This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2659751, IEEE Transactions on Software Engineering

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 precondition 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 Π_{prec} as an additional parameter in addition to the program under test. Preconditioned symbolic execution only descends into program branches that satisfy Π_{prec}, 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 Π_{prec} to be false preconditioned symbolic execution terminates without

- *Q. Yi is with the National Engineering Research Center for Fundamental Software, Institute of Software and Graduate University, Chinese Academy of Sciences, Beijing, China.*
- *Z. Yang is a corresponding author. He is with Department of Computer Science, Western Michigan University, Kalamazoo, Michigan, USA. Email: zijiang.yang@wmich.edu*
- *S. Guo is with Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, Virginia, USA*
- *C. Wang is with Department of Computer Science, University of Southern California, Los Angeles, California, USA*
- *J. Liu is a corresponding author. He is with Key Laboratory of Network Assessment Technology and Beijing Key Laboratory of Network security technology, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China. Email: liujian6@iie.ac.cn*
- *C. Zhao is with the National Engineering Research Center for Fundamental Software and State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China.*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2659751, IEEE Transactions on Software Engineering

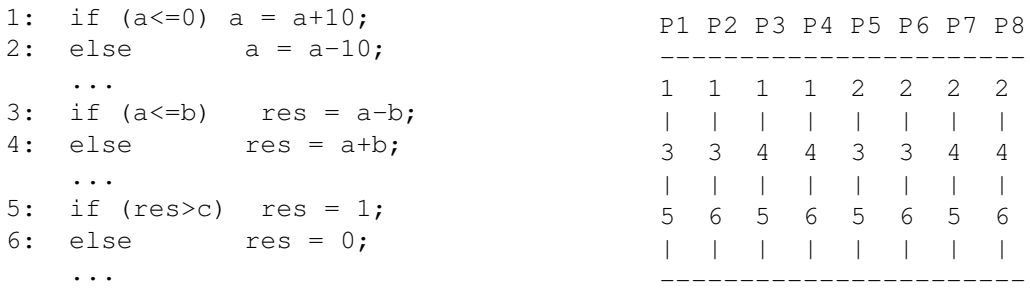

Fig. 1: A program with three branches and eight paths.

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 $\ldots \rightarrow 3 \rightarrow 5$ is shared by paths No. 1 and No. 5; and the suffix $\ldots \rightarrow 6$ 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 C 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 computational 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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2659751, IEEE Transactions on Software Engineering

a program P has a set Var of program variables and a set $Instr$ of instructions with the following syntactic categories:

e_a	$\in$	$AExp$	arithmetic expressions
e_b	$\in$	$BExp$	boolean expressions
$instr$	$\in$	$Instr$	instructions

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

a, b, c	$\in$	Var	variables
n	$\in$	Con	constants
op_a	$\in$	Op_a	arithmetic, pointer etc. operators
op_b	$\in$	Op_b	boolean operators
op_r	$\in$	Op_r	relational operators

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

$$
\begin{array}{rcl}
e_a & ::= & a \mid n \mid e_{a1}\ op_a\ e_{a2} \\
e_b & ::= & \mathsf{true} \mid \mathsf{false} \mid not\ e_b \mid e_{b1}\ op_b\ e_{b2} \\
 & & \mid e_{a1}\ op_r\ e_{a2} \\
instr & ::= & a := e_a \mid \textbf{skip} \mid \textbf{halt} \mid \textbf{abort} \mid instr_1\ ;\ instr_2 \\
 & & \mid if\ e_b\ then\ instr_1\ else\ instr_2 \\
 & & \mid while\ e_b\ do\ instr
\end{array}
$$

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:= 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 $if(\neg c)$.

With proper code transformations, the above instruction types are sufficient to represent the execution path of arbitrary C code. For example, if `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. ***Concrete 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 $instr \in Instr$ be an instruction in the program. An execution instance of $instr$ is called an event, denoted as $ev = \langle l, instr, l' \rangle$, 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 $if(\neg c)$, 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 $\pi = l_0 \xrightarrow{e_1} l_1 \xrightarrow{e_2} l_2 \ldots \xrightarrow{e_n} l_n$.*

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 $\pi = l_0 \xrightarrow{e_1} l_1 \xrightarrow{e_2} l_2 \ldots \xrightarrow{e_n} l_n$ be the concrete execution determined by t. If instruction s is in e_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 π, a suffix of π is a subsequence $\pi^i = l_i \xrightarrow{e_i} l_{i+1} \ldots \xrightarrow{e_n} l_n$, where $0 \leq i \leq n$. we also say π^i is reachable under t. Let $\mathcal{T}$ be a test input set of a program, we can easily extend the reachability of an instruction (or a suffix) under $\mathcal{T}$. Given a test input $t \in \mathcal{T}$, if an instruction s (or suffix π^i) is reachable under in t then we say that s (or suffix π^i) is reachable under $\mathcal{T}$.

If we denote the concrete execution by the pair (τ, π), 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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2659751, IEEE Transactions on Software Engineering

of a deterministic sequential program is sequence of instructions $\pi_s = l_0 \xrightarrow{instr_1} l_1 \xrightarrow{instr_2} l_2 \ldots \xrightarrow{instr_n} l_n$.

Definition 7. ***Symbolic memory state.*** *Let* $\pi_s = l_0 \xrightarrow{instr_1} l_1 \xrightarrow{instr_2} l_2 \ldots \xrightarrow{instr_n} l_n$ *be a symbolic execution, there is* symbolic memory state *on each location. A symbolic memory state is a mapping* $mem_s : Var \rightarrow AExp$. *For each program variable* $v \in Var$, *its symbolic value is represented by the expression* $mem_s[v]$. *Let* $l \xrightarrow{instr} l'$ *be an event in execution* π_s, mem_s *be symbolic memory state before the event and* mem'_s *be the following symbolic state. If* $instr$ *is* $v := exp$, *then* $mem'_s = mem[v \leftarrow exp]$, *otherwise* $mem'_s = mem_s$.

Definition 8. ***Path condition.*** *Let* $\pi_s = l_0 \xrightarrow{instr_1} l_1 \xrightarrow{instr_2} l_2 \ldots \xrightarrow{instr_n} l_n$ *be a symbolic execution, there is a* path condition *on each location. A path condition is boolean expression, denoted as* $pcon$. *Let* $l \xrightarrow{instr} l'$ *be an event in the execution* π_s, $pcon$ *be path condition before the event and* $pcon'$ *be the following one. If the* $instr$ *is* $if(c)$, *then* $pcon' = pcon \wedge c$, *otherwise* $pcon' = pcon$.

