_4 and B_2 can no longer interleave.

This example also illustrates the third priority-based reduction rule (Line 14) proposed in Section 5.5.1: A_3 starts from the 200ms, while the earliest time B_2 can start is 200ms. Since `T1` has a higher priority, and A_3 starts earlier than B_2, the execution of A_3 cannot be interrupted by B_2. Thus, any interleaving between them is guaranteed to be infeasible.

5.5.3 Stateful Exploration

Algorithm 7: Next state computation with stateful reduction.

```
1  NextSymbolicState(State s, Event t)
2  begin
3      ...
4      if t is halt then
5          s' ← normal_end_state;
6      else if ... then
7          ...
8      else if t is plc_hyperperiod_end then
9          if s ⊆ visited then
10             s' ← normal_end_state;
11         else
12             s' ← s;
13         end
14     else
15         ...
16     end
17     return s';
18 end
```

Next is the state-based reduction. Recall PLC tasks are periodic and thus never terminate. Furthermore, symbolic execution by default is geared toward detecting bugs as opposed to proving the correctness of properties. Thus, applying SYMPLC with a user-specified depth bound in general will never prove the absence of bugs in a PLC program. However, information of already-visited states may be leveraged to detect early-termination conditions. This allows SYMPLC to drastically reduce the number of test cases, as well as prove the correctness of properties.

Algorithm 7 shows the modified subroutine $NextSymbolicState$ in Algorithm 5 that implements this method. At the end of each hyper-period, it checks if the new symbolic state s' has been visited previously. If the answer is yes, it returns the $normal\_end\_state$ instead of s' which forces SYMPLC to backtrack immediately.

In general, the state of a PLC program is a valuation of all variables as well as program counters (PC) of all tasks. However, since we are concerned with the program state only at the end of a hyper-period (where all tasks have ended and local variables are out of the scope), only the

valuation of global variables needs to be considered.

Let R be the set of all reachable states of a PLC program at the end of the hyper-period. Ideally, SYMPLC should generate enough test cases to cover all states in R. Experiments show that, due to the nature of these PLC programs, the termination condition can often be met after a few hyper-periods. It also means SYMPLC should be designed to terminate as soon as the symbolic execution procedure stops generating previously unexplored states.

Consider a benchmark program named *IndustrialAuto4* from [33], which contains a state machine whose state variable, `CSTATE6`, may take a number of values. A brute-force application of SYMPLC would result in exponentially many program paths as the number of hyper-periods increases. For example, after five hyper-periods, the number of executions becomes 3176. In contrast, applying our new stateful reduction decreases the total number of executions down to 45. Furthermore, since the symbolic execution procedure detects the early-termination condition after 3 hyper-periods, all unfalsified properties are considered to be formally proved.

5.6 Evaluation

5.6.1 Subjects and Methodology

I have implemented SYMPLC and evaluated it on 93 PLC benchmark programs, including 49 single-task programs and 44 multi-task programs. In total, they consist of 26,713 lines of ST code, which translate to 62,926 lines of C code. Properties are expressed as assertions embedded in the source code. During the experiments, I evaluated the execution time of SYMPLC as well as its effectiveness in detecting property violations. I also compared the PLC-specific reduction techniques with state-of-the-art POR techniques; for comparison, we implemented the DPOR al-

gorithm [51] in SYMPLC. The experimental results show that SYMPLC can efficiently generate test cases for all benchmark programs, and for multi-task PLC programs, in particular, the new reduction techniques significantly outperform the state-of-the-art POR technique.

I have implemented SYMPLC based on the *Matiec* PLC compiler [87] and the *Cloud9* symbolic virtual machine [38]. SYMPLC uses *Matiec* compiler to translate ST code of each PLC task to ANSI C, and creates a test harness to incorporate these tasks. The resulting multithreaded C model is then executed by the extended *Cloud9*, which uses KLEE [23] internally for symbolic execution. SYMPLC extended *Cloud9* to handle the PLC-specific program features.

I designed the experimental evaluation for the following research questions:

- Can SYMPLC efficiently handle both single-task and multi-task PLC programs? Is SYMPLC effective in detecting property violations as well as proving their correctness?
- Are the PLC-specific reduction techniques (stateful, period, and priority) effective in reducing the search space? Do they outperform state-of-the-art POR techniques?

For comparison purposes, I implemented the state-of-the-art dynamic partial-order reduction (DPOR) algorithm [51] in SYMPLC to identify and remove redundant interleavings.

I evaluated SYMPLC on two sets of benchmark programs. The first set consists of 49 single-task PLC programs collected from various online sources [31, 33, 71]. The second set consists of 44 multi-task PLC programs that implement several embedded controllers [26, 27]. Each PLC program has 30 to 3,418 lines of ST code, which translate to 90 to 8,783 lines of C code. In total, they consist of 26,713 lines of ST code, which translate to 62,926 lines of C code. The C code is first compiled to LLVM bitcode and then symbolically executed by the modified *Cloud9*. Correctness properties are expressed as assertions embedded in the programs. All experiments execute on a desktop computer with 8 GB RAM and a 3.4 GHz CPU running Ubuntu 12.04 Linux.

5.6.2 Results on Single-task PLC Applications

Table 5.2 shows the experimental results on single-task PLC programs. Since each hyper-period has one task, the number of iterations is the same as the number of tasks executed. Here Columns 1–3 present the program name, the number of lines of original ST code, and the number of lines of generated C code of each benchmark. Columns 4-8 show the detailed results of SYMPLC, including the maximum number of iterations reached (#.Iter), whether stateful reduction detected convergence (Conv), the number of tests generated, execution time in seconds, and the instruction coverage (#.ICov). The last three columns show the assertion checking results, including the number of undecided, falsified, and proved assertions.

If SYMPLC finds an execution that fails an assertion, the assertion is falsified. If SYMPLC does not find such an execution before reaching early termination, the assertion is proved. Otherwise, the assertion remains undecided.

Traditionally, symbolic execution is understood as a technique more suitable for falsifying assertions than proving their correctness. That is why results in Table 5.2 are a pleasant surprise: since our stateful reduction is effective in detecting early-termination conditions, SYMPLC can prove 154 assertions (in addition to falsifying 34 assertions). As a result, there are only 18 undecided assertions. In contrast, without stateful reduction, the total number of undecided assertions would have been 172.

Furthermore, the number of iterations ranges from 2 to 14, indicating that repeatedly executing the same PLC tasks after that many hyper-periods does not lead to new program states. Instead, the main difficulty in testing these PLC programs resides in covering the input space – it is precisely what symbolic execution is designed for.

The average *Instruction Coverage* for all benchmarks is 89.7%, which did not reach 100% even

Table 5.2: Results of SYMPLC on single-task PLC programs.

Program	LOC		#.Iter	Conv	#.Tests	Time (s)	#.ICov (%)	Assertions		
	ST	C						Undet.	Falsified	Proved
G4LTL_ST1	470	1,249	5	Y	305	36.2	89.1	0	0	1
G4LTL_ST2	188	504	5	Y	316	13.2	87.5	0	0	1
G4LTL_ST3	111	252	6	Y	116	4.9	87.3	0	0	2
G4LTL_ST4	1,409	4,279	7	Y	1,498	140.2	44.5	0	1	1
G4LTL_ST5	321	855	5	Y	240	9.6	97.8	0	0	2
G4LTL_ST6	69	154	2	Y	67	2.5	98.5	0	2	1
G4LTL_ST7	86	156	2	Y	272	1231.9	78.2	0	0	1
G4LTL_ST8	488	1,661	8	Y	368	16.3	74.4	0	0	5
G4LTL_ST9	577	914	10	Y	686	69.7	92.6	0	1	1
G4LTL_ST10	257	435	6	Y	354	27.8	99.8	0	1	2
IndustrialAuto1	45	145	2	Y	12	0.6	95.1	0	0	2
IndustrialAuto2	43	150	2	Y	10	0.5	95.5	0	2	2
IndustrialAuto3	206	379	4	Y	289	10.6	99.7	0	1	3
IndustrialAuto4	65	172	3	Y	45	1.7	98.9	0	0	3
IndustrialAuto5	105	273	3	Y	65	2.1	98.8	0	1	7
IndustrialAuto6	126	276	6	Y	25	1.0	84.1	0	0	9
IndustrialAuto7	126	275	5	Y	34	1.3	92.4	0	0	8
IndustrialAuto8	199	485	13	Y	55	3.1	60.0	0	0	11
IndustrialAuto9	2,444	8,291	14	Y	143	8.8	11.7	0	2	9
IndustrialAuto10	1,195	3,266	12	No	5,084	>3600	34.3	15	1	0
IndustrialAuto11	75	218	3	Y	23	0.9	97.8	0	0	8
IndustrialAuto12	1,580	3,781	6	Y	3,216	813.1	99.9	0	0	14
IndustrialAuto13	255	607	8	Y	130	4.9	61.5	0	1	6
IndustrialAuto15	3,418	8,783	8	Y	1,349	151.8	65.7	0	1	11
IEC-1	30	90	2	Y	6	0.4	83.2	0	0	1
IEC-2	53	135	2	Y	619	25.6	97.9	0	1	1
IEC-3	118	260	2	No	8,041	>3600	98.4	2	2	0
IEC-4	66	173	6	Y	216	9.1	92.5	0	0	3
IEC-5	32	72	2	Y	6	0.3	88.2	0	0	2
IEC-6	52	140	2	Y	306	9.9	98.0	0	0	1
IEC-7	173	474	3	Y	37	2.5	85.1	0	1	1
LD-Program1	89	136	3	Y	1,084	64.5	79.1	0	0	4
LD-Program2	336	403	2	No	10,508	>3600	67.0	1	0	0
LD-Program3	92	135	11	Y	231	10.3	99.6	0	1	1
LD-Program4	110	150	2	Y	601	29.9	74.9	0	1	2
Mixer	181	251	4	No	5,088	>3600	88.6	0	2	0
Evaporator	178	238	2	Y	287	11.9	81.7	0	1	3
Hydraulic	118	128	2	Y	54	2.4	83.0	0	0	4
Safe	215	313	2	Y	1,724	199.5	92.0	0	0	2
Logic	234	322	2	Y	125	6.2	78.9	0	1	8
Lift	187	169	2	Y	160	8.1	71.6	0	0	2
Plastic	187	215	2	Y	360	22.6	75.1	0	0	2
Bargraph	126	143	4	Y	4,316	429.6	98.0	0	0	2
Jedyka	80	92	7	Y	132	4.5	99.5	0	2	0
Glowny	70	86	4	Y	62	2.1	93.5	0	2	1
IL-Tool	137	171	2	Y	362	20.4	99.2	0	1	2
Shutter	83	125	4	Y	643	49.6	94.3	0	3	0
Alarm	68	107	4	Y	326	22.4	99.5	0	1	1
Fountain	50	95	9	Y	339	28.7	99.6	0	1	1
Total	16,923	42,183			50,335	17,913		**18**	**34**	**154**

for benchmarks that converged, apparently because some of these instructions are unreachable.

