Eliminating Path Redundancy via Postconditioned Symbolic Execution

Qiuping Yi, Zijiang Yang, Senior Member, IEEE, Shengjian Guo, Member, IEEE, Chao Wang, Member, IEEE, Jian Liu, Member, IEEE and Chen Zhao

Abstract—Symbolic execution is emerging as a powerful technique for generating test inputs systematically to achieve exhaustive path coverage of a bounded depth. However, its practical use is often limited by path explosion because the number of paths of a program can be exponential in the number of branch conditions encountered during the execution. To mitigate the path explosion problem, we propose a new redundancy removal method called postconditioned symbolic execution. At each branching location, in addition to determine whether a particular branch is feasible as in traditional symbolic execution, our approach checks whether the branch is subsumed by previous explorations. This is enabled by summarizing previously explored paths by weakest pre-condition computations. Postconditioned symbolic execution can identify path suffixes shared by multiple runs and eliminate them during test generation when they are redundant. Pruning away such redundant paths can lead to a potentially exponential reduction in the number of explored paths. Since the new approach is computationally expensive, we also propose several heuristics to reduce its cost. We have implemented our method in the symbolic execution engine KLEE [1] and conducted experiments on a large set of programs from the GNU Coreutils suite. Our results confirm that redundancy due to common path suffix is both abundant and widespread in real-world applications.

1 INTRODUCTION

Dynamic symbolic execution based test input generation has emerged as a popular technique for testing real-world applications written in full-fledged programming languages such as C/C++ and Java [2], [3], [4], [5], [6], [1]. The method performs concrete analysis as well as symbolic analysis of the program simultaneously, often in an execution environment that accurately models the system calls and external libraries. The symbolic analysis is conducted by analyzing each execution path precisely, i.e., encoding the path condition as a quantifier-free first-order logic formula and then deciding the formula with a SAT or SMT solver. When a path condition is satisfiable, the solver returns a test input that can steer the program execution along this path. Due to its capability of handling real applications in their native execution environments, dynamic symbolic execution has been quite successful in practical settings—for a survey of the recent tools, see Pasareanu et al. [7].

However, a major hurdle that prevents symbolic execution from getting even wider application is path explosion. That is, the number of paths of a program can be exponential in the number of branch conditions encountered during the execution. Even for a medium-size program and a small bound for the execution depth, exhaustively covering all possible paths can be extremely expensive. Consider the program in Figure 1, which has three input variables $a$, $b$, $c$ and three consecutive if-else statements. Dynamic symbolic execution is able to compute a set of test inputs, each of which has concrete values for all input variables, to exhaustively cover the valid execution paths of the program. There are $2^3 = 8$ distinct execution paths for this program. Classic symbolic execution tools such as KLEE [1] would generate eight test inputs. The covered paths of this example, numbered from 1 to 8, are shown in Figure 1 (right). For instance, P1 is a path that passes through the if-branch at Line 1, the if-branch at Line 3, and the if-branch at Line 5.

Many efforts have been made to mitigate the path explosion problem. One of them, which has been quite effective in practice, is called preconditioned symbolic execution [8]. It achieves path reduction by taking a predefined constraint $\Pi_{pre}$ as an additional parameter in addition to the program under test. Preconditioned symbolic execution only descends into program branches that satisfy $\Pi_{pre}$, with the net effect of pruning away the subsequent steps of unsatisfied branches. By leveraging the constraint, preconditioned symbolic execution effectively reduces the search space. That is, the reduction is achieved by sacrificing the completeness of symbolic execution. For example, by setting $\Pi_{pre}$ to be false preconditioned symbolic execution terminates without
exploring any path.

In this paper, with the assumption that all potential failures are modeled as conditional abort statements, we propose a new method called postconditioned symbolic execution that eliminates redundant paths without reducing the search space. The new approach mitigates path explosion by identifying and then eliminating redundant path suffixes encountered during symbolic execution. Our method is based on the observation that many path suffixes are shared among different test runs, and repeatedly exploring these path suffixes is a main reason for path explosion. Consider Figure 1 again. Although the eight paths in Figure 1 are different, they share common path suffixes. For instance, the suffix ... → 3 → 5 is shared by paths No. 1 and No. 5; and the suffix ... → 6 is shared by paths No. 2, No. 4, No. 6, and No. 8. Of course, sharing a suffix does not necessary mean the suffix needs not be explored again. Otherwise path exploration is erroneously reduced to branch coverage. On the other hand, since the goal of testing is to uncover bugs, once a path suffix will not reveal any new program behavior, we should not explore it again in the future.

In order to avoid unnecessary exploration, postconditioned symbolic execution associates each program location \( l \) with a postcondition that summarizes the explored path suffixes starting from \( l \). During the iterative test generation process, new path suffixes are characterized and added incrementally to a postcondition, represented as a quantifier-free first-order logic constraint. In the subsequent computation of new test inputs, our method checks whether the current path condition is subsumed by the postcondition. If the answer is yes, the execution of the rest of the path is skipped.

There are major differences between preconditioned and postconditioned symbolic executions. The constraint of the former approach is predefined, while the constraints of the latter are dynamically computed. The goal of preconditioned symbolic execution is to avoid the paths that do not satisfy the predefined constraint, and thus there is no guarantee of exhaustive path coverage. In fact, if the predefined constraint is false, no path will be explored. It highlights the fact that the predefined constraint has to be carefully chosen, or else it will not be effective. In contrast, postconditioned symbolic execution has a path coverage that is equivalent to standard symbolic execution, because the dynamically computed postconditions eliminate redundant paths only.

We have implemented a software tool based on the KLEE symbolic virtual machine [1] and evaluated it using a large set of 5 programs from the GNU Coreutils suite, which implements some of the most frequently used Unix/Linux commands. These benchmarks can be considered as representatives of the systems code in Unix/Linux. They are challenging for symbolic execution due to the extensive use of error checking code, loops, pointers, and heap allocated data structures. Nevertheless, our experiments show that postconditioned symbolic execution can have a significant speedup over state-of-the-art methods in KLEE on these benchmarks.

To sum up, our main contributions are listed as follows:

- We propose a new symbolic execution method for identifying and eliminating redundant path suffixes to mitigate the path explosion problem.
- Postconditioned symbolic execution is computationally expensive. We propose several optimization heuristics to reduce its cost.
- We implement a prototype software tool based on KLEE [1] and experimentally compare our new method with the state-of-the-art techniques.
- We confirm, through our experimental analysis of real-world applications, that redundancy due to common path suffixes is both abundant and widespread, and our new method is effective in reducing the number of explored paths as well as the execution time.

The remainder of this paper is organized as follows. We first establish notations and review related techniques in Section 2. Then, we present our postconditioned symbolic execution method in Section 3 and its several optimizations in Section 4, followed by experimental results in Section 5. We review related work in Section 6, and finally give our conclusions in Section 7.

2 REVIEW OF TEST GENERATION USING SYMBOLIC EXECUTION

We first present background definitions related to the syntax and semantics for a program. We consider that...
a program $P$ has a set $Var$ of program variables and a set $Instr$ of instructions with the following syntactic categories:

$$
\begin{align*}
\text{e}_a & \in AExp \quad \text{arithmetic expressions} \\
\text{e}_b & \in BExp \quad \text{boolean expressions} \\
\text{instr} & \in Instr \quad \text{instructions}
\end{align*}
$$

We assume some countable set of variables is given; constants will not be further defined and neither will be the operators.

$$
\begin{align*}
a, b, c & \in Var \quad \text{variables} \\
n & \in Con \quad \text{constants} \\
op_a & \in Op_a \quad \text{arithmetic, pointer etc. operators} \\
op_b & \in Op_b \quad \text{boolean operators} \\
op_r & \in Op_r \quad \text{relational operators}
\end{align*}
$$

The syntax of a program $P$ is given by the following abstract syntax:

$$
\begin{align*}
\text{e}_a & ::= \text{a} \mid \text{n} \mid \text{e}_a \text{op}_a \text{e}_a \\
\text{e}_b & ::= \text{true} \mid \text{false} \mid \text{not} \text{e}_b \mid \text{e}_b \text{op}_b \text{e}_b \\
\text{instr} & ::= \text{a} \text{:=} \text{e}_a \mid \text{skip} \mid \text{halt} \mid \text{abort} \mid \text{instr}_1 ; \text{instr}_2 \\
& \mid \text{if} \text{e}_b \text{then instr}_1 \text{else instr}_2 \\
& \mid \text{while } \text{e}_b \text{do instr}
\end{align*}
$$

Given a program $P$, let $V_{in} \subseteq Var$ be the subset of input variables, which are marked in the program as symbolic, e.g., using char $x:=\text{symbolic}()$. The goal of test generation is to compute concrete values for these input variables such that the new test inputs, collectively, can cover all possible execution paths of the program.

We assume an active testing frameworks [9], where all detectable failures are modeled using a special abort instruction. The reachability of the abort statements indicates the occurrence of a runtime failure. For example, instruction assert($c$) can be modeled as if($!c$)abort;else skip, instruction $x = y/z$ can be modeled as if($z == 0$)abort;else $x = y/z$, and instruction $t -> k = 5$ can be modeled as if($t == 0$)abort;else $t -> k = 5$. Consider that abort may appear anywhere in a path, in general, detecting the failures requires the effective coverage of all valid execution paths. An instruction instr may have one of the following types:

- skip, representing a dummy statement. It will often be used for omitting the else branches.
- halt, representing the normal program termination;
- abort, representing the faulty program termination;
- assignment $v := exp$, where $v \in Var$ and $exp$ is an expression over the set $Var$ of program variables;
- branch $if(c)$, where $c$ is a conditional expression over $Var$ and $if(c)$ represents the branch taken by the execution. The else-branch is represented by else, which equals to $if(!c)$.

With proper code transformations, the above instruction types are sufficient to represent the execution path of arbitrary C code. For example, if $\text{ptr}$ points to $\&a$, $*p := 5$ can be modeled as if($p == \&a$) $a := 5$. And if $q$ points to $\&b$, $q -> x := 10$ can be modeled as if($q == \&b$) $b.x := 10$.