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 $l \xrightarrow{v:=exp} l'$, or two outgoing edges, each of which is of the form $l \xrightarrow{if(c)} l'$. The goal of symbolic execution is to compute a set $\mathcal{T}$ 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 π is a valid path of the program P under some test input, it should be able to generate test input $\tau \in \mathcal{T}$ that replays this path.

In this algorithm, a program state is represented by a tuple $\langle pcon, l, mem \rangle$, where $pcon$ is the path condition along an execution and l is a control location and mem is the symbolic memory map. Note that, for simplification, in the following sections we also use mem to denote symbolic memory states. The initial state is $\langle \mathsf{true}, l_{init}, mem_{init} \rangle$, meaning that the path condition $pcon$ is true and l_{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 $\langle pcon, l, mem \rangle$ 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 $pcon'$ 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 $l \xrightarrow{abort} l'$, the symbolic execution procedure finds a bug and terminates.
- If the event is $l \xrightarrow{halt} l'$, the symbolic execution path reaches the end.
- If the event is $l \xrightarrow{v:=exp} l'$, the new memory map mem' is computed by assigning exp to v in mem, denoted $mem[v \leftarrow exp]$.
- If the event is $l \xrightarrow{if(c)} l'$, the new path condition $pcon'$ is computed by conjoining $pcon$ with c, denoted $pcon \wedge c$.

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

Algorithm 1 StandardSymbolicExecution( )

```
1:  init_state ← ⟨true, l_init, mem_init⟩;
2:  stack.push( init_state );
3:  while ( stack is not empty )
4:      ⟨pcon, l, mem⟩ ← stack.pop();
5:      if ( pcon is satisfiable under mem )
6:          for each ( event l -instr-> l' )
7:              if ( instr is abort )
8:                  τ ← solve (pcon, mem);    //bug found
9:                  T := T ∪ {τ};
10:             else if ( instr is halt )
11:                 τ ← solve (pcon, mem);
12:                 T := T ∪ {τ};
13:             else if ( instr is if(c) )
14:                 next_state ← ⟨pcon ∧ c, l', mem⟩;
15:                 stack.push( next_state );
16:             else if ( instr is v := exp )
17:                 next_state ← ⟨pcon, l', mem[v ← exp]⟩;
18:                 stack.push( next_state );
19:             end if
20:         end for
21:     end if
22: end while
23: return T;
```

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2659751, IEEE Transactions on Software Engineering

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 ϕ, there is no need to continue beyond l if the ϕ is subsumed by $\Pi_{post}[l]$. The subsumption indicates that no program states reached after l along π 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 ϕ shown in Column 3, the new path condition ϕ' in Column 5 is the conjunction of ϕ with the negated postcondition $\neg\Pi_{post}[l]$ at location l, where $\Pi_{post}[l]$ summarizes all the explored path suffixes from location l and thus $\phi \wedge \neg\Pi_{post}[l]$ 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 ϕ', we assume that $\Pi_{post}[l] = \mathsf{false}$ for every location l. Therefore, the new path conditions at Lines 1, 3 and 5 remain the same; they are $(a \leq 0)$, $(a \leq 0) \wedge (a+10 \leq b)$ and $(a \leq 0) \wedge (a+10 \leq b) \wedge (a+10-b > c)$, respectively. A test input such as $a = -10, b = 1, c = -6$ can be computed by solving the path condition $(a \leq 0) \wedge (a+10 \leq b) \wedge (a+10-b > c)$ 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 $(a \leq b) \wedge (a - b > c)$, 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 $(a \leq 0) \wedge (a+10 \leq b) \wedge (a+10-b > c)$, 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 $\phi = (a \leq 0) \wedge (a + 10 \leq b) \wedge (a + 10 - b \leq c)$ at Line 5. In our new method, the path constraint should be $\phi' = \phi \wedge \neg(a + 10 - b > c)$, 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 $(res > c) \vee (res \leq c) \equiv \mathsf{true}$. We continue the computation backward to Line 3, where the postcondition is $(a \leq b) \wedge (a - b) \leq c$. Combining it with the previously computed postcondition, we have $((a \leq b) \wedge (a-b) > c) \vee ((a \leq b) \wedge (a-b) \leq c)$, which is the same as $(a \leq b)$ at Line 3. Similarly, the postcondition at Line 1 is updated to $(a \leq 0) \wedge (a + 10 \leq b)$.

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 $\phi' = \phi \wedge \neg\mathsf{true}$ is satisfiable. Here $\neg\mathsf{true}$ is the negation of the postcondition computed at the end of path No. 2. Since ϕ' 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 $(a \leq b) \vee (a > b)$, which is the same as true at Line 3. We compute the summary constraint $(a \leq 0) \wedge (a + 10 \leq b) \vee (a \leq 0) \wedge (a + 10 > b)$, 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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2659751, IEEE Transactions on Software Engineering