5.6.3 Results on Multi-task PLC Applications

This section analyzes the performance differences between non-stateful and stateful exploration inside SYMPLC, and then compare the various interleaving reduction techniques.

Table 5.3 shows the results on multi-task PLC programs. Columns 1-3 present the program name and statistics of the hyper-period, including the total number of tasks and global operations executed in each hyper-period, because they are closely related to the complexity of the interleaving exploration. Columns 4-9 show results of SYMPLC without stateful reduction, including the maximum number of iterations reached, the amount of generated test cases, the run time, and the assertion checking results. Columns 10-15 show results of SYMPLC with stateful reduction. Here the time bound is 10 minutes and the hyper-period iteration bound is also 10.

Since non-stateful SYMPLC cannot detect convergence, it does not prove properties. In contrast, stateful SYMPLC can prove the correctness of properties. The results show that stateful SYMPLC only needed a few hyper-periods to detect convergence. In contrast, non-stateful SYMPLC frequently timed out or generated more test cases (1.4 million versus 11K). Both two methods detected 17 assertion violations, but stateful SYMPLC also proved 27 assertions, whereas non-stateful SYMPLC did not.

Table 5.4 shows the result of comparing different interleaving reduction techniques. Here, KLEE denotes the default symbolic execution algorithm in *Cloud9* augmented with the capability of handling threads. DPOR denotes the enhanced version of KLEE where I added the implementation of dynamic partial order reduction. Among the three PLC-specific reductions, *Period* denotes the period-based reduction technique, *Priority* denotes the priority-based reduction technique, and

Table 5.3: Results of SYMPLC on multi-task PLC programs.

Program	Hyper-period		Non-Stateful						Stateful					
	#.Task	#.Ops	#.Iter	#.Test	#.Time	Assertions			#.Iter	#.Test	#.Time	Assertions		
						Und	Fal	Pro				Und	Fal	Pro
nxt2.prog1	3	16	10	10	0.2	1	0	0	3	3	0.4	0	0	1
nxt2.prog2	3	16	10	1027	2.8	0	1	0	2	7	0.4	0	1	0
nxt2.prog3	5	27	10	59048	372.2	1	0	0	2	5	0.4	0	0	1
nxt2.prog4	7	37	10	59767	>600	1	0	0	2	5	0.4	0	0	1
nxt2.prog5	5	28	4	109,669	>600	0	1	0	2	43	0.7	0	1	0
nxt2.prog6	7	38	4	71,631	>600	0	1	0	2	43	0.8	0	1	0
nxt2.prog7	7	38	3	66,907	>600	1	0	0	2	91	1.4	0	0	1
nxt3.prog1	5	15	5	43,313	>600	1	0	0	3	24	0.6	0	0	1
nxt3.prog2	7	19	4	57,396	>600	0	1	0	2	93	1.2	0	1	0
nxt3.prog3	8	28	4	15,080	>600	1	0	0	2	20	0.8	0	0	1
nxtway01	6	42	5	41,629	>600	1	0	0	2	25	0.6	0	0	1
nxtway02	6	46	3	35,449	>600	0	1	0	3	149	2.4	0	1	0
nxtway03	9	62	3	33,809	>600	0	1	0	3	978	16.2	0	1	0
nxtway04	9	66	4	26,988	>600	0	1	0	2	575	10.4	0	1	0
nxtway05	6	38	9	11122	>600	1	0	0	3	11	0.6	0	0	1
nxtway06	6	34	4	46,580	>600	0	1	0	3	860	12.1	0	1	0
nxt.pi00	6	46	5	21,808	>600	0	1	0	2	55	1.2	0	1	0
nxt.pi01	6	46	4	23,348	>600	1	0	0	2	98	1.7	0	0	1
nxt.pi02	8	62	3	21,139	>600	0	1	0	2	368	6.5	0	1	0
nxt.pi03	5	38	4	25,406	>600	0	1	0	3	179	2.9	0	1	0
trans01	6	41	3	26,367	>600	0	1	0	3	502	7.9	0	1	0
trans02	6	41	6	498	>600	1	0	0	4	27	0.9	0	0	1
trans03	6	39	4	33,572	>600	0	1	0	5	2,638	38.7	0	1	0
trans04	6	41	4	29,326	>600	1	0	0	3	73	1.4	0	0	1
trans05	9	59	5	11582	>600	1	0	0	3	19	0.8	0	0	1
attend01	6	35	5	34,658	>600	1	0	0	4	64	1.2	0	0	1
attend02	6	42	3	32,932	>600	0	1	0	3	388	5.9	0	1	0
attend03	6	48	4	20,723	>600	1	0	0	2	119	2.3	0	0	1
attend04	6	39	4	23,537	>600	1	0	0	3	101	1.9	0	0	1
att4_01	7	40	3	23,655	>600	1	0	0	3	855	14.5	0	0	1
att4_02	7	33	4	32,505	>600	1	0	0	3	105	1.9	0	0	1
race01	6	47	3	21,990	>600	0	1	0	2	166	2.6	0	1	0
race02	6	38	5	17,730	>600	0	1	0	3	41	0.9	0	1	0
race03	9	63	3	14,960	>600	1	0	0	2	275	6.7	0	0	1
nobadmode01	6	34	4	22,915	>600	1	0	0	3	88	1.5	0	0	1
nobadmode02	7	45	4	10,840	>600	1	0	0	3	86	2.3	0	0	1
nobadmode03	6	52	3	14,051	>600	1	0	0	3	614	13.1	0	0	1
nobadmode04	6	51	5	12,666	>600	1	0	0	3	102	2.2	0	0	1
ctm01	7	123	7	47067	>600	1	0	0	9	131	2.2	0	0	1
ctm02	7	120	5	82,330	>600	1	0	0	4	178	2.5	0	0	1
ctm03	6	115	9	72,135	>600	1	0	0	5	82	1.2	0	0	1
aso_01	11	67	2	20,269	>600	1	0	0	2	438	12.4	0	0	1
aso_02	9	49	2	19,204	>600	1	0	0	2	90	2.9	0	0	1
aso_03	9	53	3	26,610	>600	0	1	0	2	452	9.4	0	1	0
Total				1,423,248	24,975	**27**	**17**	**0**		11,266	199	**0**	**17**	**27**

Table 5.4: Results of comparing different reduction techniques.

Program	KLEE [23]		DPOR [51]		PLC-specific Reductions					
					Period		Priority		Period+Priority	
	#. Test	Time (s)	#. Test	Time (s)	#. Test	Time (s)	#. Test	Time (s)	#. Test	Time (s)
nxt2.prog1	43,960	>600	135	1.6	135	1.6	3	0.4	3	0.4
nxt2.prog2	44,200	>600	19	0.5	19	0.5	7	0.4	7	0.4
nxt2.prog3	44,595	>600	5,644	62.2	3,839	39.5	24	0.6	5	0.4
nxt2.prog4	45,683	>600	47,652	>600	35,531	452.3	40	0.7	5	0.4
nxt2.prog5	50,067	>600	52,666	>600	47,422	>600	144	1.5	43	0.7
nxt2.prog6	44,442	>600	46,521	>600	45,867	>600	528	5.1	43	0.8
nxt2.prog7	44,383	>600	46,280	>600	45,394	>600	827	9.8	91	1.4
nxt3.prog1	47,159	>600	48,111	>600	44,560	>600	250	3.3	24	0.6
nxt3.prog2	52,490	>600	54,248	>600	41,011	>600	5,792	59.9	93	1.2
nxt3.prog3	45,842	>600	54,851	>600	25,527	>600	3,117	37.8	20	0.8
nxtway01	53,425	>600	43,799	>600	41,003	>600	195	2.7	25	0.6
nxtway02	51,317	>600	45,304	>600	45,633	>600	1,646	22.2	149	2.4
nxtway03	51,609	>600	35,932	>600	39,562	>600	35,445	>600	978	16.2
nxtway04	60,785	>600	45,557	>600	38,916	>600	31,045	>600	575	10.4
nxtway05	51,589	>600	38,140	>600	39,562	>600	713	10.1	11	0.6
nxtway06	46,392	>600	50,025	>600	45,756	>600	4,608	68.6	860	12.1
nxt.pi00	49,470	>600	39,158	>600	40,716	>600	561	8.2	55	1.2
nxt.pi01	49,978	>600	40,446	>600	39,212	>600	1,226	16.9	98	1.7
nxt.pi02	48,670	>600	28,824	>600	29,922	>600	10,216	169.9	368	6.5
nxt.pi03	43,048	>600	40,384	>600	39,884	>600	239	3.7	179	2.9
trans01	40,869	>600	42,269	>600	40,684	>600	5,170	78.9	502	7.9
trans02	40,893	>600	38,720	>600	39,364	>600	99	2.1	27	0.9
trans03	36,714	>600	43,546	>600	43,602	>600	11,112	168.6	2,638	38.7
trans04	38,568	>600	35,123	>600	25,535	>600	357	5.3	73	1.4
trans05	49,880	>600	46,138	>600	35,480	>600	4,207	69.3	19	0.8
attend01	43,298	>600	54,320	>600	57,849	>600	222	3.1	64	1.2
attend02	32,541	>600	56,541	>600	18,866	>600	20,292	12.8	388	5.9
attend03	46,143	>600	35,023	>600	35,585	>600	343	5.1	119	2.3
attend04	45,951	>600	36,224	>600	32,570	>600	432	6.5	101	1.9
att4_01	47,312	>600	40,691	>600	36,861	>600	2,391	37.2	855	14.5
att4_02	46,969	>600	34,278	>600	40,327	>600	364	5.5	105	1.9
race01	45,610	>600	38,334	>600	37,424	>600	500	7.2	166	2.6
race02	45,277	>600	17,756	231.9	2,236	32.2	176	3.9	41	0.9
race03	44,974	>600	22,145	>600	32,658	>600	28,112	>600	275	6.7
nobadmode01	49,094	>600	46,622	>600	40,932	>600	314	4.6	88	1.5
nobadmode02	43,561	>600	51,877	>600	6,616	144.6	16,862	346.7	86	2.3
nobadmode03	43,525	>600	49,719	>600	48,408	>600	2,745	49.9	614	13.1
nobadmode04	43,962	>600	42,283	>600	37,739	>600	2,558	46.5	102	2.2
ctm01	51,345	>600	51,810	>600	776	11.4	846	10.9	131	2.2
ctm02	55,711	>600	63,599	>600	54,294	>600	621	7.8	178	2.5
ctm03	49,343	>600	51,875	>600	41,839	>600	97	1.3	82	1.2
aso_01	42,260	>600	39,386	>600	10,582	>600	20,607	>600	438	12.4
aso_02	44,403	>600	43,847	>600	12,461	>600	29,191	>600	90	2.9
aso_03	44,235	>600	40,646	>600	11,785	>600	23,108	>600	452	9.4
Total	**2,041,542**	26,400	**1,786,468**	24,296	**1,433,944**	22,822	**267,352**	4,895	**11,266**	199