For a complete treatment of all instruction types, please refer to the original dynamic symbolic execution papers on DART [2], CUTE [4], or KLEE [1].

**Definition 1. Concretes memory state.** Let $Val$ be the value space. A concrete memory state is a mapping $mem_c : Var \rightarrow Val$. For each program variable $v \in Var$, its concrete value is represented by the expression $mem_c[v]$.

**Definition 2. Concrete execution event.** For a concrete memory state $mem_c$, let instr be an instruction in the program. An execution instance of instr is called an event, denoted as $ev = (l, instr, l')$, where $l$ and $l'$ are the control locations before and after executing the instruction.

For a branch instruction $if(c)$, if $c$ is evaluated to true under $mem_c$, then the branch will be taken by the execution. Otherwise, the else-branch, represented by $if(!c)$, will be executed. A control location $l$ is a place in the execution path, not the location of the instruction (e.g., line number) in the program code. For example, if instr is executed multiple times in the same execution path, e.g., when instr is inside a loop or a recursive function, each instance of instr would give rise to a separate event, with unique control locations $l$ and $l'$. Conceptually, this corresponds to unwinding the loop or recursive calls.

**Definition 3. Concrete execution.** A concrete execution of a deterministic sequential program is a sequence of execution events $\pi = l_0 \xrightarrow{l_1} l_1 \xrightarrow{l_2} l_2 \ldots \xrightarrow{l_n} l_n$.

**Definition 4. Test input.** For a program $P$, let $V_{in} \subseteq Var$ be the subset of input variables. A test input of $P$ is a mapping $t: V_{in} \rightarrow Val$.

**Definition 5. Instruction reachable.** For a deterministic sequential program $P$, the concrete execution of $P$ is fully determined by test inputs. Let $t$ be a test input of $P$ and $\pi = l_0 \xrightarrow{l_1} l_1 \xrightarrow{l_2} l_2 \ldots \xrightarrow{l_n} l_n$ be the concrete execution determined by $t$. If instruction $s$ is in $\pi_i$, $(0 < i \leq n)$, we say that $s$ is reachable under test input $t$.

Specifically, if an abort is reachable under $t$, we say $t$ triggers an error. Moreover, for the concrete execution $\pi$, a suffix of $\pi$ is a subsequence $\pi^i = l_i \xrightarrow{l_{i+1}} \ldots \xrightarrow{l_n} l_n$, where $0 \leq i \leq n$. We also say $\pi^i$ is reachable under $t$. Let $T$ be a test input set of a program, we can easily extend the reachability of an instruction (or a suffix) under $T$. Given a test input $t \in T$, if an instruction $s$ (or suffix $\pi^i$) is reachable under $t$ then we say that $s$ (or suffix $\pi^i$) is reachable under $T$.

If we denote the concrete execution by the pair $(t, \pi)$, the corresponding symbolic execution is denoted $(*, \pi)$, where $*$ means that the test input is arbitrary. In a dynamic symbolic execution (also called concolic) framework [2], [1], [4], a symbolic execution will be calculated along with a concrete execution on-the-fly. The symbolic semantics for a concrete execution is defined as follows.

**Definition 6. Symbolic execution.** A symbolic execution
of a deterministic sequential program is sequence of instructions $\pi_s = l_0 \xrightarrow{\text{instr}_1} l_1 \xrightarrow{\text{instr}_2} l_2 \ldots \xrightarrow{\text{instr}_n} l_n$.

Definition 7. Symbolic memory state. Let $\pi_s = l_0 \xrightarrow{\text{instr}_1} l_1 \xrightarrow{\text{instr}_2} l_2 \ldots \xrightarrow{\text{instr}_n} l_n$ be a symbolic execution, there is symbolic memory state on each location. A symbolic memory state is a mapping $\text{mem}_s : \text{Var} \rightarrow A\text{Exp}$. For each program variable $v \in \text{Var}$, its symbolic value is represented by the expression $\text{mem}_s[v]$. Let $l \xrightarrow{\text{instr}} l'$ be an event in execution $\pi_s$, $\text{mem}_s$ be symbolic memory state before the event and $\text{mem}'_s$ be the following symbolic state. If $\text{instr}$ is $v := \text{exp}$, then $\text{mem}'_s = \text{mem}[v \leftarrow \text{exp}]$, otherwise $\text{mem}'_s = \text{mem}_s$.

Definition 8. Path condition. Let $\pi_s = l_0 \xrightarrow{\text{instr}_1} l_1 \xrightarrow{\text{instr}_2} l_2 \ldots \xrightarrow{\text{instr}_n} l_n$ be a symbolic execution, there is a path condition on each location. A path condition is boolean expression, denoted as $\text{pcon}$. Let $l \xrightarrow{\text{instr}} l'$ be an event in the execution $\pi_s$, $\text{pcon}$ be path condition before the event and $\text{pcon}'$ be the following one. If the $\text{instr}$ is $\text{if}(c)$, then $\text{pcon}' = \text{pcon} \land c$, otherwise $\text{pcon}' = \text{pcon}$.

The set of all possible symbolic executions of a program can be captured by a directed acyclic graph (DAG), where the nodes are control locations and the edges are instructions that move the program from one control location $l$ to another control location $l'$. The root node is the initial program state, and each terminal node represents the end of an execution. A non-terminal node $l$ may have one outgoing edge, which is of the form $l \xrightarrow{\text{instr}} l'$, or two outgoing edges, each of which is of the form $l \xrightarrow{\text{instr}} l'$. The goal of symbolic execution is to compute a set $T$ of test inputs such that, collectively, they cover all valid paths in the DAG.

Algorithm 1 shows the pseudocode of the classic symbolic execution procedure, e.g., the one implemented in KLEE. Given a program $P$ and an initial state, the procedure keeps discovering new program paths and generating new test inputs, with the goal of covering these paths. That is, if $\pi$ is a valid path of the program $P$ under some test input, it should be able to generate test input $\tau$ in $T$ that replays this path.

In this algorithm, a program state is represented by a tuple $(\text{pcon}, l, \text{mem})$, where $\text{pcon}$ is the path condition along an execution and $l$ is a control location and $\text{mem}$ is the symbolic memory map. Note that, for simplification, in the following sections we also use $\text{mem}$ to denote symbolic memory states. The initial state is $(\text{true}, l_{\text{init}}, \text{mem}_{\text{init}})$, meaning that the path condition $\text{pcon}$ is true and $l_{\text{init}}$ is the beginning of the program. We use stack to store the set of states that need to be processed by the symbolic execution procedure. Initially, stack contains the initial state only. Within the while-loop, for each state $(\text{pcon}, l, \text{mem})$ in the stack, we first find the successor state, when instr is an assignment, or the set of successor states, when the instructions are branches. For each successor state, we compute the new path condition $\text{pcon}'$ and the control location $l'$.

There are four types of events that can move the program from location $l$ to location $l'$.

- If the event is $l \xrightarrow{\text{abort}} l'$, the symbolic execution procedure finds a bug and terminates.
- If the event is $l \xrightarrow{\text{halt}} l'$, the symbolic execution path reaches the end.
- If the event is $l \xrightarrow{v:=\text{exp}} l'$, the new memory map $\text{mem}'$ is computed by assigning $\text{exp}$ to $v$ in $\text{mem}$, denoted $\text{mem}[v \leftarrow \text{exp}]$.
- If the event is $l \xrightarrow{\text{if}(c)} l'$, the new path condition $\text{pcon}'$ is computed by conjuncting $\text{pcon}$ with $c$, denoted $\text{pcon} \land c$.

Since we use a stack to hold the set of to-be-processed states, Algorithm 1 implements a depth-first search (DFS) strategy. That is, the procedure symbolically executes the first full path toward its end before executing the next paths. On a uniprocessor machine, the set of paths of a program would be executed sequentially, one after another. In addition to DFS, other frequently used search strategies include breadth-first search (BFS) and random search. These alternative strategies can be implemented by replacing the stack with a queue or some random-access data structures.

Although the classic symbolic execution procedure in Algorithm 1 can systematically generate test inputs that collectively cover the entire space of paths up to a certain depth, the number of paths (and hence test inputs) is often extremely large even for medium-size programs. Our observation is that, in practice, many program paths share common path suffixes (termed “diamond-shaped” paths). Since the goal of software testing is to uncover...
bugs, once a path suffix is tested, there is no need to generate new test inputs to cover the path suffix again in the future.

In the remainder of this paper, we shall present postconditioned symbolic execution that is able to identify and then eliminate such redundant path suffixes. In the best case scenario, removing such redundant paths can lead to an exponential reduction in the number of explored paths.

3 Postconditioned Symbolic Execution

3.1 A Motivating Example

Postconditioned Symbolic Execution (PSE) attempts to prune paths by eliminating redundant path suffixes encountered during symbolic execution. During the exploration, PSE maintains the same concrete and symbolic states as the standard symbolic execution does. Moreover, at each branch location $l$, a postcondition $\Pi_{post}[l]$ summarizes the path suffixes that have been explored from this location. Given a new path that reaches $l$ with path condition $\phi$, there is no need to continue beyond $l$ if the $\phi$ is subsumed by $\Pi_{post}[l]$. The subsumption indicates that no program states reached after $l$ along $\pi$ can cause abort. In order to obtain postconditions, we exploit weakest precondition computation. The postcondition at each branch location is updated after a new path exploration.

In this section, we illustrate the main idea behind PSE using an example. Using the information shown in Table 1, we first show how standard symbolic execution works. Then, we show how postconditioned symbolic execution works on the same example.

Columns 1-4 illustrate the process of running standard symbolic execution on the program in Figure 1. Column 1 shows the index of the path (numbered from 1 to 8). Column 2 shows the sequence of branches taken by the path. Column 3 shows the path condition accumulated by symbolic execution at each branch. Column 4 shows at which step the constraint solver is invoked to check the satisfiability (SAT) of the path condition, to compute the test input.