TABLE 1: Symbolic computation for the program in Figure 1

Path	Br No.	Path condition ϕ (original)	TestGen	Path condition ϕ' (with pruning)	TestGen	Postconditions
1	1	$(a \leq 0)$		$(a \leq 0)$		$(a \leq 0) \wedge (a + 10 \leq b) \wedge (a + 10 - b > c)$
	3	$(a \leq 0) \wedge (a + 10 \leq b)$		$(a \leq 0) \wedge (a + 10 \leq b)$		$(a \leq b) \wedge (a - b > c)$
	5	$(a \leq 0) \wedge (a + 10 \leq b) \wedge (a + 10 - b > c)$	SAT	$(a \leq 0) \wedge (a + 10 \leq b) \wedge (a + 10 - b > c)$	SAT	$(res > c)$
2	1	$(a \leq 0)$		$(a \leq 0)$		$(a \leq 0) \wedge (a + 10 \leq b)$
	3	$(a \leq 0) \wedge (a + 10 \leq b)$		$(a \leq 0) \wedge (a + 10 \leq b)$		$(a \leq b)$
	6	$(a \leq 0) \wedge (a + 10 \leq b) \wedge (a + 10 - b \leq c)$	SAT	$(a \leq 0) \wedge (a + 10 \leq b) \wedge (a + 10 - b \leq c) \wedge \neg(a + 10 - b > c)$	SAT	true
3	1	$(a \leq 0)$		$(a \leq 0)$		$(a \leq 0)$
	4	$(a \leq 0) \wedge (a + 10 > b)$		$(a \leq 0) \wedge (a + 10 > b)$	SAT	true
	5	$(a \leq 0) \wedge (a + 10 > b) \wedge (a + 10 + b > c)$	SAT	$(a \leq 0) \wedge (a + 10 > b) \wedge (a + 10 + b > c) \wedge \neg$true		true
4	1	$(a \leq 0)$			(skipped)	
	4	$(a \leq 0) \wedge (a + 10 > b)$				
	6	$(a \leq 0) \wedge (a + 10 > b) \wedge (a + 10 + b \leq c)$	SAT			
5	2	$(a > 0)$		$(a > 0)$	SAT	true
	3	$(a > 0) \wedge (a - 10 \leq b)$		$(a > 0) \wedge (a - 10 \leq b) \wedge \neg$true		true
	5	$(a > 0) \wedge (a - 10 \leq b) \wedge (a - 10 - b > c)$	SAT			true
6	2	$(a > 0)$			(skipped)	
	3	$(a > 0) \wedge (a - 10 \leq b)$				
	6	$(a > 0) \wedge (a - 10 \leq b) \wedge (a - 10 - b \leq c)$	SAT			
7	2	$(a > 0)$			(skipped)	
	4	$(a > 0) \wedge (a - 10 > b)$				
	5	$(a > 0) \wedge (a - 10 > b) \wedge (a - 10 + b > c)$	SAT			
8	2	$(a > 0)$			(skipped)	
	4	$(a > 0) \wedge (a - 10 > b)$				
	6	$(a > 0) \wedge (a - 10 > b) \wedge (a - 10 + b \leq c)$	SAT			

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 $\phi' = \phi \wedge \neg$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 $\Pi_{post}[\,]$ 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 $(pcon \wedge c) \rightarrow \Pi_{post}[l']$ holds under the current memory map at Line 19. In this case, the path condition $(pcon \wedge c)$ 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 **UpdatePostcondition()** at Lines 13, 17 and 22. When an instruction **abort** occurs, we invoke **UpdatePostcondition(**$\bot$, true, $stack.e$**)** at Line 13 so a weakest precondition computation can start from terminal node $\bot$ with the initial logic formula true. At the end of each execution path, when the current $instr$ is **halt**, 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'. In this case we invoke **UpdatePostcondition(**$l', \Pi_{post}[l'], stack.e$**)** so a weakest precondition computation can start from l' 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 $\Pi_{post}[l] =$ false for every control location l. Whenever a new test input is generated for the path π, we update $\Pi_{post}[l]$ for all control locations in π based on the weakest precondition computation along π 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 $wp[\bot] =$ true. If l is an internal location with an existing summary ϕ, initially $wp[l] = \phi$. The propagation of weakest precondition at l along $l \xrightarrow{instr} l'$ 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, $wp[l] = wp[l'][exp/v]$.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2659751, IEEE Transactions on Software Engineering

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 Π_post[l] be postedcondition of each statement ;
3:  for each l, Π_post[l] ← false;
4:  init_state ← ⟨true, l_init, mem_init⟩;
5:  stack.push( init_state );
6:  while ( stack is not empty )
7:      ⟨pcon, l, mem⟩ ← stack.pop();
8:      if ( pcon is satisfiable under mem )
9:          for each( event l --instr--> l' at location l )
10:             if ( instr is abort )
11:                 τ ← Solve ( pcon, mem ); //bug found
12:                 T := T ∪ {τ};
13:                 UpdatePostcondition(⊥, true, stack.e);
14:             else if ( instr is halt)
15:                 τ ← Solve ( pcon, mem );
16:                 T := T ∪ {τ};
17:                 UpdatePostcondition(⊥, true, stack.e);
18:             else if (instr is if(c) )
19:                 if (pcon ∧ c → Π_post[l'])
20:                     τ ← Solve ( pcon, mem );
21:                     T := T ∪ {τ};
22:                     UpdatePostcondition(l', Π_post[l'], stack.e);
23:                 else
24:                     next_state ← ⟨pcon ∧ c, l', mem⟩;
25:                     stack.push( next_state );
26:                 end if
27:             else if ( instr is v := exp )
28:                 next_state ← ⟨pcon, l', mem[v ← exp]⟩;
29:                 stack.push( next_state );
30:             end if
31:         end for
32:     end if
33: end while
34: return T;
```

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

The pseudocode for updating $\Pi_{post}[\ ]$ 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 $\Pi_{post}[l] = \Pi_{post}[l] \vee wp[l]$. 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:*

loc	instr	weakest precondition	rule applied
l_0	$\xrightarrow{if(a\leq 0)}$	$(a \leq 0) \wedge (a + 10 \leq b)$ $\wedge (a + 10 - b > c)$	$wp[l_1] \wedge c$
l_1	$\xrightarrow{a:=a+10}$	$(a + 10 \leq b)$ $\wedge (a + 10 - b > c)$	$wp[l_2][exp/v]$
l_2	$\xrightarrow{if(a\leq b)}$	$(a \leq b) \wedge (a - b > c)$	$wp[l_3] \wedge c$
l_3	$\xrightarrow{res:=a-b}$	$(a - b > c)$	$wp[l_4][exp/v]$
l_4	$\xrightarrow{if(rec>c)}$	$(res > c)$	$wp[l_5] \wedge c$
l_5	$\xrightarrow{res:=1}$	true	$wp[l_6][exp/v]$
l_6		true	terminal

Algorithm 3 UpdatePostcondition(l', ϕ, $path$)