Period+Priority denotes the full-blown implementation of the reductions in SYMPLC. All methods shown in Table 5.4 were used in conjunction with the stateful reduction. For each individual method, the result table shows the number of test cases generated and the total execution time in seconds. Since the time limit was set to 10 minutes, >600s means the corresponding method was forced to terminate after running out of time.

As shown in the total numbers in the last row, the full-blown reduction implemented in SYMPLC, denoted *(Period+Priority)*, significantly outperformed KLEE and DPOR, two state-of-the-art symbolic execution techniques. Specifically, the reduction in the number of test cases is more than two orders of magnitude. Furthermore, the full-blown reduction is significantly more efficient than *Period*-based reduction (11,266 versus 1,433,944) or *Priority*-based reduction (11,266 versus 267,352) alone. This means applying both *Period* and *Priority* based reductions has led to synergistic impact.

5.7 Conclusion

I have presented the first symbolic execution tool SYMPLC for automatically testing single- and multi-task PLC programs. SYMPLC takes the PLC source code as input, translates it into C code, and then applies symbolic execution. As such, it can systematically explore feasible paths of individual PLC tasks as well as their interleavings. Toward this end, the main contribution is developing a number of PLC-specific reduction techniques for eliminating redundant interleavings. Experimental results show that SYMPLC is efficient in handling a large number of PLC benchmark programs. On multi-task PLC programs, in particular, the new reduction techniques significantly outperform the state-of-the-art partial order reductions technique.

Chapter 6

Adversarial Symbolic Execution

Side-channel attacks are security attacks where an adversary exploits the dependency between sensitive data and non-functional properties of a program such as the execution time [43, 75], power consumption [76, 86], heat, sound [54], and electromagnetic radiation [53, 102]. For timing side channels, in particular, there are two main sources of leaks: variances in the number of executed instructions and variances in the cache behavior.

Instruction-induced leaks are caused by differences in the number and type of instructions executed along different paths: unless the differences are independent of the sensitive data, they may be exploited by an adversary. Cache is a high-speed storage between the fast CPU and the slow memory where cache-induced leaks are caused by differences in the number of cache hits and misses along different paths. The corresponding timing characteristics may reveal sensitive information of the program, thus allowing an adversary to conduct side-channel attacks.

Existing methods for detecting timing leaks or proving their absence often ignore the cache all together while focusing on instruction-induced leaks. For example, Chen et al. [32] used Cartesian

Hoare Logic [113] to prove the timing leak of a program is within a bound; Antonopoulos et al. [9] used a similar technique that partitions the set of program paths in a way that, if individual partitions are proved to be timing attack resilient, the entire program is also timing attack resilient. Unfortunately, these methods ignore the cache-timing characteristics. Even for techniques that consider the cache [13, 28, 37, 46, 78, 119], their focuses have been on leaks manifested by the program itself when running alone, without considering the cases when it is executed concurrently with some other (benign or adversarial) threads.

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

6.1 Introduction

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

Figure 6.1 shows the flow of the new leak detector SYMSC, which takes the victim thread P, a

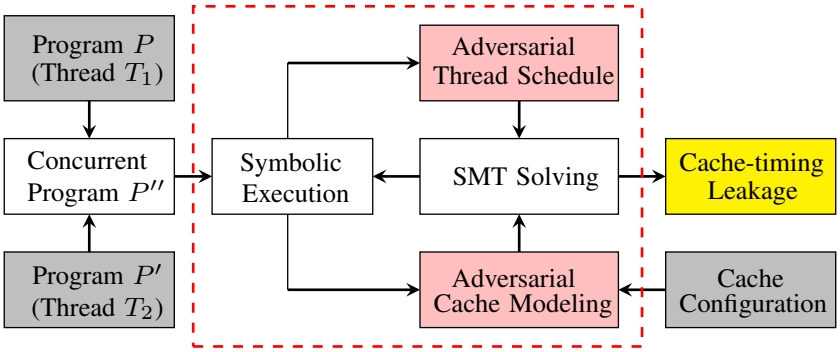


Figure 6.1: Flow of the cache timing leak detector SYMSC.

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

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

To address the second challenge, I developed a technique for modeling the cache behavior of a program based on the cache's type and configuration, as well as optimizations of the subsequent constraint solving to reduce overhead. For each concurrent execution (an interleaving of the threads) denoted $\pi = (in, sch)$, where in is the sensitive data input and sch is the interleaving schedule, SYMSC constructs a logical constraint $\tau_t(in, sch)$ for every potentially adversarial memory access t, to indicate when it leads to a cache hit. Then, SYMSC seeks two distinct values of the data input,

in and in', for which the cache behaves differently: $\tau_t(in, sch) \neq \tau_t(in', sch)$, meaning one of them is a hit but the other is a miss, and they are due to differences in the sensitive data input.

I have implemented SYMSC based on LLVM and the KLEE symbolic virtual machine [23], and evaluated it on twenty benchmark programs. These security-critical programs are ciphers taken from cryptographic libraries in the public domain; they have 14,455 lines of C code in total. Since these programs are crafted by domain experts, they do not have obvious timing leaks when running alone, such as unbalanced branching statements or variances in lookup-table accesses. However, the experiments of applying SYMSC show that they may still have timing leaks when being executed concurrently with other threads.

To summarize, I make the following contributions:

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

6.2 Motivation

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

Figure 6.2(a) shows a program whose execution time is dependent of the sensitive variable `k`. It is a revised version of the running example used in [30], for which the authors proposed the leak-free version shown in Figure 6.2(b). The two programs have the same set of instructions but differ in where the highlighted `load` instruction is located: line 5 in P and line 9 in P_r.