Columns 5-7, in contrast, illustrate our new method. Column 7 shows the summary of explored path suffixes, which is computed for each if-else statement, after the execution of this path terminates. In contrast to the original path condition $\phi$ shown in Column 3, the new path condition $\phi'$ in Column 5 is the conjunction of $\phi$ with the negated postcondition $-\Pi_{post}[l]$ at location $l$, where $\Pi_{post}[l]$ summarizes all the explored path suffixes from location $l$ and thus $\phi \land -\Pi_{post}[l]$ represents that the following path exploration will avoid the explored path suffixes. It is worth pointing out that the postcondition $\Pi_{post}[l]$ is computed at the end of the previous symbolic execution path.

For path No. 1, since the summary does not yet exist when we compute the path condition $\phi'$, we assume that $\Pi_{post}[l] = false$ for every location $l$. Therefore, the new path conditions at Lines 1, 3 and 5 remain the same; they are $(a \leq 0), (a \leq 0) \land (a + 10 \leq b)$ and $(a \leq 0) \land (a + 10 \leq b) \land (a + 10 - b > c)$, respectively. A test input such as $a = -10, b = 1, c = -6$ can be computed by solving the path condition $(a \leq 0) \land (a + 10 \leq b) \land (a + 10 - b > c)$ at Line 5.

At the end of Path No. 1, we summarize its path suffix by performing a weakest precondition computation. Here, we informally explain how the postconditions in Column 7 are obtained. At the end of executing path No. 1, we scan the path in reverse order to find the last branch instruction, which is the one at Line 5. Since the branch has been covered, we record the summary constraint $(res > c)$. Similarly, for the branch at Line 3, we record the summary constraint $(a \leq b) \land (a - b > c)$, which corresponds to the path suffix that passes through Line 3 and Line 5. For the branch at Line 1, we record the summary constraint $(a \leq 0) \land (a + 10 \leq b) \land (a + 10 - b > c)$, which corresponds to the path suffix that passes through Lines 1, 3, and 5.

Path No. 2 starts from the else-branch at Line 6. The original path condition is $\phi = (a \leq 0) \land (a + 10 \leq b) \land (a + 10 - b \leq c)$ at Line 5. In our new method, the path constraint should be $\phi' = \phi \land \neg(a - b > c)$, where $a + 10 - b > c$ is the current state of summary $res > c$ of the already explored path suffix. Since $\phi' \equiv \phi$, we do not gain anything by applying this reduction. A test input such as $a = -10, b = 2, c = 4$ can be computed by solving the path condition at Line 5.

At the end of path No. 2, we know that the branch characterized by $(res > c)$ has been explored. Furthermore, the branch characterized by $(res \leq c)$ has been explored. Therefore, the combined postcondition at Line 5 becomes $(res > c) \lor (res \leq c) \equiv true$. We continue the computation backward to Line 3, where the postcondition is $(a \leq b) \land (a - b \leq c)$. Combining it with the previously computed postcondition, we have $((a \leq b) \land (a - b > c)) \lor ((a \leq b) \land (a - b \leq c))$, which is the same as $(a \leq b)$ at Line 3. Similarly, the postcondition at Line 1 is updated to $(a \leq 0) \land (a + 10 \leq b)$.

Path No. 3 starts from the else-branch at Line 4 and then reaches Line 5. Since we are interested in exploring path suffixes not yet covered at Line 5, we check whether $\phi' = \phi \land \neg true$ is satisfiable. Here $\neg true$ is the negation of the postcondition computed at the end of path No. 2. Since $\phi'$ is unsatisfiable, the symbolic execution terminates before executing Line 5. In this case, the test input computed by the solver by solving the path condition at Line 4 can be $a = -6, b = -10, c = *$, meaning it is immaterial whether Line 5 or Line 6 is executed.

At the end of path No. 3, we compute the summary constraint $(a \leq b) \lor (a > b)$, which is the same as true at Line 3. We compute the summary constraint $(a \leq 0) \land (a + 10 \leq b) \lor (a \leq 0) \land (a + 10 > b)$, which is the same as $(a \leq 0)$ at Line 1.

In our method, path No. 4 will be skipped. In traditional symbolic execution, when executing line 5, path
No. 4 is generated through forking a new path from path No. 3 along the false-branch of the branch at line 5. However, path No. 3 terminates before executing line 5 in PSE, thus path 4 will be totally skipped in PSE.

Path No. 5 starts from the else-branch at Line 2 under the constraint \( a > 0 \). Once it reaches Line 3, our pruning algorithm shows that \( \phi' = \phi \land \lnot \text{true} \) is unsatisfiable, and therefore symbolic execution will not go beyond Line 3. In this case, the test input computed at Line 2 can be \( a = 1, b = *, c = * \), where * means that is immaterial which of the four path suffixes will be executed.

At the end of path No. 5, we compute the summary constraints. At this time, all postconditions become true, indicating that no future symbolic execution is needed. Therefore, paths No. 6-8 are skipped.

For ease of comprehension, the program used in this example is over-simplified and there are only simple data and control dependencies between the branch statements and the assignment statements. In nontrivial programs, the computation of postconditions is more complicated and the conditional expressions can be transformed due to data dependency. In the rest of this section, we present algorithms that handle the general programs.

### 3.2 Overall Algorithm

The pseudocode of our postconditioned symbolic execution is shown in Algorithm 2. The overall flow remains the same as in Algorithm 1. However, there are several notable additions.

We maintain a global key-value table called \( \Pi_{post}[l] \) at Line 2, which maps a control location \( l \) in the execution path to the summary \( \Pi_{post}[l] \) of all explored path suffixes originated from the location \( l \). Such summary table enables early termination that leads to pruning of partial or whole paths. In addition to the cases where \( instr \) is an \( abort \) at Line 10 or \( halt \) at Line 14, we also terminate the forward symbolic execution when \( (\text{pcon} \land c) \rightarrow \Pi_{post}[l'] \) holds under the current memory map at Line 19. In this case, the path condition \( (\text{pcon} \land c) \) is fully subsumed by the summary \( \Pi_{post}[l'] \) of all explored path suffixes originated from the next control location \( l' \). We can terminate early and compute the new test input at this point, because from this point on, all path suffixes have already been tested.

The summary table is created and then updated by the procedure call \( \text{UpdatePostcondition() at Lines 13, 17 and 22. When an instruction abort occurs, we invoke UpdatePostcondition(., \text{true, stack.e}) at Line 13 so a weakest precondition computation can start from terminal node } \bot \text{ with the initial logic formula true. At the end of each execution path, when the current instr is } \text{halt}, \text{ a similar computation is invoked at Line 17. The third procedure call happens at Line 22 when the current path condition is subsumed by the summary at } l'. \text{ In this case we invoke UpdatePostcondition(l', } \Pi_{post}[l'], \text{ stack.e}) so a weakest precondition computation can start from } l' \text{ with the initial logic formula being its current summary. This new procedure, to be discussed in Section 3.3, updates the summaries for all control locations along the current path.}

### 3.3 Summarizing Path Suffixes

We construct the summaries for the visited control locations incrementally. Initially \( \Pi_{post}[l] = \text{false for every control location } l \). Whenever a new test input is generated for the path \( \pi \), we update \( \Pi_{post}[l] \) for all control locations in \( \pi \) based on the weakest precondition computation along \( \pi \) in the reverse order. The weakest precondition, defined below, is a logical constraint that characterizes the suffix starting from a location \( l \) of the current execution path. If \( l \) is a terminal location \( \bot \), initially \( wp[\bot] = \text{true} \). If \( l \) is a internal location with an existing summary \( \phi \), initially \( wp[l] = \phi \). The propagation of weakest precondition at \( l \) along \( l \rightarrow l' \) is based on the type of \( instr \) as following:

- For a location \( l \) with the outgoing edge \( l \xrightarrow{v=exp} l' \), \( wp[l] \) is the logic formula computed by substituting \( v \) with \( exp \) in \( wp[l] \). That is, \( wp[l] = wp[l'][exp/v] \).
Algorithm 2 PostconditionedSymbolicExecution()

1: Let stack be the exploration state stack; a state is triple \( (pc, l, mem) \); stack.e denotes the executed event stack, i.e. a projection of all the second elements in stack.
2: Let \( \Pi_{post}[l] \) be postcondition of each statement ;
3: for each \( l, \Pi_{post}[l] \leftarrow \text{false} ; \)
4: init_state \( \leftarrow (\text{true}, l_{\text{init}}, \text{mem}_{\text{init}}) ; \)
5: stack.push( init_state ) ;
6: while ( stack is not empty )
7: \( \langle \text{peon}, l, \text{mem} \rangle \leftarrow \text{stack.pop}() ; \)
8: if ( peon is satisfiable under mem )
9: for each( event \( l \xrightarrow{\text{instr}} l' \) )
10: if ( instr is abort )
11: \( \tau \leftarrow \text{Solve} ( \text{peon}, \text{mem} ) ; / \backslash \text{bug found} \)
12: \( \mathcal{T} := \mathcal{T} \cup \{ \tau \} \); UpdatePostcondition( \( \mathcal{T}, \text{true}, \text{stack.e} \) );
13: else if ( instr is halt )
14: \( \tau \leftarrow \text{Solve} ( \text{peon}, \text{mem} ) ; \)
15: \( \mathcal{T} := \mathcal{T} \cup \{ \tau \} \); UpdatePostcondition( \( \mathcal{T}, \text{true}, \text{stack.e} \) );
16: else if ( instr is if(c) )
17: if ( peon \& c \rightarrow \Pi_{post}[l'] )
18: \( \tau \leftarrow \text{Solve} ( \text{peon}, \text{mem} ) ; \)
19: \( \mathcal{T} := \mathcal{T} \cup \{ \tau \} \); UpdatePostcondition( \( l', \Pi_{post}[l'], \) stack.e );
20: else
21: next_state \( \leftarrow (\text{peon} \wedge c, l', \text{mem}) ; \)
22: stack.push( next_state ) ;
23: end if
24: if ( instr is \( v := \text{exp} \) )
25: next_state \( \leftarrow (\text{peon}, l', \text{mem}[v \leftarrow \text{exp}] ) ; \)
26: stack.push( next_state ) ;
27: end if
28: return \( \mathcal{T} ; \)

- For a location \( l \) with the outgoing edge \( l \xrightarrow{\text{if}(c)} l' \), \( \text{wp}[l] = \text{wp}[l'] \wedge c \).