```
1:  Let path = ⟨e_1, e_2, ..., e_n⟩ be the sequence of executed events;
2:  wp[l'] ← φ;
3:  while ( event = path.pop() exists )
4:      Let l --instr--> l' be the event;
5:      if ( instr is v := exp )
6:          wp[l] ← wp[l'][exp/v];
7:      else if ( instr is if(c) )
8:          wp[l] ← wp[l'] ∧ c;
9:          Π_post[l] ← Π_post[l'] ∨ wp[l];
10:     end if
11: end while
```

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 $pcon$ of current execution π has been computed. Here, $pcon$ denotes the set of program states that can be reached from some initial states via π. In particular, when state $S = \langle pcond, l, mem\rangle$ is reached and there exists $l \xrightarrow{if(c)} l'$, Algorithm 2 revises Algorithm 1 to enable path suffix elimination as follows:

- If $pcon \wedge c \rightarrow \Pi_{post}[l']$, 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 $pcon \wedge c \not\rightarrow \Pi_{post}[l']$, 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 $pcon \wedge c \rightarrow \Pi_{post}[l']$ can be decided using a constraint

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2659751, IEEE Transactions on Software Engineering

solver to check the satisfiability of its negation ($pcon \wedge c \wedge \neg\Pi_{post}[l']$).

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* $pcon = (a \leq 0) \wedge (a+10 > b)$. *The next instruction is* $if(res > c)$. *At the next control location* l', *the summary of explored path suffixes is* $\Pi_{post}[l'] =$ true. *To check whether* $(a \leq 0) \wedge (a + 10 > b) \wedge (a + 10 + b > c) \rightarrow$ true *holds, we check the satisfiability of its negation:* $(a \leq 0) \wedge (a+10 > b) \wedge (a+10+b > c) \wedge \neg$true. *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 $pcon = (a \leq 0) \wedge (a + 10 > b)$, *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 ($pcon \wedge c \wedge \neg\Pi_{post}[l']$), we have to update the variables in $\Pi_{post}[l']$ with their values as stored in memory mem. That is, $\Pi_{post}[l'] \leftarrow \forall_{x \in V}(\Pi_{post}[l'])[x/eval(x, S)]$, where V is the variable set of $\Pi_{post}[l']$.

Example 3. *Consider the following code snippet. Suppose*

```
0 x = SYMX, y = SYMY;
1 if (x > 0)
      //l1: sum1: y>=0 && x<=-5 && x<=0
2     y = -1;
3 else if (x > -5)
      //l2: sum2: y>=0 && x<=-5
4     y = 0;
 else
5     y = 1;
6 if (y >= 0)
      //l3: sum3: y>=0
7     return 1;
  else
8     return 0;
```

Fig. 2: Code snippet for explanation of subsumption check

the path executed first is $\langle 1, 3, 5, 6, 7\rangle$. *After its execution the summary at* $l3$ *becomes* $y \geq 0$. *When the second execution reaches at* $l3$ *with path prefix* $\langle 1, 3, 4\rangle$, *the current value of* y *is 0. Therefore* $\Pi'_{post}[l3]$ *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* $\mathcal{T}$ *and* $\mathcal{T}'$ *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* $\mathcal{T}$, *the suffix will also be reachable under* $\mathcal{T}'$.

Proof: We argue the correctness by contradiction. That is, there exists a suffix that is (1) reachable under $\mathcal{T}$, (2) but unreachable under $\mathcal{T}'$. Let $s = l_i \xrightarrow{e_i} \ldots \xrightarrow{e_{n-1}} l_n$ be a suffix of $\pi = l_0 \xrightarrow{e_0} \ldots l_{i-1} \xrightarrow{e_{i-1}} l_i \ldots \xrightarrow{e_{n-1}} l_n$ $(0 \leq i \leq n)$ by running a test input in $t \in \mathcal{T}$. We assume the opposite that s is not covered by any test input in $\mathcal{T}'$. Assume s_{max} is the longest subsequence from s that can be constructed by running any test input $t' \in \mathcal{T}'$. Then, s_{max} will be either as a form $l_i \xrightarrow{e_i} \ldots \xrightarrow{e_{j-1}} l_j$ $(i \leq j < n)$ 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), π 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 $v := exp$. According to Lines from 27 to 29 in Algorithm 2, $\pi = l_i \xrightarrow{e_i} \ldots \xrightarrow{e_i} l_{i+1}$ is also reachable under t'. This contradicts the assumption that s_{max} is the longest reachable subsequence from s under $\mathcal{T}'$.

Case III. e_j is a $if(c)$. 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 $pcon \wedge c \rightarrow \Pi_{post}[j+1]$. According to (2), $t' \in \mathcal{T}'$ is the test input, under which $l_i \xrightarrow{e_i} \ldots \xrightarrow{e_{j-1}} l_j$ 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 $pcon \wedge c \rightarrow \Pi_{post}[j+1]$, suffix $l_i \xrightarrow{e_i} \ldots \xrightarrow{e_{j-1}} l_j \xrightarrow{e_j} l_{j+1}$ is also reachable under t', which contradicts the assumption that s_{max} is the longest subsequence from s that is reachable under $\mathcal{T}'$.
- The second scenario is that $l_j \xrightarrow{e_j} l_{j+1}$ and $pcon \wedge c \not\rightarrow \Pi_{post}[j+1]$. According to (1), $pcon \wedge c$ is satisfiable under mem, and $pcon \wedge c \not\rightarrow \Pi_{post}[j+1]$. Thus there is a divergent path from l_{j+1} that has not been explored before. Lines 24 to 25 in Algorithm 2 pushes $\langle pcon \wedge c, l+1, mem\rangle$ into $stack$. It explores the sub-space under $l+1$ and generate other test inputs into $\mathcal{T}'$. Therefore $l_i \xrightarrow{e_i} \ldots \xrightarrow{e_{j-1}} l_j \xrightarrow{e_j} l_{j+1}$ is reachable under $\mathcal{T}'$, contradicting the assumption that s_{max} is the longest subsequence from s that is reachable under $\mathcal{T}'$.

If s_{max} is empty sequence, by considering e_{i-1}, similar contradictions can be deduced. □

Corollary 1. *For abort statement* s *in a program* P, *let* $\mathcal{T}$ *and* $\mathcal{T}'$ *be the test input sets generated by Algorithm 1 and*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2659751, IEEE Transactions on Software Engineering

Algorithm 2, respectively. If s is reachable under $\mathcal{T}$, it will be reachable under $\mathcal{T}'$ 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 $\Pi_{post}[\ ]$ 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 $\Pi_{post}[\ ]$ 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 $\Pi_{post}[\ ]$, 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 $\Pi_{post}[\ ]$ as well as the size of each logic constraint $\Pi_{post}[l]$ may be large. The summary $\Pi_{post}[l]$, 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 $\Pi_{post}[l]$ 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 $\Pi_{post}[l]$ can be used in Algorithm 3 to replace $\Pi_{post}[l]$ 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 underapproximation $\Pi^{-}_{post}[l]$ to replace $\Pi_{post}[l]$ can make the computation significantly cheaper. The reason why it is always safe to do so is that, by definition, we have $\Pi^{-}_{post}[l] \rightarrow \Pi_{post}[l]$. Therefore, if $(pcon \wedge c) \rightarrow \Pi^{-}_{post}[l]$ holds, so does $(pcon \wedge c) \rightarrow \Pi_{post}[l]$.

A main advantage of our new pruning method is that it allows for the use of (any kind of) under-approximation of the table $\Pi_{post}[\ ]$ 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 $\Pi_{post}[l]$. 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., $key(l) = key(l')$, 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 $\Pi_{post}[l]$. That is, we replace Line 9 in Algorithm 3 by the following statement:
if (**size**($\Pi_{post}[l]$) $<$ bd) $\Pi_{post}[l] \leftarrow \Pi_{post}[l] \vee wp$;

4.2 Summary Simplification

Algorithm 4 CombineSummary(S, d)