6.2.1 A Self-leaking Program and the Repair

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

(a) A leaky program P

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

0≤k≤255:  <miss, miss, hit>
```

(b) The leak-free version P_r

Figure 6.2: A program with cache-timing leak (cf. [30]).

Consider executing the two programs under a 512-byte direct-mapped cache with one byte per cache line, as shown in Figure 6.3. The choice of one-byte-per-cache-line — same as in [30] — is meant to simplify analysis without loss of generality. Specifically, the 256-byte array `p` is associated with the first 256 cache lines, while variable `k` is associated with the 257-th cache line. Due to the finite cache size, `q[255]` has to share the cache line with `p[0]`.

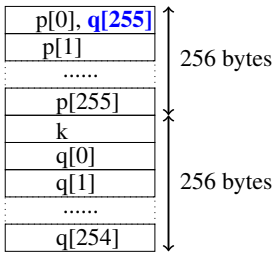


Figure 6.3: The direct-mapped cache layout (cf. [30]).

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

Assume that `k`'s value is 0, executing P means taking the `then` branch and accessing `p[0]`,

q[255], and p[0]. The first access to p[0] is a cold miss since the cache is empty at the moment. The access to q[255] is a conflict miss because the cache line (shared by q[255] and p[0]) is occupied by p[0]; as a result q[255] evicts p[0]. The next access to p[0] is also a conflict miss since the cache line is occupied by q[255]. All in all, the cache behavior is <miss,miss,miss> for k=0.

This sequence is also unique in that all other values of k would produce <miss,miss,hit> as shown at the bottom of Figure 6.2(a). This means P, when running alone, leaks information about k. For example, upon observing the delay caused by <miss,miss,miss> via monitoring, an adversary may infer that k's value is 0.

Program P_r is a repaired version [30] where the load is moved from line 5 to line 9 as in Figure 6.2(b). Thus, the load accessing p[k] at line 9 always generates a cold miss (0<k≤255) or a conflict miss (k=0). Consequently, the store at line 11 is always a hit. Thus, for all values of k, the cache behavior remains <miss,miss,hit> – no information of k is leaked.

6.2.2 New Leak Induced by Concurrency

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

Figure 6.4 shows a two-threaded program comprising T_1 and an adversarial T_2 that accesses a new variable tmp. Assume tmp is mapped to the same cache line as p[1]. Then, it is possible for T_2 to cause T_1 to leak information of its secret data. There are various ways of mapping tmp to the same cache line as p[1], e.g., by dynamically allocating the memory used by tmp or invoking a recursive (or non-recursive) function within which tmp is defined as a stack variable.

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

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

Figure 6.4: Concurrent program with side-channel leak.

Table 6.1 shows the six interleavings of threads T_1 and T_2. The left half of this table contains three interleavings where T_1 took the `then` branch of the if-statement, while the right half contains three interleavings where T_1 took the `else` branch. In each case, the four columns show the ID, the execution order, the cache sequence of thread T_1, and the value range of `k`. For example, in `6-9-11-13`, the `store` at line 11 is a cache hit because its immediate predecessor (line 9) already loads `p[k]` into the cache. Since the last `load` at line 13 comes from thread T_2, the cache behavior sequence of T_1 is <`miss,miss,hit`>, denoted <`m,m,h`> for brevity.

Table 6.1: Interleavings and thread T_1's cache sequences.

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

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

Figure 6.5 shows details of interleaving `6-9-13-11`. The blue and orange rectangles represent the `load` and `store` accesses, respectively, and the red dashed poly-line shows their execution order. The first three `load` operations all cause cache misses, whereas the last `store` could be a

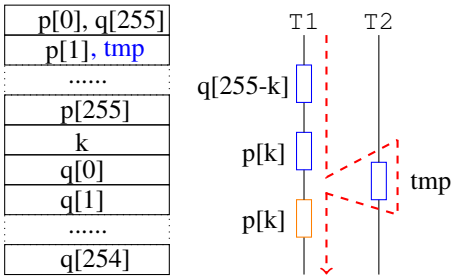


Figure 6.5: Interleaving `6-9-13-11` and the cache layout.

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

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

6.2.3 Adversarial Symbolic Execution

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

Case 1. Thread T_2 is given, together with fixed addresses of the memory region accessed by T_2. In this case, T_2 is an integral part of the concurrent system that also contains the security-critical computation in T_1. Since the only source of non-determinism is thread interleaving, SYMSC aims

to check if the concurrent system itself has timing leaks.

Case 2. Thread T_2 is not given, but created by SYMSC, *and thus the addresses of the memory region accessed by T_2 are assumed to be symbolic.* This is when, inside the cache layout of Figure 6.5, the address of `tmp` would be made symbolic, thus allowing it to be mapped to any cache line (as opposed to be fixed to the 2nd line). There are now two sources of non-determinism: thread interleaving and memory layout. SYMSC explores both to check if T_1 may leak information due to interference from T_2.

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

6.3 The Threat Model

This section reviews the technical background and present the threat model, which defines what an adversary can or cannot do.

6.3.1 Cache and the Timing Side Channels

The execution time of a program depends on the CPU cycles taken to execute the instructions and the time needed to access memory. The first component is easy to compute but also less important in practice, because security-critical applications often execute the same set of instructions regardless of values of their sensitive variables [126]. In contrast, leaks are more likely to occur in the second component: the time taken to access memory. Compared to the time needed to execute an

instruction, which may be 1-3 clock cycles, the time taken to access memory, during a cache miss, may be tens or even hundreds of clock cycles.

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

Let's assume the security-critical program P implements a function $c \leftarrow f(k, x)$, where k is the sensitive input (secret), x is the non-sensitive input (public), and c is the output. In block ciphers, for example, k would be the cryptographic key, x would be the plaintext, c would be the ciphertext, and f would be the encryption or decryption procedure. Let the execution time of P be $\tau_P(k, x)$. Since there may be multiple paths inside P, when referring to a particular path $p \in P$, we use $\tau_p(k, x)$. But if there is no ambiguity, we may omit the detail and simply use $\tau(k, x)$. We say P is leak-free if $\tau(k, x)$ remains the same for all input values. That is,

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

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

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

Here, SYMSC searches for two values of k that can lead to differences. If the set of instructions executed by P remains the same, SYMSC only needs to check whether $\tau(k_1, x)$ and $\tau(k_2, x)$ have the same number of cache hits and misses. Furthermore, in the threat model where the attacker

can only observe (passively) the execution time of P, but not control or observe x, SYMSC can reduce the computational cost by fixing a value $\overline{x}$ of x arbitrarily and then checking if $\tau(k_1)$ and $\tau(k_2)$ have the same number of cache hits and misses.

6.3.2 Example of an Attack

This section shows a concrete example of exploiting cache timing leaks in concurrent systems. The goal is to illustrate what an adversary may be able to achieve in practice.

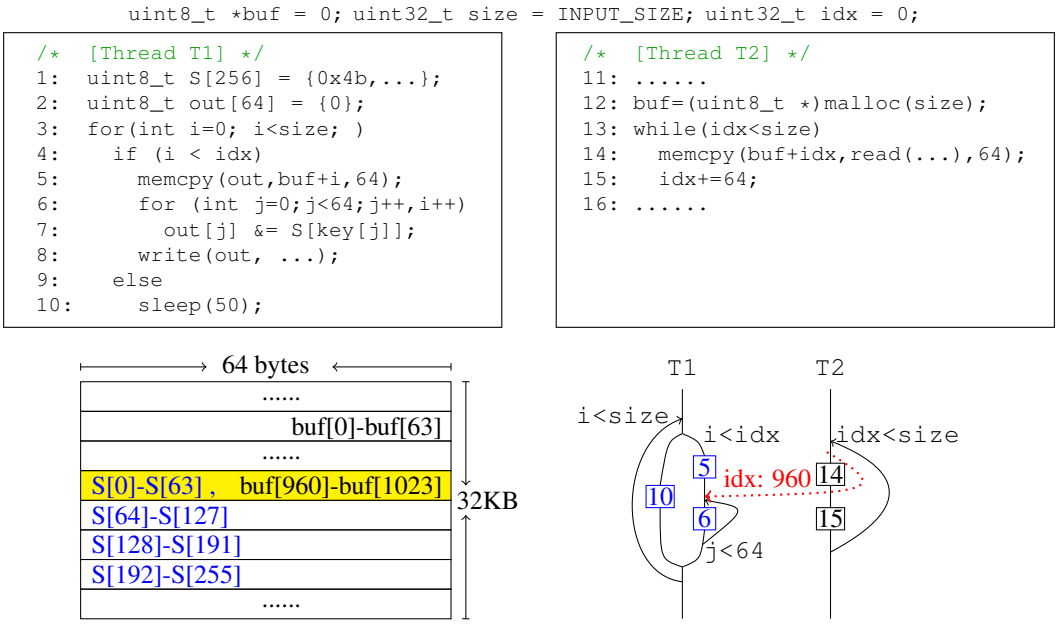


Figure 6.6: A two-threaded encryption program.

Figure 6.6 shows a two-threaded program, its cache mapping, and the thread-local control flows. Initially, `T2` allocates a memory area (`buf`) whose size matches the input. Although the input size may be arbitrary, here, let's assume it is an integral multiple of 64, e.g., 1024 bytes (`INPUT_SIZE`=1024). In the *while*-loop (line 14) `T2` reads 64 bytes from input every time to fill `buf`. Thread `T1` tracks the progress (`idx`) of `T2` (line 4) and repeatedly retrieves 64-byte data from `buf` to the array `out` (line 5). The encryption on `out` involves the S-Box array `S` and a given

`key` (lines 6-7). Once the data is encrypted, `T1` sends it out (line 8). When `T1` finds that `buf` runs out of data, it sleeps for 50ms (line 10).

First, I explain why the program has a timing leak. Here a 32KB direct-mapped cache is used and each cache line is 64 bytes. The S-Box array `S` hence maps to 4 cache lines and the `buf` array maps to 16 cache lines. For brevity, I only focus on the important arrays (`S` and `buf`) while assuming other variables do not affect the cache mapping. Furthermore, in this example I assume `S` and `buf` share one cache line as highlighted in Figure 6.6.

The graph in Figure 6.6 shows an interleaving of `T1` and `T2`, where the dotted red arrow represents a context switch after `T2` executes the `memcpy` statement (line 14) while `T1` just reaches the *for*-loop at line 6. The text above the arrow means `idx`'s value is 960 at the moment, indicating thread `T2` has just accessed the last 64 bytes of `buf` at line 14.

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

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

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

Thus, the attacker could mutate the input to alter the buffer size, hence affecting the memory layout. Furthermore, real applications sometimes use relatively large fixed buffers. For example, in OpenSSH [6], the `scp` program has a 16KB buffer for `COPY_BUFLEN` and the `sftp` program has a 32KB buffer for `DEFAULT_COPY_BUFFER`. Moreover, OpenSSH's `SSHBUF_SIZE_MAX` buffer for a socket channel is as large as 256MB. These large buffers allow room for attackers to construct the desired cache layout.

A similar scenario can be found in the open-source implementation of HPN-SSH [5], which is an enhancement of OpenSSH [6] by leveraging multi-threading to accelerate the data encryption. Figure 6.7 shows the code snippet directly taken from the HPN-SSH [5] repository: On the left-hand side are threads created to run the `thread_loop` function, shown on the right-hand side, which repeatedly calls `AES_encrypt` to encrypt data given by the user (line 327). By controlling the size and content of the data, as well as the number of threads, a malicious user is able to affect both the memory layout and the thread interleaving.

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

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

Figure 6.7: Concurrency-related code in HPN-SSH [5].

The experimental evaluation in Section 6.7) shows that the AES subroutine from OpenSSL in-

deed has cache timing leaks, which may subject HPN-SSH to attack scenarios similar to the one illustrated in Figure 6.6.

6.4 Adversarial Symbolic Execution

This section presents the baseline algorithm and then enhances it to search for cache timing leaks.

6.4.1 The Baseline Algorithm

Here let's assume that the entire program consists of a finite set $\{T_1, \ldots, T_n\}$ of threads where each thread T_i ($1 \leq i \leq n$) is a sequential program. Without loss of generality, let's assume T_1 is critical and any of $T_2, \ldots, T_n$ may be adversarial.

Still, Algorithm 1 acts as the baseline symbolic execution procedure except that, for the purpose of detecting timing leaks, it considers two events as *dependent* also when they are mapped to the same cache line. Here, an execution is characterized by $\pi = (in, sch)$ where $in = \{k, x\}$ is the data input and sch is the thread schedule, corresponding to a total order of events $e_1 \ldots e_n$, and the stack S is a container for symbolic states. Each $s \in S$ is a tuple $\langle \mathcal{M}, pcon, branch, enabled, done, crt \rangle$, where $\mathcal{M}$, $pcon$, $branch$, $enabled$ and $done$ retain their original meanings in Algorithm 1 and the new item crt denotes the event chosen to execute at s. I omit other details since they are consistent with those in Algorithm 1.

Also note that, in the prior work, symbolic execution would allow interleavings between global (γ) events only if they have *data conflicts*, i.e., two events from different threads are accessing the same memory address, and at most one of them is a memory read. This is because only such

accesses may lead to different states if they are executed in different orders. However, in SYMSC, whether these events are mapped to the same cache line also matters.

6.4.2 Enhanced Algorithm

Algorithm 8 shows the enhancement in SYMSC over the baseline algorithm 1, where the main difference is in the interleaving points (highlighted in blue). Upon entering the *while*-loop at line 8, it first checks if an enabled event t may lead to a timing leak by invoking *DivergentCacheBehavior(s,t)*. Details of the subroutine will be presented in Section 6.5, but at the high level, it constructs a cache behavior constraint τ_t and then searches for two values, $\overline{k_1}$ and $\overline{k_2}$, such that $\tau_t(\overline{k_1}) \neq \tau_t(\overline{k_2})$.

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

For the running example in Figure 6.4, in particular, Algorithm 8 would explore the first three interleavings in Table 6.1 before detecting the leak. The process is partially illustrated by Figure 6.8, where events t_1:`load q[255-k]`, t_2:`load p[k]` and t_3:`store p[k]` belong to thread T_1 whereas t_4:`load tmp` belongs to thread T_2.

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

Algorithm 8: Symbolic Execution in SYMSC