The pseudocode for updating \( \Pi_{post}[\cdot] \) is shown as the procedure UpdatePostcondition() in Algorithm 3. Since \( \Pi_{post}[l] \) is defined as the summation of multiple paths, we accumulate the effect of newly computed weakest precondition at control location \( l \) by \( \Pi_{post}[l] = \Pi_{post}[l] \vee \text{wp}[l] \). Note that updates happen only when \( l \) is the sink of a branch statement as these are the only control locations where pruning is possible.

Example 1. Consider the motivating example introduced in Section 3.1. At the end of executing path No. 1, we invoke UpdatePostcondition(), which carries out the summary computation as follows:

<table>
<thead>
<tr>
<th>loc</th>
<th>instr</th>
<th>weakest precondition</th>
<th>rule applied</th>
</tr>
</thead>
<tbody>
<tr>
<td>( l_0 )</td>
<td>if(( a \leq 0 ))</td>
<td>( (a \leq 0) \wedge (a + 10 \leq b) ) ( \wedge (a + 10 - b &gt; c) )</td>
<td>wp[( l_1 )] \wedge c</td>
</tr>
<tr>
<td>( l_1 )</td>
<td>a:=a+10</td>
<td>( (a + 10 \leq b) ) ( \wedge (a + 10 - b &gt; c) )</td>
<td>( \text{wp}[l_2][\text{exp/v}] )</td>
</tr>
<tr>
<td>( l_2 )</td>
<td>if(( a \leq b ))</td>
<td>( (a \leq b) \wedge (a - b &gt; c) )</td>
<td>wp[( l_3 )] \wedge c</td>
</tr>
<tr>
<td>( l_3 )</td>
<td>if(( \text{res} &gt; c ))</td>
<td>( (\text{res} &gt; c) )</td>
<td>( \text{wp}[l_4][\text{exp/v}] )</td>
</tr>
<tr>
<td>( l_4 )</td>
<td>if(( \text{res} = 1 ))</td>
<td>( \text{true} )</td>
<td>( \text{wp}[l_5][\text{exp/v}] )</td>
</tr>
</tbody>
</table>

3.4 Pruning Redundant Path Suffixes

We compute the summary of previously explored path suffixes in Algorithm 2, with the goal of using it to prune redundant paths. The application of summary is enabled at Line 19, where the path condition \( \text{pcon} \) of current execution \( \pi \) has been computed. Here, \( \text{pcon} \) denotes the set of program states that can be reached from some initial states via \( \pi \). In particular, when state \( \mathcal{S} = \langle \text{pcond}, l, \text{mem} \rangle \) is reached and there exists \( l \xrightarrow{\text{if}(c)} l' \), Algorithm 2 revises Algorithm 1 to enable path suffix elimination as follows:

- If \( \text{pcon} \wedge c \rightarrow \Pi_{post}[l'] \), no new path suffix can be reached by extending this execution; in such case, we force symbolic execution to backtrack from \( l \) immediately, thereby skipping the potentially large set of redundant test inputs that would have been generated for all these path suffixes.
- If \( \text{pcon} \wedge c \not\rightarrow \Pi_{post}[l'] \), there may be some path suffixes that can be reached by extending the current path from location \( l \). In this case, we continue the execution as in the standard symbolic execution procedure.

During the actual implementation, the validity of \( \text{pcon} \wedge c \rightarrow \Pi_{post}[l'] \) can be decided using a constraint
satisfiability of its negation \((pcon \land c \land \neg \Pi_{post}[l'[i]])\).

**Example 2.** Consider the motivating example introduced in Section 3.1. When executing path No. 3 in Table 1 symbolically up to Line 4, we have path condition \(pcon = (a \leq 0) \land (a + 10 > b)\). The next instruction is \(i(f(res > c))\). At the next control location \(l'\), the summary of explored path suffixes is \(\Pi_{post}[l'[i]] = true\). To check whether \((a \leq 0) \land (a + 10 > b) \land (a + 10 + b > c)\) holds, we check the satisfiability of its negation: \((a \leq 0) \land (a + 10 > b) \land (a + 10 + b > c)\) \land false. Obviously it is unsatisfiable as the formula is equivalent to false.

While running Algorithm 2, we would go inside the if-branch at Line 19. By invoking the constraint solver on \(pcon = (a \leq 0) \land (a + 10 > b)\), we can compute a test input such as \(a = -6, b = -10, c = *\), where * means the value of c does not matter.

After that, and before backtracking, we also invoke UpdatePostcondition() to summarize the partial execution path to include it into the summary table, as if we have reached the end of path No. 3.

Note that before checking the satisfiability of \((pcon \land c \land 
\neg \Pi_{post}[l'[i]])\), we have to update the variables in \(\Pi_{post}[l'[i]]\) with their values as stored in memory mem. That is, \(\Pi_{post}[l'[i]] = \forall x \in V(\Pi_{post}[l'[i]])[x/eval(x,S)], V = the variable set of \(\Pi_{post}[l'[i]]\).

**Example 3.** Consider the following code snippet. Suppose
\[
0 \ x = SYMX, \ y = SYMY;
1 \ if \ (x > 0) \ \\
1 \ \ \ \ //11: \ sum1: \ y > = 0 \ \&\& \ \ x <= -5 \ \&\& \ \ x = = 0
2 \ y = -1;
3 \ else \ if \ (x > -5) \ \\
2 \ \ \ \ //12: \ sum2: \ y > = 0 \ \&\& \ \ x <= -5
3 \ y = 0;
4 \ else
5 \ y = 1;
6 \ if \ (y > 0) \ \\
1 \ \ \ \ //13: \ sum3: \ y > = 0
7 \ return 1;
8 \ else
8 \ return 0;
\]

**Fig. 2:** Code snippet for explanation of subsumption check
the path executed first is \((1,3,5,6,7)\). After its execution the summary at L3 becomes \(y \geq 0\). When the second execution reaches at L3 with path prefix \((1,3,4)\), the current value of \(y\) is 0. Therefore \(\Pi_{post}[l'[3]]\) becomes \(0 \geq 0\) and the we can stop the execution of the current path.

### 3.5 Soundness

In the context of active testing frameworks, postconditioned symbolic execution achieves the same path coverage with standard symbolic execution, because the dynamically computed postconditions eliminate redundant paths only. In this section, we present a theorem that formally proves the claim.

**Theorem 1.** For a program \(P\), let \(T\) and \(T'\) be the test input sets generated by Algorithm 1 and Algorithm 2, respectively. For any suffix that is reachable within the specified depth bound under \(T\), the suffix will also be reachable under \(T'\).

**Proof:** We argue the correctness by contradiction. That is, there exists a suffix that is reachable under \(T\), \((2)\) but unreachable under \(T'\). Let \(s = l_1 \xrightarrow{c_1} \ldots \xrightarrow{c_{n-1}} l_n\) be a suffix of \(\pi = l_0 \xrightarrow{c_0} \ldots l_{n-1} \xrightarrow{c_{n-1}} l_n\) \((0 \leq i \leq n)\) by running a test input in \(t \in T\). We assume the opposite that \(s\) is not covered by any test input in \(T'\).

Assume \(s_{max}\) is the longest subsequence from \(s\) that can be constructed by running any test input \(t' \in T'\). Then, \(s_{max}\) will be either as a form \(l_i \xrightarrow{c_i} \ldots \xrightarrow{c_{j-1}} l_j\) \((i \leq j < n)\) or empty sequence. We first discuss when \(s_{max}\) is not a empty sequence. There are three cases to consider:

Case I. \(e_j\) is abort or halt. From \((1)\), \(\pi\) is feasible path and there is an execution instance such that \(l_j \xrightarrow{e_j} l_{j+1}\). However, when \(e_j\) is abort or halt, there is no feasible execution following \(l_j\).

Case II. \(e_j\) is a \(v ::= \text{exp}\). According to Lines 27 to 29 in Algorithm 2, \(\pi = l_i \xrightarrow{c_i} \ldots \xrightarrow{c_{j-1}} l_j\) \((j \leq i < n)\) is also reachable under \(T'\). This contradicts the assumption that \(s_{max}\) is the longest reachable subsequence from \(s\) under \(T'\).

Case III. \(e_j\) is a \(i\) \((c)\). Lines from 18 to 26 in Algorithm 2 indicate that there are two scenarios under this case:

- The first scenario is that \(l_j \xrightarrow{e_j} l_{j+1}\) and \(pecon \land c \to \Pi_{post}[j+1]\). According to \((2)\), \(T'\) is the test input, under which \(l_i \xrightarrow{c_i} \ldots \xrightarrow{c_{j-1}} l_j\) is reachable. Moreover, According to Algorithm 3 that processes executed events stack.e from Algorithm 2, \(\Pi_{post}[j+1]\) is a summation of weakest preconditions for multiple paths from \(j+1\). From \(pecon \land c \to \Pi_{post}[j+1]\), suffix \(l_j \xrightarrow{c_j} \ldots \xrightarrow{c_{j-1}} l_j \xrightarrow{c_{j-1}} l_{j+1}\) is also reachable under \(T'\), which contradicts the assumption that \(s_{max}\) is the longest subsequence from \(s\) that is reachable under \(T'\).

- The second scenario is that \(l_j \xrightarrow{e_j} l_{j+1}\) and \(pecon \land c \not\to \Pi_{post}[j+1]\). According to \((1)\), \(pecon \land c\) is satisfiable under mem, and \(pecon \land c \not\to \Pi_{post}[j+1]\). Thus there is a divergent path from \(l_{j+1}\) that has not been explored before. Lines 24 to 25 in Algorithm 2 pushes \(\langle pecon \land c, l+1, mem\rangle\) into stack. It explores the sub-space under \(l+1\) and generate other test inputs into \(T'\). Therefore \(l_i \xrightarrow{c_i} \ldots \xrightarrow{c_{j-1}} l_j \xrightarrow{c_{j-1}} l_{j+1}\) is reachable under \(T'\), contradicting the assumption that \(s_{max}\) is the longest subsequence from \(s\) that is reachable under \(T'\).