```
1:  for (each item d' ∈ S)
2:      flag ← false;
3:      if (d' and d are combinable)
4:          Let c and c' be the divergence root of d and d';
5:          d' ← remove c' from d';
6:          S ← S\{d'};
7:          return CombineSummary(S, d');
8:      end if
9:  end for
10: if (flag == false)
11:     S ← S ∪ {d};
12: end if
13: return S
```

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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2659751, IEEE Transactions on Software Engineering

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 $d = c_1 \wedge \ldots c_n \wedge c_{n+1}$ and $d' = c'_1 \wedge \ldots c'_n \wedge c'_{n+1}$ are *combinable* if (1) $c_i = c'_i (1 \leq i \leq n)$ and (2) $c_{n+1} \vee c'_{n+1} = \text{true}$. We call c_{n+1} and c'_{n+1} the *divergence roots* of d and d'. For instance, $d = c_1 \wedge c_2 \wedge \overline{c_3}$ and $d' = c_1 \wedge c_2 \wedge c_3$ are combinable and their divergence roots are $c3$ and $\overline{c3}$. Two disjunctive terms can be replaced by one by throwing away the divergence roots. In our example, d and d' are replaced by $c_1 \wedge c_2$.

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 straight-forward 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.

```
1:  if (x>0) {
2:      a++;
3:      if (a>0)
4:          b++;
5:      else
6:          b--;
7:  } else
8:      a--;

9:  if (y>0)
10:     c++, e++;
11:     if (c>0)
12:         d++;
13:     else
14:         d--;
15: } else
16:     e--;
17: assert(e > 0);
```

Fig. 3: A simple program with variable definitions omitted.

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

Sum_9	Sum'_9
I1: $e-1>0 \wedge y \leq 0$	I1: true
I2: $e-1 \leq 0 \wedge y \leq 0$	
I3: $(e>0 \vee e<0) \wedge c+1 \leq 0 \wedge y>0$	
I4: $(e>0 \vee e<0) \wedge c+1>0 \wedge y>0$	

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 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.

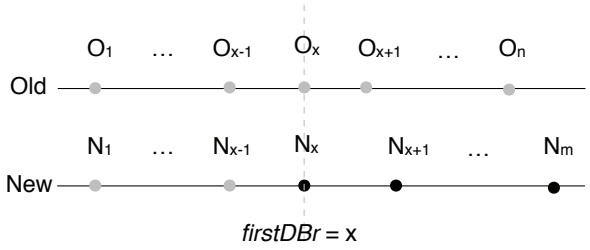

Fig. 4: Two adjacent paths: the previous path(Old) and the present one(New).

Figure 4 presents two adjacent paths Old and 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 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 Old that has already been computed and stored. Thus, the weakest precondition computation along path New only needs to propagate N_x to N_m along 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 $WP[p]^i_j$ 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.

$$\begin{aligned} WP[New][i] = &\quad WP_{old}[i] \wedge WP_{new}[i] \\ WP_{old}[i] = &\quad \wedge_{j=i}^{diffDBr-1} WP[Old]^i_j \\ WP_{new}[i] = &\quad \wedge_{j=max(i,diffDBr)}^{m} WP[Old]^i_j \end{aligned} \tag{1}$$

The new computation of the weakest precondition at the i-th branch contains two parts: $WP_{old}[i]$ and $WP_{new}[i]$, which respectively represent the reused part

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2659751, IEEE Transactions on Software Engineering

TABLE 2: Symbolic computation for the program in Figure 3. The bold items are the newly updated items.

path idx	path	Summary at Line 1	Summary at Line 1(after simplification)
1	1 9 17	**I1: $e-1>0 \wedge y \leq 0 \wedge x \leq 0$**	**I1: $e-1>0 \wedge y \leq 0 \wedge x \leq 0$**
2	1 9 17	I1: $e-1>0 \wedge y \leq 0 \wedge x \leq 0$ **I2: $e-1 \leq 0 \wedge y \leq 0 \wedge x \leq 0$**	I1: $e-1>0 \wedge y \leq 0 \wedge x \leq 0$ **I2: $e-1 \leq 0 \wedge y \leq 0 \wedge x \leq 0$** **$\Rightarrow$ I1: $y \leq 0 \wedge x \leq 0$**
3	1 9 11	I1: $e-1>0 \wedge y \leq 0 \wedge x \leq 0$ I2: $e-1 \leq 0 \wedge y \leq 0 \wedge x \leq 0$ **I3: $(e>0 \vee e<0 \wedge x \leq 0) \wedge c+1 \leq 0 \wedge y>0 \wedge x \leq 0$**	I1: $y \leq 0 \wedge x \leq 0$ **I2: $c+1 \leq 0 \wedge y>0 \wedge x \leq 0$**
4	1 9 11	I1: $e-1>0 \wedge y \leq 0 \wedge x \leq 0$ I2: $e-1 \leq 0 \wedge y \leq 0 \wedge x \leq 0$ I3: $(e>0 \vee e<0) \wedge c+1 \leq 0 \wedge y>0 \wedge x \leq 0$ **I4: $(e>0 \vee e<0) \wedge c+1>0 \wedge y>0 \wedge x \leq 0$**	I1: $y \leq 0 \wedge x \leq 0$ I2: $c+1 \leq 0 \wedge y>0 \wedge x \leq 0$ **I3: $(e>0 \vee e<0) \wedge c+1>0 \wedge y>0 \wedge x \leq 0$** **$\Rightarrow$ I1: $x \leq 0$**
5	1 3	I1: $e-1>0 \wedge y \leq 0 \wedge x \leq 0$ I2: $e-1 \leq 0 \wedge y \leq 0 \wedge x \leq 0$ I3: $(e>0 \vee e<0) \wedge c+1 \leq 0 \wedge y>0 \wedge x \leq 0$ I4: $(e>0 \vee e<0) \wedge c+1>0 \wedge y>0 \wedge x \leq 0$ **I5: $Sum_9 \wedge a \leq 0 \wedge x>0$**	I1: $x \leq 0$ **I2: $Sum'_9 \wedge a \leq 0 \wedge x>0 \Rightarrow a \leq 0 \wedge x>0$**
6	1 3	I1: $e-1>0 \wedge y \leq 0 \wedge x \leq 0$ I2: $e-1 \leq 0 \wedge y \leq 0 \wedge x \leq 0$ I3: $(e>0 \vee e<0) \wedge c+1 \leq 0 \wedge y>0 \wedge x \leq 0$ I4: $(e>0 \vee e<0) \wedge c+1>0 \wedge y>0 \wedge x \leq 0$ I5: $Sum_9 \wedge a \leq 0 \wedge x>0$ **I6: $Sum_9 \wedge a>0 \wedge x>0$**	I1: $x \leq 0$ I2: $a \leq 0 \wedge x>0$ **I3: $Sum'_9 \wedge a>0 \wedge x>0$** **$\Rightarrow$ true**

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:

```
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, $D^1 \wedge C$, $D^2 \wedge C^1 \wedge B$, $D^3 \wedge C^2 \wedge B^1 \wedge A$ 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: $\neg D$, $\neg D^1 \wedge C$, $\neg D^2 \wedge C^1 \wedge B$, $\neg D^3 \wedge C^2 \wedge B^1 \wedge A$, 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-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2659751, IEEE Transactions on Software Engineering

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

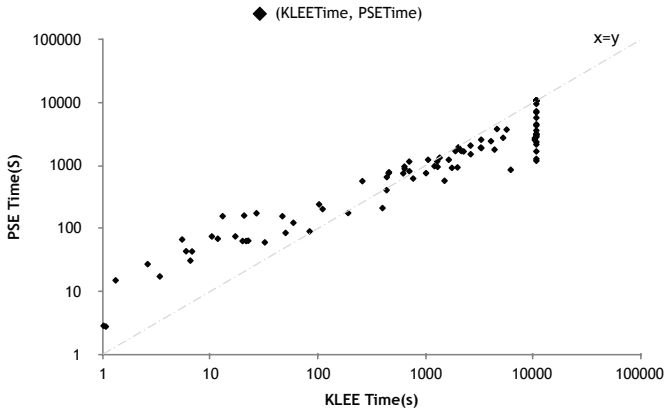

Fig. 5: KLEE v.s. postconditioned symbolic execution.

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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2659751, IEEE Transactions on Software Engineering

TABLE 4: Comparison between KLEE and PSE.

Test Program	#Explored Paths			#Explored Instructions			Time(s)		
Name	PSE	KLEE	KLEE/PSE	PSE	KLEE	KLEE/PSE	PSE	KLEE	Speedup in time
arch	855	1375	1.61X	196761	12230344	62.16X	936.77	1994.71	2.13X
base64	441	1058	2.40X	148328	9417607	63.49X	574.48	1511.49	2.63X
chcon	1398	3752	>2.68X	4380247	33297373	>7.60X	2963.82	TO	>3.64X
chgrp	1418	3920	>2.76X	17226071	35310651	>2.05X	4334.54	TO	>2.49X
chmod	1453	3661	>2.52X	11231299	33568918	>2.99X	2374.96	TO	>4.55X
chown	1672	3925	>2.35X	30090329	35531856	>1.18X	4524.18	TO	>2.39X
comm	885	2522	2.85X	303775	22936623	75.51X	2442.85	4087.59	1.67X
cp	1276	3636	>2.85X	2142868	35361275	>16.51X	2170.98	TO	>4.97X
csplit	930	3235	3.48X	1029691	31998354	31.08X	2560.72	10444.05	4.08X
dircolors	415	1178	2.84X	567953	13579980	23.91X	1237.65	1655.75	1.34X
dirname	220	564	2.56X	139147	4923714	35.38X	656.86	439.13	0.67X
du	297	949	3.20X	12256156	628662146	51.29X	1517.39	2646.26	1.74X
expand	415	762	1.84X	549802	7820299	14.22X	759.56	456.87	0.60X
expr	69	651	9.43X	96507	3919884	40.62X	212.01	400.52	1.89X
factor	1387	3812	>2.75X	12689346	36507426	>2.88X	6987.52	TO	>1.55X
fmt	206	792	3.84X	172154	6489860	37.70X	921.54	1770.91	1.92X
fold	423	967	2.29X	1753427	10143108	5.78X	1238.27	1063.06	0.86X
ginstall	1166	4911	4.21X	813196	48857663	60.08X	2729.9	10467.79	3.83X
head	1114	3721	>3.34X	11166367	59874383	>5.36X	5744.35	TO	>1.88X
hostid	855	1375	1.61X	142635	12278096	86.08X	1335.15	1360.31	1.02X
hostname	505	1375	2.72X	635920	13029976	20.49X	1167.23	1292.04	1.11X
id	423	1340	3.17X	25599403	54103365	2.11X	981.52	1218.79	1.24X
join	636	3699	>5.82X	2021675	33468346	>16.55X	3106.17	TO	>3.48X
link	1348	2748	>2.04X	13756342	32876548	>2.39X	7284.34	TO	>1.48X
ln	495	1510	3.05X	981995	19212957	19.57X	3729.76	5714.31	1.53X
logname	855	1375	1.61X	109812	12247184	111.53X	1945.39	2029.9	1.04X
ls	1275	3192	>2.50X	47579632	895733467	>18.83X	7785.94	TO	>1.39X
mkdir	1134	2783	>2.45X	1231570	33579827	>27.27X	3197.3	TO	>3.38X
mkfifo	867	2649	>3.06X	1108867	31240588	>28.17X	1194.03	TO	>9.04X
mknod	1109	1756	1.58X	2546180	15913533	6.25X	1678.84	2275.7	1.36X
mktemp	1102	3279	2.98X	3323201	31495317	9.48X	2764.32	5299.09	1.92X
mv	963	3446	>3.58X	1162161	34060036	>29.31X	2851.12	TO	>3.79X
nice	44	637	14.48X	290825	4894266	16.83X	759.57	1017.04	1.34X
nl	1100	2148	>1.95X	6204453	19874892	>3.20X	3127.31	TO	>3.45X
nohup	105	924	8.80X	120661	9846460	81.60X	752.84	624.17	0.83X
od	832	2851	3.43X	2931253	43443017	14.82X	1794.58	4409.97	2.46X
printenv	937	3663	3.91X	2217897	8047843	3.63X	2079.58	2637.18	1.27X
printf	1288	2014	>1.56X	2192027	21631625	>9.87X	1278.46	TO	>8.45X
ptx	532	1914	3.60X	1114112	46610662	41.84X	2579.68	3312.43	1.28X
readlink	400	745	1.86X	82000	5598617	68.28X	891.12	643.47	0.72X
rmdir	1134	3132	>2.76X	9368427	33400578	>3.57X	3609.24	TO	>2.99X
setuidgid	358	1848	5.16X	375051	32832492	87.54X	3811.23	4646.58	1.22X
shuf	457	1611	3.53X	191179	14473864	75.71X	857.33	6240.45	7.28X
sleep	855	3173	>3.71X	7422576	46972683	>6.33X	1683.14	TO	>6.42X
sort	396	1801	4.55X	524288	22398578	42.72X	1789.56	2114.36	1.18X
split	298	738	2.48X	162246	4421955	27.25X	784.08	460.14	0.59X
touch	305	1288	4.22X	205836	2337850	11.36X	954.32	1303.89	1.37X
tr	776	1690	2.18X	876831	16302848	18.59X	1897.87	3323.09	1.75X
tsort	381	580	1.52X	131983	5176989	39.22X	963.9	640.34	0.66X
tty	1193	1927	1.62X	1119129	20813531	18.60X	1933.18	3288.5	1.70X
uname	1013	2162	>2.13X	7842392	18587205	>2.37X	9535.2	TO	>1.13X
unexpand	259	812	3.14X	475743	8682733	18.25X	623.71	771.77	1.24X
uniq	368	939	2.55X	515168	8559681	16.62X	408.01	437.42	1.07X
unlink	477	1375	2.88X	654482	12939395	19.77X	1704.79	2164.11	1.27X
uptime	64	577	9.02X	451747	5475778	12.12X	1157.96	709.71	0.61X
users	378	577	1.53X	390157	5257283	13.47X	813.82	712.63	0.88X
whoami	855	1375	1.61X	302790	12437691	41.08X	1701.18	1909.14	1.12X
Average	-	-	**>3.34X**	-	-	**>23.03X**	-	-	**>2.26X**