```
1  Initially: State stack S = ∅; Start Explore(s_0) with the symbolic state s_0.
2  Explore(State s)
3  begin
4      S.push(s);
5      if s is a b-PP node then
6          ...
7      else if s is an i-PP node then
8          while ∃t ∈ s.enabled\s.done do
9              if s is a memory access point then
10                 if DivergentCacheBehavior(s, t) then
11                     generate test case;
12                     s.done ← s.done ∪ {t};
13                     continue;
14                 end
15             end
16             Explore(NextSymbolicState(s, t));
17             s.done ← s.done ∪ {t};
18         end
19     else if s is other sequential computation node then
20         ...
21     else
22         ...
23     end
24     S.pop();
25 end
26 NextSymbolicState(State s, Event t)
27 begin
28     ...
29     s.crt ← t;
30     return s';
31 end
32 DivergentCacheBehavior(State s, Event t)
33 begin
34     if AdversarialAccess(s, t) then
35         τ_t ← compute t's cache hit constraint;
36         if ∃k, k' such that τ_t(k) ≠ τ_t(k') then
37             return true;
38         end
39     end
40     return false;
41 end
42 AdversarialAccess(State s, Event t)
43 begin
44     if t is from the critical thread then
45         let s' ∈ S and t' = s'.crt;
46         if ∃t' that t and t' may map to the same cache line then
47             return true;
48         end
49     end
50     return false;
51 end
```

differs from t_1's access `q[255-k]`, meaning the cache behavior at t_2 remains the same for all values of k.

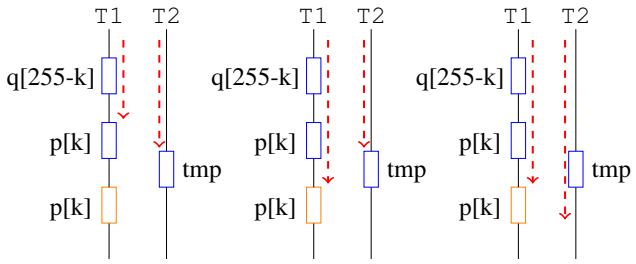


Figure 6.8: The three interleavings generated by SYMSC.

If t_2 were executed before t_4, we would have the second scenario in Figure 6.8. At this moment, *s.enabled* = { t_3, t_4 }. If t_4 is executed after t_3, the interleaving would be `6-9-11-13`, which does not have timing leaks either. But if t_4 were executed before t_3, the third scenario would appear in Figure 6.8, where *AdversarialAccess(*s, t_3*)* evaluates to true, $\tau_{t_3}(k)$ evaluates to *false* for (k=1) but to *true* for (k$\neq$1)$\wedge$(k$\leq$127), as shown in Table 6.2, leading to divergent cache behaviors in `6-9-13-11`.

6.5 Adversarial Cache Analysis

SYMSC's method for detecting divergent cache behaviors works as follows. First, it constructs the behavioral constraint for each memory access. Then, it solves the constraint to compute a pair of sensitive values that allows the constraint to return divergent results.

6.5.1 Cache Modeling

Recall that the entire program contains T_1 and T_2, among other threads, where T_1 invokes the critical computation and T_2 is potentially adversarial. During symbolic execution, SYMSC conducts context switches when `load` or `store` instructions may be mapped to the same cache line.

Here, each interleaving p corresponds to a data input in and a thread schedule sch. The data input is divided further into $in = \{k, x\}$, where k is sensitive (secret) and x is non-sensitive (public). Whenever the value of x is immaterial, SYMSC assumes $in = \{k\}$.

- An interleaving p is a sequence of memory accesses denoted $p(sch, in) = \{A_0, ..., A_n\}$ where sch represents the order of these accesses and in represents the data input.
- Each A_i, where $i \in [0, n]$, denotes a memory access.
- $pcon_i(k)$ is the path condition under which A_i is reached.

Thus, when $pcon_i(k)$ is true, meaning A_i is reachable, SYMSC checks if A_i leads to a cache hit:

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

Thus, the *cache-hit condition* defines as follows:

$$\tau_i(k) \equiv \bigvee_{0 \le j < i} \Big(tag(addr_j) = tag(addr_i) \wedge \forall l \in [j{+}1, i{-}1] \,\big|\, line(addr_l) \neq line(addr_i) \Big) \quad (6.1)$$

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

If A_i always causes a cache hit, or a miss, it cannot leak sensitive information because it implies

$\forall k_1, k_2 \,.\, \tau_i(k_1) = \tau_i(k_2)$. In contrast, if $\tau_i(k)$ evaluates to *true* for some value of k but to *false* for a different value of k, then it is a leak.

6.5.2 Leakage Detection

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

Precise Solution The precise formulation is as follows:

$$\exists k_1, k_2 \,.\, (k_1 \neq k_2) \wedge \tau_i(k_1) \neq \tau_i(k_2) \tag{6.2}$$

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

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

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

Since the formula solved in each step is (almost twice) smaller, the solving time can be reduced significantly. Furthermore, a valid solution ($\overline{k_1}$ and $\overline{k_2}$) is guaranteed to be a valid solution for the original formula as well. However, in general, the two-step approach is an under-approximation:

when it fails to find any solution, it is not a proof that no such solution exists. To make the two-step approach precise, one would have to apply it repeatedly, each time with a different $\overline{k_1}$ computed in the first step, until all solutions of $\overline{k_1}$ is covered. Nevertheless, experiments show that, in practice, applying it once is often accurate enough to detect the actual leak.

6.5.3 The Running Example

Let's revisit the example in Figure 6.4 to show how SYMSC detects the leak. Recall that SYMSC would generate the six interleavings shown in Table 6.1. For each interleaving, Table 6.2 shows the line number (#line) of every access A_i, path condition $pcon_i$, memory address $addr_i$, and the cache-hit constraint τ_i.

Table 6.2: Cache-related information of interleaving p.

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

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

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

cache miss.

Similarly, τ_3 for the `store` at line 11 is shown in the last row of Table 6.2. It is worth mentioning that τ_3 only compares `p[k]` ($addr_3$) with `tmp` ($addr_2$) and `p[k]` ($addr_1$) but not `q[255-k]` ($addr_0$) because SYMSC finds that, if the access to `tmp` does not evict the cache line used by its preceding access to `p[k]` ($addr_1$), the last `store` to `p[k]` ($addr_3$) must be a cache hit; SYMSC stops here to avoid further (and unnecessary) analysis.

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

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

6.6 Optimizations

Symbolic execution, when applied directly to cipher programs, may have a high computational overhead because of the heavy use of arithmetic computations and look-up tables in these programs. This section presents techniques for reducing the overhead based on two insights. First, when conducting cache analysis, we are not concerned with the actual numerical computations inside the cipher unless they affect the addresses of memory accesses that may depend on sensitive data, e.g., indices of lookup tables such as S-Boxes. Second, for the purpose of detecting leaks, as opposed to proving their absence, we are free to under-approximate as long as it does not diminish the leak-detection capability of our analysis.

6.6.1 Domain-specific Reduction

By studying real-world cipher programs, I have found the computational overhead is often associated with symbolic indices of lookup tables such as the one shown in Figure 6.9.