If \(s_{max}\) is empty sequence, by considering \(e_{j-1}\), similar contradictions can be deduced. \(\square\)

**Corollary 1.** For abort statement \(s\) in a program \(P\), let \(T\) and \(T'\) be the test input sets generated by Algorithm 1 and
Algorithm 2, respectively. If $s$ is reachable under $T$, it will be reachable under $T'$ as well.

### 3.6 The Impact of Search Strategies

In Algorithm 2, the states waiting to be processed by the symbolic execution procedure are stored in a stack, which leads to a Depth-First Search (DFS) of the directed acyclic graph that represents all possible execution paths. At any moment during the symbolic execution, the table $\Pi_{\text{post}}[\_\_]$ has the up-to-date information about which path suffixes have been explored. This is when our postconditioned symbolic execution method performs at its best.

In contrast, if Algorithm 2 is implemented by replacing the state stack with a queue, it would lead to a Breadth-First Search (BFS) strategy. This is when our path suffix elimination method performs at its worst. To see why using the BFS strategy makes it impossible to pruning redundant paths, consider the running example introduced in Section 3.1.

With BFS, the symbolic execution procedure would have symbolically propagated the path condition along all eight paths, before solving the first one to compute the test input. During the process, the table $\Pi_{\text{post}}[\_\_]$ is empty because there does not yet exist any explored path. By the time we compute the test inputs for path No. 1 and No. 2, and update the table $\Pi_{\text{post}}[\_\_\_\_\_\_]$, it would have been too late. In particular, we would have already constructed the path conditions for all the other six paths.

### 4 Optimization

When the execution paths are long and many, the size of the table $\Pi_{\text{post}}[\_\_]$ as well as the size of each logic constraint $\Pi_{\text{post}}[\_\_]$ may be large. The summary $\Pi_{\text{post}}[\_\_]$, in particular, needs to be stored in a persistent media, meaning that the keys of the table are the global control locations, and the values are the logic formulas representing $\Pi_{\text{post}}[\_\_\_]$ at global control location $l$. In general, these logical formulas can become complex.

Therefore, in practice, we use various heuristic approximations to reduce the computational cost associated with the construction, storage, and retrieval of the summaries. Our goal is to reduce the cost while maintaining the soundness of the pruning, i.e., the guarantee of no missed paths.

#### 4.1 Under-Approximation

We prove that, in general, any under-approximation of $\Pi_{\text{post}}[\_\_]$ can be used in Algorithm 3 to replace $\Pi_{\text{post}}[\_\_]$ while maintaining the soundness of our redundant path pruning method; that is, we can still guarantee to cover all valid paths of the program. In many cases, using an under-approximation $\Pi_{\text{post}}[\_\_]$ to replace $\Pi_{\text{post}}[\_\_]$ can make the computation significantly cheaper. The reason why it is always safe to do so is that, by definition, we have $\Pi_{\text{post}}[\_\_\_] \rightarrow \Pi_{\text{post}}[\_\_]$. Therefore, if $(\text{pcon} \land c) \rightarrow \Pi_{\text{post}}[\_\_]$ holds, so does $(\text{pcon} \land c) \rightarrow \Pi_{\text{post}}[\_\_]$.

A main advantage of our new pruning method is that it allows for the use of (any kind of) under-approximation of the table $\Pi_{\text{post}}[\_\_]$ while maintaining soundness. This is in contrast to ad hoc reduction techniques for test reduction, where one has to be careful not to accidentally drop any executions that may lead to bugs. Using our method, one can concentrate on exploring the various practical ways of trading off the pruning power for reduced computational cost, while not worrying about the soundness of these design choices.

In principle, we can use a hash table with a fixed number of entries to limit the storage cost for $\Pi_{\text{post}}[\_\_]$. With a bounded table, two global control locations $l$ and $l'$ may be mapped to the same hash table entry. In this case, we may use a lossy insertion: Upon a key collision, i.e., $\text{key}(l) = \text{key}(l')$, we heuristically remove one of the entries, effectively making it false (hence an under-approximation).

In the actual implementation, we exploit the knowledge of program structures to reduce the number of summary constraints. Considering that loops are the major component of a trace, we only store and check the summary at the first instance of a branch in loops. The intuition is that when one subsuming check fails at some branch in a loop, there is little chance of success at its following check points in the same loop. Our experiments confirm that such strategy is not only efficient in reducing the checking time and memory consumption, but also incurring little affect on the number of pruned paths and instructions.

We also use a fixed threshold to bound the size of the individual logic formula of $\Pi_{\text{post}}[\_\_]$. That is, we replace Line 9 in Algorithm 3 by the following statement:

\[
\text{if (size}(\Pi_{\text{post}}[\_\_])(\_\_\_\_\_\_] < \text{bd}) \quad \Pi_{\text{post}}[\_\_\_] \leftarrow \Pi_{\text{post}}[\_\_] \lor \text{wp};
\]

#### 4.2 Summary Simplification

\begin{algorithm}
\begin{algorithmic}
\State 1: \textbf{for} each item $d' \in S$
\State 2: \quad \text{flag} \leftarrow \text{false};
\State 3: \quad \text{if} (d' \text{ and } d \text{ are combinable})
\State 4: \quad \quad \text{Let } c \text{ and } c' \text{ be the divergence root of } d \text{ and } d';
\State 5: \quad \quad \quad d' \leftarrow \text{remove } c' \text{ from } d';
\State 6: \quad \quad \quad S \leftarrow S \setminus \{d'\};
\State 7: \quad \text{return } CombineSummary(S, d');
\State 8: \quad \textbf{end if}
\State 9: \textbf{end for}
\State 10: \textbf{if} (\text{flag} \text{ == false})
\State 11: \quad S \leftarrow S \cup \{d\};
\State 12: \textbf{end if}
\State 13: \text{return } S
\end{algorithmic}
\end{algorithm}

The size of summary constraints certainly has impact on the efficiency of subsumption checking. Besides a brute-force approach to bound its size, we simplify the
constraints before they become too big. In particular, we combine two constraint terms in a summary if possible.

The computation of summary is based on weakest precondition computation, as presented in Algorithm 3. The procedure produces summary constraints in the format of conjunctive normal form (CNF). Two disjunctive terms \( d = c_1 \land \ldots \land c_n \land c_{n+1} \) and \( d' = c'_1 \land \ldots \land c'_n \land c'_{n+1} \) are combinable if (1) \( c_i = c'_i (1 \leq i \leq n) \) and (2) \( c_{n+1} \lor c'_{n+1} = \text{true} \). We call \( c_{n+1} \) and \( c'_{n+1} \) the divergence roots of \( d \) and \( d' \). For instance, \( d = c_1 \land c_2 \land c_3 \) and \( d' = c_1 \land c_2 \land c_3 \) are combinable and their divergence roots are \( c_3 \) and \( c_3 \). Two disjunctive terms can be replaced by one by throwing away the divergence roots. In our example, \( d \) and \( d' \) are replaced by \( c_1 \land c_2 \).

Algorithm 4 gives the pseudo-code of the summary simplification procedure. When a disjunctive term \( d \) is added to a summary \( S \), our method carries out an iterative search to locate a \( d' \) that is combinable with \( d \). When such term is found (Line 3), the algorithm recursively looks for more combinable terms at Line 7.

We find such straightforward optimization effective in practice. Consider the program given in Figure 3. At the end of the postconditioned symbolic execution, six paths are explored. Table 2 describes the process of summary computation for the branch statement at Line 1 in Figure 3. The first column gives the unique path index, followed by a sequence of branches that each path goes through. Underlined line numbers indicate else-branches. The following two columns give the summary constraints with and without simplification, respectively. We also list the summary constraint at Line 9 in Figure 3, with and without simplification, in Table 3. As observed in both tables, the simplification can significantly reduce the size of summary constraints.

**TABLE 3: Symbolic computation for the program in Figure 3.**

<table>
<thead>
<tr>
<th>( Sum_a )</th>
<th>( Sum'_a )</th>
</tr>
</thead>
<tbody>
<tr>
<td>I: ( e - 1 &gt; 0 \land y \leq 0 )</td>
<td>I: true</td>
</tr>
<tr>
<td>II: ( e - 1 \leq 0 \land y \leq 0 )</td>
<td></td>
</tr>
<tr>
<td>III: ( (e &gt; 0 \lor e &lt; 0) \land c + 1 \leq 0 \land y &gt; 0 )</td>
<td></td>
</tr>
<tr>
<td>IV: ( (e &gt; 0 \lor e &lt; 0) \land c + 1 &gt; 0 \land y &gt; 0 )</td>
<td></td>
</tr>
</tbody>
</table>

4.3 Avoid Unnecessary Weakest Precondition Computation

Our method requires weakest precondition computation along each explored path, which has a strong impact on the efficiency of the method, as confirmed by our empirical study on overhead. Since postconditioned symbolic execution always adopts depth-first search, we exploit this fact to improve the performance of weakest precondition computation by reusing the results of previous computation on path prefix.

![Fig. 4: Two adjacent paths: the previous path (Old) and the present one (New).](image)

Figure 4 presents two adjacent paths \( \text{Old} \) and \( \text{New} \) with each node indicating a branch. They share a path prefix up to index \( x - 1 \) and separate at \( x \). In order to enable reuse, we store the result of the latest weakest precondition at each branch instance. When we conduct the weakest precondition computation along path \( \text{New} \), there is no need to propagate condition \( N_1 \) to \( N_{x-1} \) along the path, because it is the same as that with \( O_1 \) to \( O_{x-1} \) along path \( \text{Old} \) that has already been computed and stored. Thus, the weakest precondition computation along path \( \text{New} \) only needs to propagate \( N_x \) to \( N_m \) along \( \text{New} \) and reuses the weakest precondition from \( N_1 \) to \( N_{x-1} \).

Let \( WP[p][i] \) be the weakest precondition of path \( p \) at the \( i \)-th branch instance, and \( WP[p]_j \) be the transferred condition of that from the \( j \)-th branch instance when it arrives the \( i \)-th branch along path \( p \). We can get the following weakest precondition computation equation.

\[
WP[\text{New}][i] = WP_{\text{old}}[i] \land WP_{\text{new}}[i]
\]

\[
WP_{\text{old}}[i] = \land_{j=1}^{\text{diffDBr}-1} WP[\text{Old}][j]
\]

\[
WP_{\text{new}}[i] = \land_{j=\max(i, \text{diffDBr})}^{m} WP[\text{Old}][j]
\]

The new computation of the weakest precondition at the \( i \)-th branch contains two parts: \( WP_{\text{old}}[i] \) and \( WP_{\text{new}}[i] \), which respectively represent the reused part...
TABLE 2: Symbolic computation for the program in Figure 3. The bold items are the newly updated items.