```
1     uint8_t SBOX1[64] =
      {0x6f,0x3c,0x77,0xb7,0x2f,0x7b,0x5f,0xc6, ...};
2     uint8_t SBOX2[64] =
      {0x3d,0x4c,0x5f,0xb6,0xd1,0xff,0x3e,0xed, ...};
3     void encrypt(uint8_t *block){
4       for (int idx = 0; idx < 64; idx++){
5         block[idx] |= SBOX1[block[idx]];
6         block[idx] ^= SBOX2[block[idx]];
7       }
8     }
9
```

Figure 6.9: Example code for accessing S-Box lookup tables.

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

Also, a program may have multiple S-Box arrays, like `SBOX1` and `SBOX2` in Figure 6.9. Two successive accesses to `SBOX1` and `SBOX2` (at lines 5 and 6) cannot form a cache hit no matter what the lookup indices are. Therefore, SYMSC do not need to invoke the SMT solver to check the equivalence of these symbolic addresses. This can significantly cut down the constraint-solving time.

6.6.2 Layout-directed Reduction

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

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

6.7 Evaluation

I have implemented SYMSC using the LLVM compiler [82] and *Cloud9* [20] symbolic execution which builts upon KLEE [23]. SYMSC enhanced *Cloud9* in three aspects. First, it extended its support for multi-threading by allowing context switches prior to accessing global memory; the original *Cloud9* only allows context switches prior to executing a synchronization primitive (e.g.,

lock/unlock). Second, it made *Cloud9* fork new states to flip the execution order of two simultaneously enabled events when they may be mapped to the same cache line; the original *Cloud9* does not care about cache lines. Third, it let *Cloud9* record the address of each memory access along the execution, so it can incrementally build the cache-hit constraint. Based on these enhancements, I implemented the cache timing leak detector and optimized it for efficient constraint solving.

After compiling the C code of a program to LLVM bitcode, SYMSC executes it symbolically to generate interleavings according to Algorithm 8. The cache constraint at each memory access is expressed in standard KQuery expressions defined in KLEE [23]. By solving these constraints, SYMSC can obtain a concrete execution that showcases the leak, including a thread schedule, two input values $\overline{k_1}, \overline{k_2}$ and the adversarial memory address.

6.7.1 Subjects and Methodology

I evaluated SYMSC on a diverse set of open-source cipher programs. Specifically, the first group has five programs from a lightweight cryptographic system named *FELICS* [45], which was designed for resource-constrained devices. The second group has four programs from *Chronos* [42], a real-time Linux kernel. The third group has four programs from the GNU cryptographic library *Libgcrypt* [3], while the remaining programs are from the *LibTomCrypt* [2], the *OpenSSL* [4], and a recent publication [28]. They include multiple versions of several well-known algorithms such as `AES` [4, 42] and `DES` [3, 42], which are useful in evaluating the impact of cipher implementations on the performance of SYMSC.

Table 6.3 shows the statistics of these benchmark programs. The **LOC** and **LL** columns denote the lines of C code and the corresponding LLVM bit-code. The **KS** column shows the size of the sensitive input in bytes. The maximum number of memory accesses on program paths of each

Table 6.3: Benchmark statistics

Name	LOC	LL	KS	MA	Name	LOC	LL	KS	MA
AES[4]	1,429	4,384	24	771	FCrypt[42]	437	1,623	12	428
AES[42]	1,368	4,144	24	788	KV_name[28]	1,350	1,402	4	19
Camellia[2]	776	5,319	16	1,301	LBlock[45]	930	4,010	10	1,618
CAST5[2]	735	2,790	16	909	Misty1[1]	391	1,199	16	270
CAST5[42]	883	4,190	16	1,180	Piccolo[45]	301	1,034	12	350
Chaskey[45]	248	638	16	242	PRESENT[45]	194	272	10	94
DES[3]	596	2,166	8	963	rfc2268 [3]	388	870	16	149
DES[42]	1,010	3,926	8	1,029	Seed[3]	607	3,535	16	979
Kasumi[1]	350	1224	16	259	TWINE[45]	256	562	10	229
Khazad[42]	838	463	16	123	Twofish[3]	1,048	4,510	16	1,180

benchmark is shown in the **MA** column, which indicates the computational cost of the program.

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

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

The experiments were designed to answer two questions:

- Can SYMSC detect cache-timing leaks exposed by concurrently running a program with other threads?
- Are the optimizations in Section 6.6 effective in reducing the cost of symbolic execution and constraint solving?

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

6.7.2 Results Obtained with Fixed Addresses

Table 6.4: Results of leak detection with *fixed* addresses.

Name	Precise			Two-Step		
	#.Inter	#.Test	Time (m)	#.Inter	#.Test step1 / step2	Time (m)
AES[4]	57	55	430.2	57	55 / 55	140.3
AES[42]	1	0	288.9	1	1 / 0	41.4
Camellia[2]	1	0	0.1	1	1 / 0	0.1
CAST5[2]	1	0	0.1	1	1 / 0	0.1
CAST5[42]	1	0	0.1	1	1 / 0	0.1
Chaskey[45]	1	0	0.1	1	1 / 0	0.1
DES[3]	16	15	7.8	16	16 / 15	3.5
DES[42]	1	0	0.1	1	1 / 0	0.1
FCrypt[42]	16	15	4.1	16	15 / 15	8.1
Kasumi[1]	1	0	0.1	1	1 / 0	0.2
Khazad[42]	25	23	206.5	25	23 / 23	83.0
KV_Name[28]	1406	0	0.5	1406	1406 / 0	0.4
LBlock[45]	1	0	0.1	1	1 / 0	0.1
Misty1[1]	1	0	0.1	1	1 / 0	0.1
Piccolo[45]	1	0	0.1	1	1 / 0	0.1
PRESENT[45]	1	0	0.1	1	1 / 0	0.1
rfc2268[3]	1	0	0.1	1	1 / 0	0.1
Seed[3]	1	0	0.1	1	1 / 0	0.1
TWINE[45]	1	0	0.1	1	1 / 0	0.1
Twofish[3]	1	0	0.1	1	1 / 0	0.2

Table 6.4 shows the results obtained using fixed addresses in the cache layout (Case 1 in Section 6.2.3). Column 1 shows the benchmark name. Columns 2–4 present the result of computing the precise solution for our cache analysis problem. The last three columns show the result of running the simplified, two-step version, where the solution for $\exists k_1, k_2 \,.\, \tau(k_1) \neq \tau(k_2)$ is computed in two steps, by first computing a value of k_1 and then computing a value of k_2. In each method, the result table shows the number of interleavings explored (#.Inter), the number of leaky memory accesses detected (#.Test), and the execution time in minutes (m). For the two-step approach, we

also show the number of leakage points detected after the first step and after the second step.

Among these twenty programs, SYMSC detected leakage points in four: `ASE` from *OpenSSL* [4], `DES` from *Libgcrypt* [3], `FCrypt` from *Chronos* [42], and `Khazad` from *Chronos* [42]. I manually inspected these four programs in a way similar to what is described in Section 6.3.2, and confirmed that all these leakage points are realistic. Furthermore, the two-step approach returned exactly the same results as the precise analysis for all benchmark programs, but in significantly less time.

I also conducted our experiments using *4-way set-associative* cache instead of *direct-mapped* cache. The results of these experiments are similar to the ones reported in Table 6.4. Therefore, I omit them for brevity.

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

6.7.3 Results Obtained with Symbolic Addresses

The results shown in Table 6.4 are useful, but also somewhat *conservative*. A more aggressive analysis is to assume the adversarial thread T_2 may access memory regions whose cache layout is symbolic (refer to Case 2 in Section 6.2.3).

Table 6.5 shows the experimental results obtained using direct-mapped cache and symbolic ad-

Table 6.5: Results of leak detection with *symbolic* addresses.

Name	**#.Acc**	**Precise**			**Two-Step**		
		#.Inter	#.Test	Time(m)	#.Inter	#.Test step1 / step2	Time(m)
AES [4]	1,026	224	220	1016.4	224	220 / 220	237.5
AES[42]	2,568	141	139	>1600	256	302 / 254	548.3
Camellia[2]	2,590	176	172	830.8	176	172 / 172	303.5
CAST5[2]	1,815	167	164	>1600	384	381 / 381	1337.4
CAST5[42]	1,392	183	180	>1600	384	381 / 381	1392.5
Chaskey[45]	1,380	1	0	0.1	1	1 / 0	0.1
DES[3]	2,135	144	127	38.6	144	164 / 127	27.2
DES[42]	2,539	119	114	>1600	194	187 / 183	1191.5
FCrypt[42]	428	64	60	15.1	64	60 / 60	20.1
Kasumi[1]	1,785	83	82	>1600	96	94 / 94	151.9
Khazad[42]	684	114	103	>1600	248	254 / 240	165.3
KV_Name[28]	140	1406	0	0.5	1406	1406 / 0	0.5
LBlock[45]	4,068	1	0	0.1	1	1 / 0	0.1
Misty1[1]	2,966	76	75	>1600	96	94 / 94	265.1
Piccolo[45]	5,103	1	0	0.1	1	1 / 0	0.1
PRESENT[45]	8,233	1	0	0.2	1	1 / 0	0.2
rfc2268[3]	3,190	113	112	303.4	113	112 / 112	42.9
Seed[3]	1,632	201	197	>1600	320	316 / 316	1505.1
TWINE[45]	10,492	1	0	0.1	1	1 / 0	0.1
Twofish[3]	12,400	2514	84	>1600	900	84,063 / 76	>1600

dresses in thread T_2 (Case 2 in Section 6.2.3). The first two columns show the benchmark name and the maximum number of memory addresses accessed by an interleaving at run time. The *Precise* column shows the result of computing the precise solution for our cache analysis problem. The *Two-Step* column shows the result of running the simplified version. In both cases, SYMSC reports the total number of interleavings explored by symbolic execution (#.Inter), the number of leaky memory accesses detected (#.Test), and the total execution time in minutes (m). For *Two-Step*, the number of leaky accesses is further divided into two subcolumns: the leaky accesses detected after the first step and the leaky accesses detected after the second step.

The results show that, for most of the benchmark programs, the overhead of precisely solving our cache analysis is too high: on nine of the twenty programs, it could not complete within the time limit. In contrast, the two-step analysis was able to complete nineteen out of the twenty programs. In terms of accuracy, the two-step approach is almost as good as precise analysis: in all completed

programs, they detected the same number of leakage points, which indicate a possible combination of adversarial threads and memory layout that *can* trigger timing leaks.

The results also show that, for the same type of cryptographic algorithms (such as AES), different implementations may lead to drastically different overhead. For example, SYMSC detected 34 more leakage points in the `AES` implementation of *Chronos* [42] than that of *OpenSSL* [4]. However, the `AES` of *Chronos* took almost twice as long for our tool to analyze. For `DES` implementations from *Libgcrypt* [3] and *Chronos* [42], SYMSC detected a slightly different number of leakage points, but the time taken is significantly different (27.1 minutes versus 1191.5 minutes). In contrast, for the two versions of `CAST5`, SYMSC detected the same number of leakage points in roughly the same amount of time. For the benchmark where *Two-Step* took a long time, I found it is due to the increasing size of symbolic constraints which consist of the addresses in S-Box accesses. Typically the later a S-Box access in a loop, the larger its symbolic address expression would be. In `Twofish`, SYMSC timed out because of the large number of "may-be-related" event pairs (i.e., accessing the same S-Box but not the same cache line), which made SMT solving difficult.

6.7.4 Discussion

Based on the results, the two research questions have clear answers. First, SYMSC is able to identify cache timing leaks in concurrent programs. Specifically, using *symbolic* addresses in the adversarial thread demonstrates the possibility of triggering leaks in a concurrent system, whereas using *fixed* addresses in the analysis allows shows that such leaks are more practical. Second, SYMSC's optimization techniques are effective in reducing the computational overhead, which is demonstrated on a diverse set of real-world cipher programs.

SYMSC searches for sensitive inputs as well as an interleaving schedule that, together, trigger

divergent cache behaviors. If an individual program path has a constant cache behavior, e.g., all the memory accesses refer to fixed memory addresses regardless of the value of the sensitive input, then timing leaks are impossible. By checking for and leveraging such conditions, SYMSC can reduce the computation cost even further. For instance, with naive exploration, SYMSC would have generated 1,406 interleavings for the benchmark program named `KV_name`. However, with the above analysis, it does not have to generate any interleaving. In this example, `KV_name`'s 4-byte symbolic input only affects the branch conditions but does not taint any memory access address. Thus, many paths are explored by symbolic execution. However, no leak is detected on these paths.

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

6.8 Conclusion

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

Chapter 7

Conclusions

In this dissertation I first introduce the background of my research works and then establish the fundamental concepts in Chapter 1 and Chapter 2, respectively.

From Chapter 3 to Chapter 6 I describe in detail the motivations, key algorithms, implementations, and evaluations of my main contributions — four new symbolic execution based methods for analyzing various concurrent software systems . These methods cover the testing and verification of standard multithreading semantics, non-standard concurrency semantics and program non-functional properties.

Specifically, the *Assertion Guided Symbolic Execution* is a new state space reduction technique for eliminating redundant program paths and thread interleavings during symbolic execution; the *Incremental Symbolic Execution* is a method for generating test inputs by exploring only new behaviors introduced by code changes between two program versions; the SYMPLC is a method for systematically testing PLC software written in industrial languages specified by the IEC 61131-3 standard; and the *Adversarial Symbolic Execution* is a method that analyzes the timing character-

istics of concurrent program executions detect cache timing side-channel leaks.

All of these aforementioned works have been published as peer-reviewed papers in mainstream *Software Engineering* conferences. I believe that these works have been making considerable contributions to the crossing research directions of symbolic execution and concurrency.

Bibliography

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

[2] *LibTomCrypt*. http://www.libtom.net/LibTomCrypt/, .

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

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

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

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

[7] Borja Fernandez Adiego, Dániel Darvas, Enrique Blanco Viñuela, Jean-Charles Tournier, Simon Bliudze, Jan Olaf Blech, and Víctor Manuel González Suárez. Applying model checking to industrial-sized PLC programs. *IEEE Trans. Industrial Informatics*, 11(6):1400–1410, 2015.

[8] Vikram Adve, Chris Lattner, Michael Brukman, Anand Shukla, and Brian Gaeke. LLVM: A low-level virtual instruction set architecture. In *ACM/IEEE international symposium on Microarchitecture*, San Diego, California, Dec 2003.

[9] Timos Antonopoulos, Paul Gazzillo, Michael Hicks, Eric Koskinen, Tachio Terauchi, and Shiyi Wei. Decomposition instead of self-composition for proving the absence of timing

channels. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 362–375, 2017.

[10] John D. Backes, Suzette Person, Neha Rungta, and Oksana Tkachuk. Regression verification using impact summaries. In *International SPIN Workshop on Model Checking Software*, pages 99–116, 2013.

[11] Thomas Ball. A theory of predicate-complete test coverage and generation. In *Formal Methods for Components and Objects, Third International Symposium, FMCO 2004, Leiden, The Netherlands, November 2 - 5, 2004, Revised Lectures*, pages 1–22, 2004.

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

[13] Tiyash Basu and Sudipta 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.

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

[15] Sebastian Biallas, Jörg Brauer, and Stefan Kowalewski. Counterexample-guided abstraction refinement for PLCs. In *International Workshop on Systems Software Verification*, 2010.

[16] Sebastian Biallas, Mirco Giacobbe, and Stefan Kowalewski. Predicate abstraction for programmable logic controllers. In *Formal Methods for Industrial Critical Systems - 18th International Workshop*, pages 123–138, 2013.

[17] Peter Boonstoppel, Cristian Cadar, and Dawson R. Engler. RWset: Attacking path explosion in constraint-based test generation. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 351–366, 2008.

[18] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, OOPSLA '09, pages 243–262, New York, NY, USA, 2009. ACM.

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

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

[21] Jacob Burnim and Koushik Sen. Heuristics for scalable dynamic test generation. In *ASE*, pages 443–446, 2008.

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

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

[24] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Pasareanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic execution for software testing in practice:

Preliminary assessment. In *International Conference on Software Engineering*, pages 1066–1071, 2011.

[25] Henrik Carlsson, Bo Svensson, Fredrik Danielsson, and Bengt Lennartson. Methods for reliable simulation-based PLC code verification. *IEEE Trans. Industrial Informatics*, 8(2): 267–278, 2012.

[26] Sagar Chaki, Arie Gurfinkel, and Ofer Strichman. Time-bounded analysis of real-time systems. In *International Conference on Formal Methods in Computer-Aided Design*, pages 72–80, 2011.

[27] Sagar Chaki, Arie Gurfinkel, and Nishant Sinha. Efficient verification of periodic programs using sequential consistency and snapshots. In *International Conference on Formal Methods in Computer-Aided Design*, pages 51–58, 2014.

[28] Sudipta 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.

[29] Sudipta Chattopadhyay, Moritz Beck, Ahmed Rezine, and Andreas Zeller. Quantifying the information leak in cache attacks through symbolic execution. *CoRR*, abs/1611.04426, 2016.

[30] Sudipta Chattopadhyay, Moritz Beck, Ahmed Rezine, and Andreas 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.

[31] Gang chen, Xiaoyu Song, and Ming Gu. PLC program verification and analysis using the

coq theorem prover. *Acta Scientiarum Naturalium Universitatis Pekinensis*, 46(1):30–34, 2010.

[32] Jia Chen, Yu Feng, and Isil 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.

[33] Chih-Hong Cheng, Chung-Hao Huang, Harald Ruess, and Stefan Stattelmann. G4LTL-ST: automatic generation of PLC programs. In *International Conference on Computer Aided Verification*, pages 541–549, 2014.

[34] Chih-Hong Cheng, Yassine Hamza, and Harald Ruess. Structural synthesis for GXW specifications. In *International Conference on Computer Aided Verification*, pages 95–117, 2016.

[35] Ankit Choudhary, Shan Lu, and Michael Pradel. Efficient detection of thread safety violations via coverage-guided generation of concurrent tests. In *International Conference on Software Engineering*, pages 266–277, 2017.

[36] Duc-Hiep Chu and Joxan Jaffar. A framework to synergize partial order reduction with state interpolation. In *International Haifa Verification Conference*, pages 171–187, 2014.

[37] Duc-Hiep Chu, Joxan Jaffar, and Rasool 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.

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

[39] Lori A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Software Eng.*, 2(3):215–222, 1976.

[40] Dániel Darvas, István Majzik, and Enrique Blanco Viñuela. Formal verification of safety PLC based control software. In *Integrated Formal Methods - 12th International Conference*, pages 508–522, 2016.

[41] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 337–340, 2008.

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

[43] Jean-François Dhem, François Koeune, Philippe-Alexandre Leroux, Patrick Mestré, Jean-Jacques Quisquater, and Jean-Louis Willems. A practical implementation of the timing attack. In *Smart Card Research and Applications, This International Conference, CARDIS '98, Louvain-la-Neuve, Belgium, September 14-16, 1998, Proceedings*, pages 167–182, 1998.

[44] E. Dijkstra. *A Discipline of Programming*. Prentice Hall, NJ, 1976.

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

[46] Goran Doychev, Dominik Feld, Boris Köpf, Laurent Mauborgne, and Jan 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.

[47] Jean-Marie Farines, Max Hering de Queiroz, Vinicius G. da Rocha, Ana Maria M. Carpes, François Vernadat, and Xavier Crégut. A model-driven engineering approach to formal

verification of PLC programs. In *IEEE Conference on Emerging Technologies & Factory Automation*, pages 1–8, 2011.

[48] Azadeh Farzan, Andreas Holzer, Niloofar Razavi, and Helmut Veith. Con2colic testing. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 37–47, 2013.

[49] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

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

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

[52] Haojie Fu, Zan Wang, Xiang Chen, and Xiangyu Fan. A systematic survey on automated concurrency bug detection, exposing, avoidance, and fixing techniques. *Software Quality Journal*, 26(3):855–889, 2018.

[53] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic analysis: Concrete results. In *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*, number Generators, pages 251–261, 2001.

[54] Daniel Genkin, Adi Shamir, and Eran Tromer. RSA key extraction via low-bandwidth acoustic cryptanalysis. In *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I*, pages 444–461, 2014.

[55] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*. Springer, 1996. ISBN 3-540-60761-7.

[56] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Programming Language Design and Implementation*, pages 213–223, June 2005.

[57] Patrice Godefroid. Compositional dynamic test generation. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 47–54, 2007.

[58] Patrice Godefroid. Software model checking improving security of a billion computers. In *International SPIN Workshop on Model Checking Software*, page 1, 2009.

[59] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *Network and Distributed System Security Symposium*, 2008.

[60] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*, pages 72–83, 1997.

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

[62] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *Software Tools for Technology Transfer*, 2(4), 2000.

[63] Kim Herzig, Michaela Greiler, Jacek Czerwonka, and Brendan Murphy. The art of testing less without sacrificing quality. In *International Conference on Software Engineering*, ICSE '15, pages 483–493, Piscataway, NJ, USA, 2015. IEEE Press.

[64] Krystof Hoder, Nikolaj Bjørner, and Leonardo de Moura. muZ - an efficient engine for fixed points with constraints. In *International Conference on Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 457–462, 2011.

[65] G. Holzmann, E. Najm, and A. Serhrouchni. SPIN model checking: An introduction. *International Journal on Software Tools for Technology Transfer*, 2(4):321–327, 2000.

[66] Shin Hong, Jaemin Ahn, Sangmin Park, Moonzoo Kim, and Mary Jean Harrold. Testing concurrent programs to achieve high synchronization coverage. In *International Symposium on Software Testing and Analysis*, pages 210–220, 2012.

[67] Shin Hong, Matt Staats, Jaemin Ahn, Moonzoo Kim, and Gregg Rothermel. The impact of concurrent coverage metrics on testing effectiveness. In *IEEE International Conference on Software Testing, Verification and Validation*, pages 232–241, 2013.

[68] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, January 1990.

[69] Joxan Jaffar, Vijayaraghavan Murali, and Jorge A. Navas. Boosting concolic testing via interpolation. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 48–58, 2013.

[70] Vilas Jagannath, Qingzhou Luo, and Darko Marinov. Change-aware preemption prioritization. In *International Symposium on Software Testing and Analysis*, pages 133–143, 2011.

[71] Karl-Heinz John and Michael Tiegelkamp. *IEC 61131-3: Programming Industrial Automation Systems: Concepts and Programming Languages, Requirements for Programming Systems, Decision-making Aids*. Springer Science & Business Media, 2010.

[72] Kari Kähkönen and Keijo Heljanko. Testing multithreaded programs with contextual un-

foldings and dynamic symbolic execution. In *International Conference on Application of Concurrency to System Design*, pages 142–151, 2014.

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

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

[75] Paul 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.

[76] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, pages 388–397, 1999.

[77] Lock-Jo Koo, Chang Mok Park, Chang Ho Lee, SangChul Park, and Gi-Nam Wang. Simulation framework for the verification of PLC programs in automobile industries. *International Journal of Production Research*, 49(16):4925–4943, 2011.

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

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

[80] E. V. Kuzmin, Valery A. Sokolov, and D. A. Ryabukhin. Construction and verification of PLC-programs by LTL-specification. *Automatic Control and Computer Sciences*, 49(7): 453–465, 2015.

[81] Shuvendu K. Lahiri, Kenneth L. McMillan, Rahul Sharma, and Chris Hawblitzel. Differential assertion checking. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 345–355, 2013.

[82] Chris Lattner and Vikram Adve. The LLVM Compiler Framework and Infrastructure Tutorial. In *LCPC'04 Mini Workshop on Compiler Research Infrastructures*, West Lafayette, Indiana, Sep 2004.

[83] Edward A. Lee. The problem with threads. *IEEE Computer*, 39(5):33–42, 2006.

[84] Steffen Lehnert. A taxonomy for software change impact analysis. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution*, pages 41–50, 2011.

[85] You Li, Zhendong Su, Linzhang Wang, and Xuandong Li. Steering symbolic execution to less traveled paths. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 19–32, 2013.

[86] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks - Revealing the Secrets of Smart Cards*. Springer, 2007.

[87] MATIEC. IEC 61131-3 compiler. URL: https://www.openhub.net/p/matiec.

[88] A. W. Mazurkiewicz. Trace theory. In *Advances in Petri Nets*, pages 279–324. Springer, 1986.

[89] Kenneth L. McMillan. Lazy annotation for program testing and verification. In *International Conference on Computer Aided Verification*, pages 104–118, 2010.

[90] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, January 2005.

[91] Mayur Naik and Alex Aiken. Conditional must not aliasing for static race detection. *SIGPLAN Not.*, 42(1):327–338, January 2007.

[92] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for java. *SIGPLAN Not.*, 41(6):308–319, June 2006.

[93] NBDS. Non-blocking data structures. URL: https://code.google.com/p/nbds/.

[94] NED. ned productions: nedmalloc URL: http://www.nedprod.com/programs/portable/nedmalloc/.

[95] Johanna Nellen, Erika Ábrahám, and Benedikt Wolters. A CEGAR tool for the reachability analysis of PLC-controlled plants using hybrid automata. In *Formalisms for Reuse and Systems Integration*, pages 55–78. 2015.

[96] Johanna Nellen, Kai Driessen, Martin Neuhäußer, Erika Ábrahám, and Benedikt Wolters. Two CEGAR-based approaches for the safety verification of PLC-controlled plants. *Information Systems Frontiers*, pages 1–26, 2016.

[97] Sang C. Park, Chang Mok Park, Gi-Nam Wang, Jonggeun Kwak, and Sungjoo Yeo. PLCStudio: Simulation based PLC code verification. In *Proceedings of the 2008 Winter Simulation Conference*, pages 222–228, 2008.

[98] Corina S. Pasareanu, Neha Rungta, and Willem Visser. Symbolic execution with mixed concrete-symbolic solving. In *International Symposium on Software Testing and Analysis*, pages 34–44, 2011.

[99] Corina S. Pasareanu, Quoc-Sang Phan, and Pasquale Malacaria. Multi-run side-channel analysis using symbolic execution and max-smt. In *IEEE 29th Computer Security Foun-*

dations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016, pages 387–400, 2016.

[100] Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. Directed incremental symbolic execution. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 504–515, 2011.

[101] Quoc-Sang Phan, Lucas Bang, Corina S. Pasareanu, Pasquale Malacaria, and Tevfik Bultan. Synthesis of adaptive side-channel attacks. *IACR Cryptology ePrint Archive*, 2017:401, 2017.

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

[103] Shruti Raghavan, Rosanne Rohana, David Leon, Andy Podgurski, and Vinay Augustine. Dex: A semantic-graph differencing tool for studying changes in large code bases. In *IEEE International Conference on Software Maintenance*, pages 188–197, 2004.

[104] Niloofar Razavi, Franjo Ivancic, Vineet Kahlon, and Aarti Gupta. Concurrent test generation using concolic multi-trace analysis. In *Asian Symposium on Programming Languages and Systems*, pages 239–255, 2012.

[105] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Points-to analysis for java using annotated constraints. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, pages 43–55, New York, NY, USA, 2001. ACM.

[106] Neha Rungta, Suzette Person, and Joshua Branchaud. A change impact analysis to charac-

terize evolving program behaviors. In *IEEE International Conference on Software Maintenance*, pages 109–118, 2012.

[107] K. Sen. *Scalable Automated Methods for Dynamic Program Analysis*. PhD thesis, UIUC, 2006.

[108] K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *International Conference on Computer Aided Verification*, pages 419–423. Springer, 2006.

[109] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 263–272, 2005.

[110] Koushik Sen, Swaroop Kalasapur, Tasneem G. Brutch, and Simon Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for javascript. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 488–498, 2013.

[111] Koushik Sen, George C. Necula, Liang Gong, and Wontae Choi. Multise: Multi-path symbolic execution using value summaries. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 842–853, 2015.

[112] Junaid Haroon Siddiqui and Sarfraz Khurshid. Scaling symbolic execution using ranged analysis. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 523–536, 2012.

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

[114] Herb Sutter and James R. Larus. Software and the concurrency revolution. *ACM Queue*, 3 (7):54–62, 2005.

[115] SV-COMP. 2014 software verification competition. URL: http://sv-comp.sosy-lab.org/2014/, 2014.

[116] SV-COMP. 2015 software verification competition. URL: http://sv-comp.sosy-lab.org/2015/, 2015.

[117] Valerio Terragni, Shing-Chi Cheung, and Charles Zhang. RECONTEST: effective regression testing of concurrent programs. In *International Conference on Software Engineering*, pages 246–256, 2015.

[118] Nikolai Tillmann and Jonathan de Halleux. PEX – white box test generation for .NET. In *International Conference on Tests and Proofs*, pages 134–153, 2008.

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

[120] David Trabish, Andrea Mattavelli, Noam Rinetzky, and Cristian Cadar. Chopped symbolic execution. In *International Conference on Software Engineering*, pages 350–360, 2018.

[121] Willem Visser, Corina S. Pasareanu, and Radek Pelánek. Test input generation for java containers using state matching. In *International Symposium on Software Testing and Analysis*, pages 37–48, 2006.

[122] Bjoern Wachter, Daniel Kroening, and Joel Ouaknine. Verifying multi-threaded software with Impact. In *International Conference on Formal Methods in Computer-Aided Design*, pages 210–217, 2013.

[123] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. Cached: Identifying cache-based timing channels in production software. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 235–252, 2017.

[124] Mark Weiser. Program slicing. In *International Conference on Software Engineering*, ICSE, pages 439–449, 1981.

[125] Jerod W. Wilkerson. A software change impact analysis taxonomy. In *IEEE International Conference on Software Maintenance*, pages 625–628, 2012.

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

[127] Guowei Yang, Sarfraz Khurshid, Suzette Person, and Neha Rungta. Property differencing for incremental checking. In *International Conference on Software Engineering*, pages 1059–1070, 2014.

[128] Wuu Yang. Identifying syntactic differences between two programs. *Softw., Pract. Exper.*, 21(7):739–755, 1991.

[129] Y. Yang, X. Chen, G. Gopalakrishnan, and R. Kirby. Efficient stateful dynamic partial order reduction. In *SPIN Workshop on Model Checking Software*, pages 288–305, 2008.

[130] Qiuping Yi, Zijiang Yang, Shengjian Guo, Chao Wang, Jian Liu, and Chen Zhao. Post-conditioned symbolic execution. In *IEEE International Conference on Software Testing, Verification and Validation*, pages 1–10, 2015.

[131] Jie Yu, Satish Narayanasamy, Cristiano Pereira, and Gilles Pokam. Maple: A coverage-driven testing tool for multithreaded programs. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 485–502, 2012.

[132] Tingting Yu. TACO: test suite augmentation for concurrent programs. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 918–921, 2015.

[133] Tingting Yu, Witawas Srisa-an, and Gregg Rothermel. Simrt: An automated framework to support regression testing for data races. In *International Conference on Software Engineering*, pages 48–59. ACM, 2014.