<table>
<thead>
<tr>
<th>path idx</th>
<th>path</th>
<th>Summary at Line 1</th>
<th>Summary at Line 1(after simplification)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 9 14</td>
<td>I1: e - 1 &gt; 0 ∧ y ≤ 0 ∧ x ≤ 0</td>
<td>I1: e - 1 &gt; 0 ∧ y ≤ 0 ∧ x ≤ 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>I2: e - 1 &gt; 0 ∧ y ≤ 0 ∧ x ≤ 0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1 9 17</td>
<td>I1: e - 1 &gt; 0 ∧ y ≤ 0 ∧ x ≤ 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>I2: e - 1 ≤ 0 ∧ y ≤ 0 ∧ x ≤ 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>I3: (e &gt; 0 ∧ e &lt; 0) ∧ c + 1 ≤ 0 ∧ y &gt; 0 ∧ x ≤ 0</td>
<td>=⇒ I1: y ≤ 0 ∧ x ≤ 0</td>
</tr>
<tr>
<td>3</td>
<td>1 9 11</td>
<td>I1: e - 1 &gt; 0 ∧ y ≤ 0 ∧ x ≤ 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>I2: e - 1 ≤ 0 ∧ y ≤ 0 ∧ x ≤ 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>I3: (e &gt; 0 ∧ e &lt; 0) ∧ c + 1 ≤ 0 ∧ y &gt; 0 ∧ x ≤ 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>I4: (e &gt; 0 ∧ e &lt; 0) ∧ c + 1 &gt; 0 ∧ y &gt; 0 ∧ x ≤ 0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1 9 11</td>
<td>I1: e - 1 &gt; 0 ∧ y ≤ 0 ∧ x ≤ 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>I2: e - 1 ≤ 0 ∧ y ≤ 0 ∧ x ≤ 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>I3: (e &gt; 0 ∧ e &lt; 0) ∧ c + 1 ≤ 0 ∧ y &gt; 0 ∧ x ≤ 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>I4: (e &gt; 0 ∧ e &lt; 0) ∧ c + 1 &gt; 0 ∧ y &gt; 0 ∧ x ≤ 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>=⇒ I1: y ≤ 0 ∧ x ≤ 0</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1 3</td>
<td>I1: e - 1 &gt; 0 ∧ y ≤ 0 ∧ x ≤ 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>I2: e - 1 ≤ 0 ∧ y ≤ 0 ∧ x ≤ 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>I3: (e &gt; 0 ∧ e &lt; 0) ∧ c + 1 ≤ 0 ∧ y &gt; 0 ∧ x ≤ 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>I4: (e &gt; 0 ∧ e &lt; 0) ∧ c + 1 &gt; 0 ∧ y &gt; 0 ∧ x ≤ 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>I5: Sumₐ ∧ a ≤ 0 ∧ x &gt; 0</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1 3</td>
<td>I1: e - 1 &gt; 0 ∧ y ≤ 0 ∧ x ≤ 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>I2: e - 1 ≤ 0 ∧ y ≤ 0 ∧ x ≤ 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>I3: (e &gt; 0 ∧ e &lt; 0) ∧ c + 1 ≤ 0 ∧ y &gt; 0 ∧ x ≤ 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>I4: (e &gt; 0 ∧ e &lt; 0) ∧ c + 1 &gt; 0 ∧ y &gt; 0 ∧ x ≤ 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>I5: Sumₐ ∧ a ≤ 0 ∧ x &gt; 0</td>
<td>=⇒ a ≤ 0 ∧ x &gt; 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>I6: Sumₐ ∧ a &gt; 0 ∧ x &gt; 0</td>
<td>=⇒ true</td>
</tr>
</tbody>
</table>

and the changed part compared with the previously explored path.

In order to understand the reason that such optimization is efficient, consider the following code snippet:

```plaintext
if (A) {...} else {...}
if (B) {...} else {...}
if (C) {...} else {...}
if (D) {...} else {...}
```

After the first path $ABCD$ is explored, the summary constraints produced by the weakest precondition computation are $D, D^1 \land C, D^2 \land C^1 \land B, D^3 \land C^2 \land B^1 \land A$ at the four control locations in reverse order. Here we use superscripts to indicate the transformations of the conditions. The second path is $ABC\neg D$, which shares prefix $ABC$ with the previous path. In our implementation only $\neg D$ is propagated, while $C^i, B^i$ and $A^i$ are reused during the weakest precondition computation. Finally, we can easily construct the weakest precondition at the four control locations along the second path: $\neg D, \neg D^1 \land C, \neg D^2 \land C^1 \land B, \neg D^3 \land C^2 \land B^1 \land A$, in which only $\neg D, \neg D^1, \neg D^2$ and $\neg D^3$ need to be reconstructed.

5 EXPERIMENTS

To evaluate the effectiveness of postconditioned symbolic execution in pruning redundant test cases, we consider the following research questions:

- Q1: (Efficiency of pruning) Compared with KLEE, how much redundancy is there in real-world applications?
  - Q1.1: How many redundant paths can be pruned by PSE?
  - Q1.2: How many redundant instructions can be pruned by PSE?
  - Q1.3: How much faster is PSE than KLEE?
- Q2: (Pruning overhead) How much computational overhead does the pruning method introduce? With such computational overhead, are there net benefits for applying the pruning?
  - Q2.1: How much computational overhead of weakest precondition computation does the pruning method introduce?
  - Q2.2: How much computational overhead of subsumption check does the pruning method introduce?
- Q3: (Optimizations Evaluation) How does our optimizations affect the result of before research questions?
  - Q3.1: How does the under-approximation strategy affect the efficiency of path pruning?
  - Q3.2: How does the optimization of summary simplification affect the memory consumption?
  - Q3.3: How does the strategy of avoiding unnecessary weakest precondition computation between two adjacent paths affect the efficiency of path pruning?

We have implemented the proposed method in KLEE, which is a state-of-the-art symbolic execution tool built on the LLVM platform. It provides stub functions for standard library calls, e.g., using uclibc to model glibc calls, and concrete-value based under-approximated modeling of other external function calls. In practice, this is a crucial feature because system calls as well as calls to external libraries are common in real-world applications.

5.1 Subjects and Methodology

We have conducted experiments on a large set of C programs from the GNU Coreutils suite, which imple-
ments the basic commands in the Unix/Linux operating system. These programs are of medium size, each with between 2000 to 6000 lines of code. They are challenging for symbolic execution tools partly because they have extensive use of error checking code, pointers, and heap allocated data structures such as lists and trees.

Each program is first transformed into the LLVM bytecode using the standard Clang/LLVM tool-set. The symbolic execution procedures take the LLVM bytecode program and a set of user annotated symbolic variables as input. The symbolic inputs are variables that represent the values of the program’s command-line arguments.

5.2 Effectiveness of Pruning

We evaluate the effectiveness of our pruning method by comparing postconditioned symbolic execution against KLEE, which uses the standard symbolic execution procedure described in Algorithm 1. We run both methods on each benchmark program for up to three hours (10800 seconds). For PSE, we run it with all optimization options enabled.

For these experiments, we have used the symbolic command-line arguments and stdin as inputs of the programs, while bounding the string sizes of the content of each argument to 2. The programs are all terminating due to the proper test harness and the bound on the size of the symbolic inputs. All experiments were performed on a computer with a 2.66 GHz Intel dual core CPU and 4 GB RAM.

Figure 5 shows the scatter diagram that compares the performance of PSE against KLEE. The X-axis and Y-axis give the execution time in seconds of all the 94 benchmarks. If the experiment of a benchmark exceeds the time limit, we show its execution time as 10800 seconds in the figure. The figure clearly indicates that the standard symbolic execution is more efficient when a benchmark is small, and PSE starts to outperform KLEE when the size of benchmarks become larger. This is what we have expected as postconditioned symbolic execution incurs significant overhead, which is to be presented in Section 5.3. Due to the interest in the cases when applying standard symbolic execution is challenging, for the rest of the section we only give the experimental results of the 57 benchmarks that take PSE less than 3 hours and take KLEE more than 300 seconds (5 minutes) to complete.

The experimental results are given in Table 4. Column 1 lists the names of the benchmarks. Columns 2 to 4 compare the number of explored paths by PSE and KLEE. Columns 5 to 7 compare the number of executed instructions by PSE and KLEE. Columns 8 to 10 compare the execution time of PSE and KLEE in seconds, where TO (Time Out) indicate that the corresponding program did not terminate within the given time limit. Note that the table gives the improvement in terms of the reduction ratio in the number of explored paths and instructions, as well as the speedup ratio in the execution time, of PSE over KLEE. With pruning of PSE, the number of paths required to achieve exhaustive coverage is reduced by about 3.34X. That is, on average about 60% of paths are considered redundant by our method. Most of the reduction actually comes from path-suffix elimination rather than whole-path elimination. This is indicated by the column that compares the number of executed instructions. Compared with KLEE, PSE reduces the executed instructions by about 23.03X. The results confirm our conjecture that redundancy due to common path suffix is abundant and widespread in real-world applications and our new method is effective in eliminating the redundant paths.

The speedup in time, however, is less drastic. Other than the 18 benchmarks that KLEE cannot complete within the three hour time whereas PSE can, the average speedup achieved by PSE is 2.26X. This is in contrast with 3.34X and 23.03X in the path and instruction reductions. In Section 5.3, we give the reasons. Note that for all the programs on which KLEE terminates PSE also terminates.

5.3 Pruning Overhead

There are two major sources of overhead incurred by the pruning.

- Weakest precondition computation: After each path exploration, we need to conduct weakest precondition computation along the path, which increases the computational cost as well as memory usage as we need to store complex postconditions at control locations.

- Subsumption check: We need to conduct SMT solving to check whether the current execution is subsumed by previous paths. The solving is expensive and it may also increase the internal SMT memory consumption.

Figures 6 and 7 show the pruning overhead. Figure 6 presents three kinds of time: check time, wp time and all time, which represents the time spent on subsumption check, weakest precondition computation, and complete execution, respectively. It is obvious that in most cases
### TABLE 4: Comparison between KLEE and PSE.

<table>
<thead>
<tr>
<th>Test Program</th>
<th>#Explored Paths</th>
<th>#Explored Instructions</th>
<th>Time(s)</th>
<th>KLEE Speedup in time</th>
</tr>
</thead>
<tbody>
<tr>
<td>arch</td>
<td>855</td>
<td>1375</td>
<td>1.61X</td>
<td>1.53X</td>
</tr>
<tr>
<td>base64</td>
<td>441</td>
<td>1058</td>
<td>2.40X</td>
<td>1.13X</td>
</tr>
<tr>
<td>chcon</td>
<td>1398</td>
<td>3752</td>
<td>&gt;2.68X</td>
<td>&gt;2.13X</td>
</tr>
<tr>
<td>chgrp</td>
<td>1418</td>
<td>3920</td>
<td>&gt;2.76X</td>
<td>&gt;2.76X</td>
</tr>
<tr>
<td>chmod</td>
<td>1453</td>
<td>3661</td>
<td>&gt;2.52X</td>
<td>&gt;2.52X</td>
</tr>
<tr>
<td>chown</td>
<td>1672</td>
<td>3925</td>
<td>&gt;2.35X</td>
<td>&gt;2.35X</td>
</tr>
<tr>
<td>comm</td>
<td>885</td>
<td>2222</td>
<td>2.85X</td>
<td>1.53X</td>
</tr>
<tr>
<td>cp</td>
<td>1276</td>
<td>3636</td>
<td>&gt;2.85X</td>
<td>&gt;2.85X</td>
</tr>
<tr>
<td>cpast</td>
<td>930</td>
<td>3235</td>
<td>1.48X</td>
<td>1.07X</td>
</tr>
<tr>
<td>dircolors</td>
<td>415</td>
<td>1178</td>
<td>2.84X</td>
<td>2.13X</td>
</tr>
<tr>
<td>dimname</td>
<td>220</td>
<td>564</td>
<td>2.76X</td>
<td>2.13X</td>
</tr>
<tr>
<td>du</td>
<td>297</td>
<td>949</td>
<td>3.20X</td>
<td>2.13X</td>
</tr>
<tr>
<td>expand</td>
<td>415</td>
<td>762</td>
<td>1.84X</td>
<td>1.84X</td>
</tr>
<tr>
<td>expr</td>
<td>69</td>
<td>651</td>
<td>9.43X</td>
<td>9.43X</td>
</tr>
<tr>
<td>factor</td>
<td>1387</td>
<td>3812</td>
<td>&gt;2.75X</td>
<td>&gt;2.75X</td>
</tr>
<tr>
<td>fmtree</td>
<td>206</td>
<td>792</td>
<td>3.84X</td>
<td>3.84X</td>
</tr>
<tr>
<td>fold</td>
<td>423</td>
<td>967</td>
<td>2.29X</td>
<td>2.29X</td>
</tr>
<tr>
<td>ginstall</td>
<td>1166</td>
<td>4911</td>
<td>4.21X</td>
<td>4.21X</td>
</tr>
<tr>
<td>head</td>
<td>1111</td>
<td>3721</td>
<td>&gt;3.34X</td>
<td>&gt;3.34X</td>
</tr>
<tr>
<td>hostid</td>
<td>855</td>
<td>1375</td>
<td>1.61X</td>
<td>1.61X</td>
</tr>
<tr>
<td>hostname</td>
<td>505</td>
<td>1375</td>
<td>2.72X</td>
<td>2.72X</td>
</tr>
<tr>
<td>id</td>
<td>423</td>
<td>1340</td>
<td>3.17X</td>
<td>3.17X</td>
</tr>
<tr>
<td>join</td>
<td>636</td>
<td>3699</td>
<td>&gt;5.82X</td>
<td>&gt;5.82X</td>
</tr>
<tr>
<td>link</td>
<td>1348</td>
<td>2782</td>
<td>2.04X</td>
<td>2.04X</td>
</tr>
<tr>
<td>ln</td>
<td>495</td>
<td>1510</td>
<td>3.05X</td>
<td>3.05X</td>
</tr>
<tr>
<td>logname</td>
<td>855</td>
<td>1375</td>
<td>1.61X</td>
<td>1.61X</td>
</tr>
<tr>
<td>ls</td>
<td>1275</td>
<td>3192</td>
<td>&gt;2.50X</td>
<td>&gt;2.50X</td>
</tr>
<tr>
<td>mkdf</td>
<td>1273</td>
<td>2768</td>
<td>&gt;2.45X</td>
<td>&gt;2.45X</td>
</tr>
<tr>
<td>mkfifo</td>
<td>867</td>
<td>2649</td>
<td>&gt;3.06X</td>
<td>&gt;3.06X</td>
</tr>
<tr>
<td>mknod</td>
<td>1109</td>
<td>1756</td>
<td>1.58X</td>
<td>1.58X</td>
</tr>
<tr>
<td>mktemp</td>
<td>1102</td>
<td>3279</td>
<td>2.98X</td>
<td>2.98X</td>
</tr>
<tr>
<td>mv</td>
<td>965</td>
<td>3446</td>
<td>&gt;3.58X</td>
<td>&gt;3.58X</td>
</tr>
<tr>
<td>nice</td>
<td>44</td>
<td>67</td>
<td>14.48X</td>
<td>14.48X</td>
</tr>
<tr>
<td>nl</td>
<td>1102</td>
<td>2148</td>
<td>1.95X</td>
<td>1.95X</td>
</tr>
<tr>
<td>nfhup</td>
<td>107</td>
<td>924</td>
<td>5.80X</td>
<td>5.80X</td>
</tr>
<tr>
<td>od</td>
<td>832</td>
<td>2851</td>
<td>3.43X</td>
<td>3.43X</td>
</tr>
<tr>
<td>printenv</td>
<td>937</td>
<td>3663</td>
<td>3.91X</td>
<td>3.91X</td>
</tr>
<tr>
<td>printf</td>
<td>1288</td>
<td>2014</td>
<td>&gt;1.56X</td>
<td>&gt;1.56X</td>
</tr>
<tr>
<td>ptx</td>
<td>332</td>
<td>1914</td>
<td>3.60X</td>
<td>3.60X</td>
</tr>
<tr>
<td>readline</td>
<td>400</td>
<td>745</td>
<td>1.86X</td>
<td>1.86X</td>
</tr>
<tr>
<td>rm</td>
<td>1134</td>
<td>3132</td>
<td>&gt;2.76X</td>
<td>&gt;2.76X</td>
</tr>
<tr>
<td>setuidgid</td>
<td>256</td>
<td>1848</td>
<td>5.16X</td>
<td>5.16X</td>
</tr>
<tr>
<td>shuf</td>
<td>457</td>
<td>1611</td>
<td>3.11X</td>
<td>3.11X</td>
</tr>
<tr>
<td>sleep</td>
<td>855</td>
<td>3173</td>
<td>&gt;3.71X</td>
<td>&gt;3.71X</td>
</tr>
<tr>
<td>sort</td>
<td>396</td>
<td>1801</td>
<td>4.55X</td>
<td>4.55X</td>
</tr>
<tr>
<td>split</td>
<td>298</td>
<td>738</td>
<td>2.48X</td>
<td>2.48X</td>
</tr>
<tr>
<td>touch</td>
<td>305</td>
<td>1288</td>
<td>4.22X</td>
<td>4.22X</td>
</tr>
<tr>
<td>tr</td>
<td>776</td>
<td>1690</td>
<td>2.18X</td>
<td>2.18X</td>
</tr>
<tr>
<td>tsort</td>
<td>381</td>
<td>580</td>
<td>1.52X</td>
<td>1.52X</td>
</tr>
<tr>
<td>tty</td>
<td>1193</td>
<td>1927</td>
<td>1.62X</td>
<td>1.62X</td>
</tr>
<tr>
<td>uname</td>
<td>1013</td>
<td>2162</td>
<td>&gt;2.13X</td>
<td>&gt;2.13X</td>
</tr>
<tr>
<td>unexpand</td>
<td>259</td>
<td>812</td>
<td>3.14X</td>
<td>3.14X</td>
</tr>
<tr>
<td>uniq</td>
<td>368</td>
<td>939</td>
<td>2.55X</td>
<td>2.55X</td>
</tr>
<tr>
<td>unlink</td>
<td>477</td>
<td>1375</td>
<td>2.88X</td>
<td>2.88X</td>
</tr>
<tr>
<td>uptime</td>
<td>64</td>
<td>577</td>
<td>9.02X</td>
<td>9.02X</td>
</tr>
<tr>
<td>users</td>
<td>378</td>
<td>577</td>
<td>1.53X</td>
<td>1.53X</td>
</tr>
<tr>
<td>whoami</td>
<td>855</td>
<td>1375</td>
<td>1.61X</td>
<td>1.61X</td>
</tr>
<tr>
<td>Average</td>
<td>-</td>
<td>-</td>
<td>&gt;3.34X</td>
<td>&gt;23.0X</td>
</tr>
</tbody>
</table>
5.4 Evaluation of Optimizations

In this subsection we evaluate the effectiveness of our optimizations. Note that when evaluating an optimization, we enable all the other optimizations and present the result with / without the evaluated optimization.

Under-approximation. We have evaluated subsumption check with and without under-approximation. The under-approximation is mainly to conduct summary at selected branch instances. In our current implementation, we skip all the instances but the first one of a branch. In addition, we bound the size of each summary to 200. Among the benchmarks, six of them, including \texttt{factor}, \texttt{fold}, \texttt{join}, \texttt{ls}, \texttt{mkdir}, and \texttt{uname}, fail to terminate within the three-hour time limit without such under-approximation. For other programs, we give the time usage and memory consumption ratio with and without the optimization in Figure 8. As expected, our under-approximation strategy not only speeds up the subsumption check, but also greatly reduces its memory consumption. On average, there is a speed up of 1.97X, and 2.34X reduction in memory consumption. In addition, only about 7% more instructions are executed.

Summary simplification. Summary simplification can not only reduce the memory consumption, but also speed up subsumption check. In Figure 9, we compare the memory consumption with and without this optimization. On average, about 43% more memory is consumed without summary simplification. Figure 10 depicts the time usage for subsumption check. On average, summary simplification leads to a 2X speedup.

Reusing weakest precondition computation. Figure 11 compares the time usage for weakest precondition computation with and without reusing previous results. On average, such optimization achieves a speedup of 2.56X. In addition, five test cases, including \texttt{factor}, \texttt{head}, \texttt{mktemp}, \texttt{prin...
is, mkdir, and `uname` are omitted from Figure 11 because without the reusing heuristics, weakest precondition computation fails to terminate within the time limit of three hours.

5.5 Threats to Validity of Evaluation

Based on our experiments on GNU Coreutils suite, we learn that program semantics do not have direct impact on postconditioned symbolic execution. However, the experimental results depend on two factors that are the program structure of the benchmarks and experiment parameters.

Program structure. The program structure of the benchmarks may affect the effectiveness of postconditioned symbolic execution. First, our method is based on the observation that many path suffixes are shared among different test runs. The occurrence rate of common suffixes in a benchmark may have an impact on the results for Q1. As showed in Table 4, our tool achieves various speed-up. In the extreme case where there is no common path suffix, our method probably deteriorates performance. Although our empirical study on Coreutils suite confirms that redundancy due to common path suffixes is both abundant and widespread, more experiments on diverse benchmarks are helpful to verify our observation. Second, from Section 5.3, we have learned that around 70% test time is spent on subsumption check and weakest precondition computation. Therefore, the two types of overhead have significant impact on the performance of postconditioned symbolic execution. The
cost of subsumption check and weakest precondition computation is sensitive to data and control dependencies in a program. Third, our optimization methods are all based on some insights in symbolic execution. They are effective only under specific program patterns. For example, in the under-approximation optimization, we skip all the instances except the first because of the knowledge about loop structure mentioned in Section 4.1. The occurrence rate of program snippets also affects results of experiment for Q3.1, which is the reason that we obtain different time usage and memory consumption ratios for the programs in Figure 8.

**Experiment parameters.** In our evaluation, we used *ad hoc* values for some parameters in the experiment settings and tool implementation. For example, we set the string size in each argument of Coreutils programs to 2 and set the time limit to three hours for the experiments. These parameters affect evaluation results from Q1 to Q3. We also tuned some parameters in the implementation of our tool. For instance, we limited the size of each summary to 200 when implementing the under-approximation optimization. As the under-approximation is used to balance memory consumption and pruning precision, this bound value has an impact on the experimental results for Q3.1.
6 RELATED WORK

As we have mentioned earlier, there is a large body of work on test input generation based on symbolic execution [2], [3], [4], [5], [6], [1]. Some symbolic execution methods and tools have been used in real applications [10], [11], [12], [13]. However, a major obstacle that prevents these methods from getting even wider application is the path explosion problem. Although there are efforts on mitigating the problem, e.g., by using methods based on compositionality [14], abstraction-refinement [15], interpolation [16], [17], [18], [19], and parallelization [20], [21], [22], [23], [24], path explosion remains a bottleneck in scaling symbolic execution to larger applications.

6.1 Reduction by Interpolation

McMillan proposed a redundancy removal method for symbolic execution, called lazy annotation [25], [16]. The method computes an interpolant from an unsatisfiable formula due to the unreachability of certain branch conditions in a program. The interpolant can be regarded as an over-approximated set of forward reachable states. Similarly, Yogi [26], [27] uses interpolants along infeasible program paths, chosen based on concrete test-case executions, in order to strengthen a partition-graph of the state space of a program. Jaffar et al. [17], [18], [19] proposed a similar method in the context of dynamic programming, for computing resource-constrained shortest paths and analyzing the worst-case execution time. In TRACER [28], [29], [30], [31], they extended works by using interpolation in the context of symbolic execution, which aim to tackle path exponential exploration and infinite-length symbolic paths due to unbounded loops. The idea of interpolant has also been used in other program analysis techniques, such as model checking [32], [33], [34], [35] and program verification [36], [37]. Although interpolant is more general than weakest precondition, it is also more expensive to compute and requires special constraint solvers. For example, TRACER [30], [31] performs symbolic execution computing approximated weakest preconditions as interpolants and employs CLP(R) [38] as solver. However, our tool works on quantifier-free first-order logic formula and decides formulas by STP [39], a SMT solver.

6.2 Reduction by Compositionality

There are also pruning methods based on computing summaries. For example, Godefroid [14] proposed a function summary based compositional test generation algorithm, where the input-output summary of a previously explored function is computed and stored into a database; when the function is executed again, the symbolic constraints are reused. This was extended in [40] with a demand-driven top-down approach that uses execution trees. Instead each method summary is represented as a first-order logic formula with uninterpreted functions and the composition is performed entirely using SMT solving. A further extension [41] was proposed, named compositional may-must program analysis, to speed up symbolic execution using the result from over-approximated analysis and vice versa. Majumdar and Sen [15] proposed a demand-driven abstraction-refinement style hybrid concolic testing algorithm, which can achieve a similar reduction. Păsăreanu et al. [42] described a preliminary investigation of compositional symbolic execution for Java bytecodes. They use partial evaluation, a well-established technique that aims at automatically specializing a program with respect to some of its input, to build method summaries consisting of several path-specialized versions of the method code. Qiu et al. [43] proposed a new structure, called memo- rization trees, to capture heap operations when calling a function and then replay it in symbolic execution to get speed up.

However, our new method is significantly different from above works in that our common path suffix elimination method is not restricted to the function boundary, and does not need the abstract-refinement loop.

6.3 Reduction by Reusing Constraint Solutions and Constraint Reducing

Constraint solution reuse is an effective approach to save the time of constraint solving in symbolic execution [44], [45], [46]. The GREEN tool [44] by Visser et al. provides a wrapper around constraint satisfiability solvers to check if the results are already available from prior invocations, and reuse the results if available. As such, they can achieve significant reuse among multiple calls to the solvers during the symbolic execution of different paths. GREEN achieves this by distilling constraints into their essential parts and representing them in a canonical form. The reuse achieved by GREEN is at a much lower level. GreenTrie [45] and Recal [46] are extensions to the Green framework, which support constraint reuse based on the logical implication relations among constraints. GreenTrie provides L-Trie to store constraints and solutions into tries. Recal proposed powerful simplifications and new canonical form of the constraints to reuse equivalent constraints in large repositories. As such, the reuse is orthogonal to the pruning by our method. Therefore, it would be interesting to see if GREEN and GreenTrie can be plugged into our symbolic execution framework to achieve more reduction—we leave this for future work. Reducing the size of constraints is another popular optimization approach of SAT/SMT solvers and symbolic executors [1], [4], [47]. For example, KLEE [1] does some constraint reductions before solving, such as expression rewriting, constraint set simplification and implied value concretization. As our work is building on KLEE, these reduction techniques have been integrated into our tool.

6.4 Reduction by State Merging

Recently several techniques for state merging [48], [49], [50], [51], [52] have been proposed to tackle the path-
explosion problem. The state merging reduction proposed by Kuznetsov et al. [48] was based on the idea of merging the forward reachable states obtained on different paths, which can lead to a decrease of the number of paths that need to be explored. Collingbourne et al. [51] use φ-node folding to replace control-flow forking with predicated select instructions in order to reduce the number of paths explored by symbolic execution. Bugrara and Engler [49] present a technique that attacks path explosion by eliminating paths that cannot reach new code before they are executed. However, the methods differ significantly from our work in that state merging is a reduction based on the forward paths (prefixes), whereas our method is a reduction based on the backward analysis (suffixes). In general, these techniques are orthogonal to our method and may be used together to complement each other.

MergePoint [50] alternates between path-based exploration of dynamic symbolic execution (DSE) and state-merging based exploration of static symbolic execution. The main idea of MergePoint is exploiting static symbolic execution to mitigate the difficulty of solving formulas, while alleviating the high overhead associated with a path-based DSE approach. So, MergePoint is orthogonal to our pruning method. MULTISE [52] proposes a technique for merging states incrementally during symbolic execution, without using auxiliary variables, which maps each variable to a set of guarded symbolic expressions. MULTISE is a new symbolic execution framework with abstract state representation and execution semantics using value summaries, and is completely different from DSE. However, our method is an effective reduction method based on DSE.

6.5 Other Symbolic Execution Research
There also exist techniques for quickly achieving structural coverage in symbolic execution [53], [54], [55], [56] or increasing the coverage of less-traveled paths [57], [58]. Another line of research in this domain is guided symbolic execution, such as reaching a statement [59], [60], [61], checking a rule [62], [63] and exploring the difference between versions of a program [64], [65], [66], [67]. These techniques differ from ours in that they are for specific target or application, and do not attempt to achieve the complete path coverage. Our method, in contrast, focuses on sound pruning techniques for achieving the complete path coverage.

7 CONCLUSIONS AND FUTURE WORK
We have presented a new redundancy removal method for symbolic execution, which can identify and eliminate path suffixes that are shared by multiple test runs. We have implemented a prototype software tool and evaluated it on real applications. Our experiments show that redundancy due to common path suffixes are abundant and widespread in practice, and our method is effective in eliminating redundant paths. However, the speedup in execution time is less impressive due to memory and computation overheads. In the future, we plan to more carefully examine the trade-offs between effective redundancy removal and the computational cost of detecting and eliminating such redundancy. We believe that heuristics based on static program analysis can make the pruning more efficient. In addition, we plan to develop parallel algorithms that speed up postconditioned symbolic execution.

8 ACKNOWLEDGMENTS
This work was supported in part by the National Science Foundation of China (NSFC) under grant 61472318, 61632015 and 61572481, and the National Science Foundation (NSF) under grants CCF-1149454, DGE-1522883, and CCF-1500024. Any opinions, findings, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES