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2659751, IEEE Transactions on Software Engineering

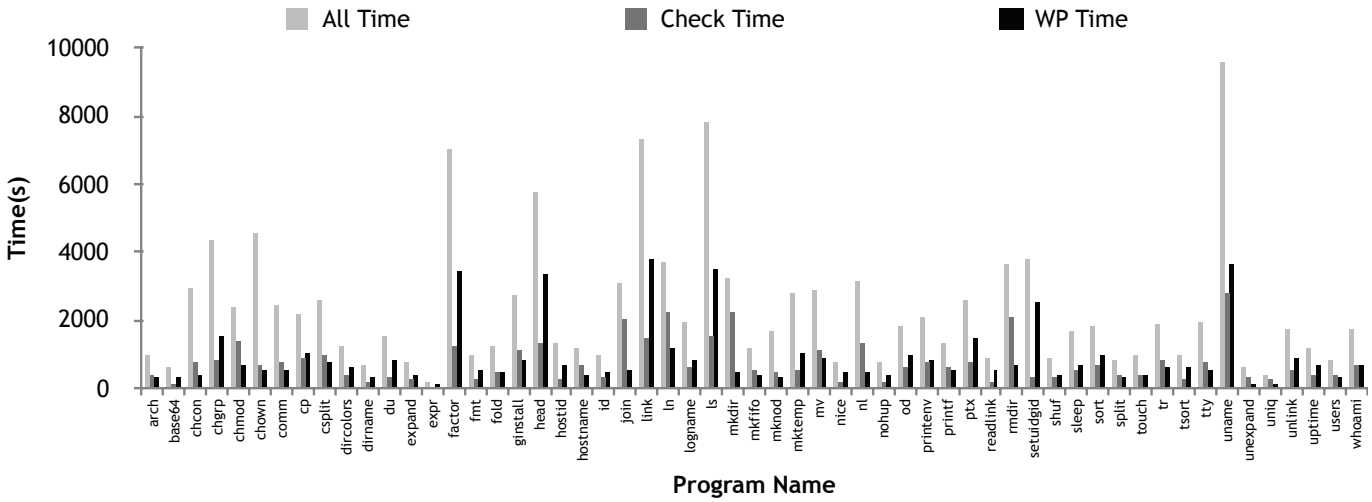

Fig. 6: Check time and weakest precodition computation time of PSE.

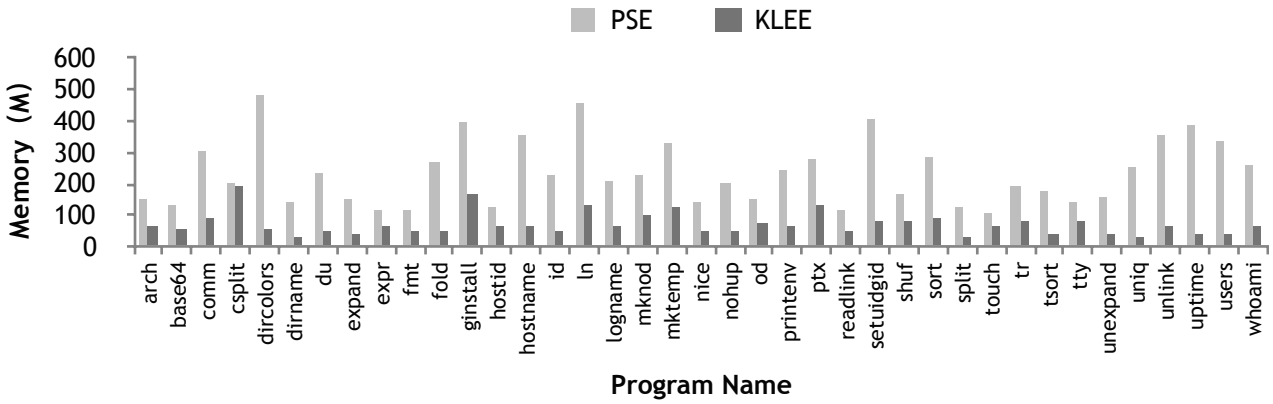

Fig. 7: Memory consumed by KLEE v.s. PSE.

the check time and wp time are the major source of the pruning overhead. On average, the majority of the time (70%) is spent on subsumption check and weakest precondition computation. These computations are not needed in standard symbolic execution. It is worth pointing out that, despite the large computational overhead, PSE has achieved considerable time speedup for large programs. Figure 7 compares the memory used by PSE and KLEE. On average, compared with KLEE, PSE needs about 1.87X more memory.

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 *factor*, *fold*, *join*, *ls*, *mkdir*, and *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 2.64X 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 *factor*, *head*,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2659751, IEEE Transactions on Software Engineering

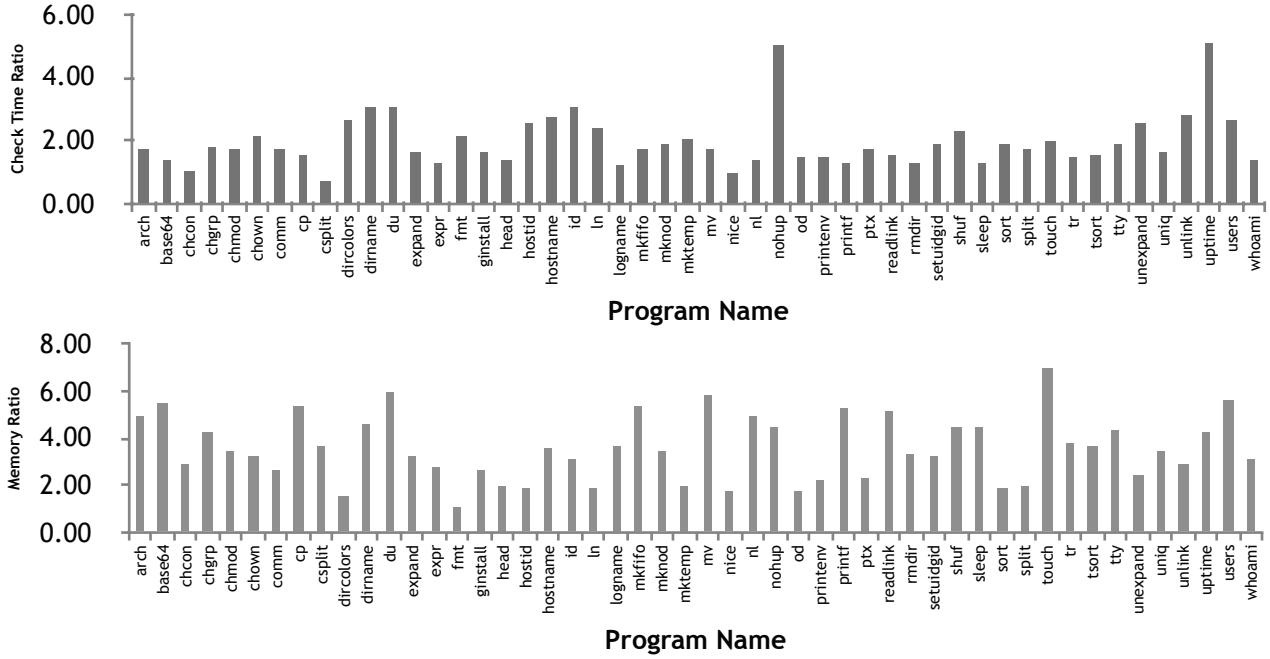

Fig. 8: Time usage and memory consumption ratio with and without under-approximation.

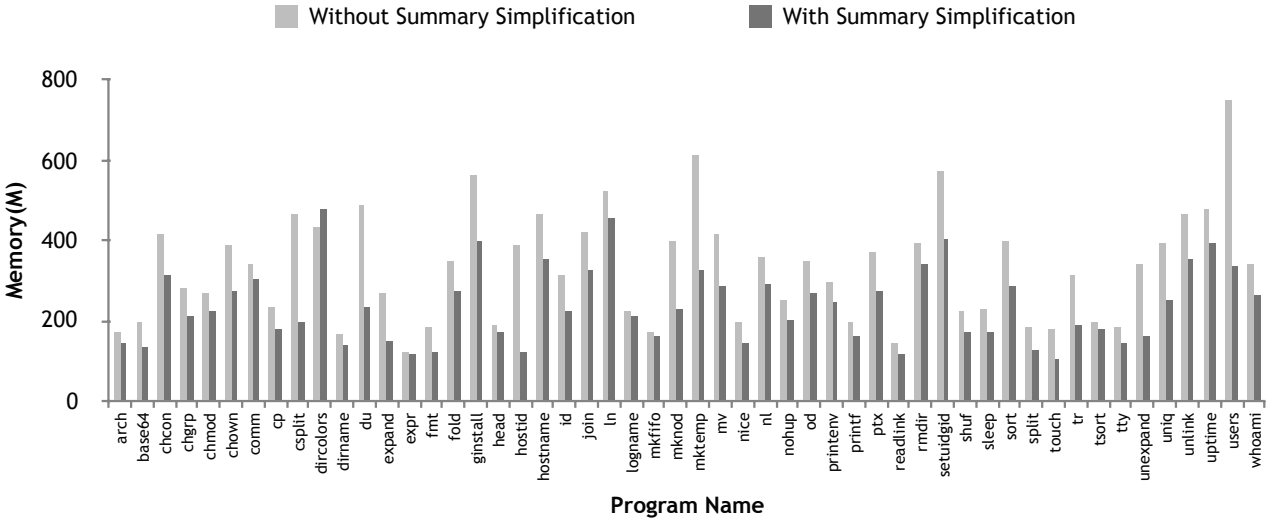

Fig. 9: Memory consumption with and without summary simplification.

ls, *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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2659751, IEEE Transactions on Software Engineering

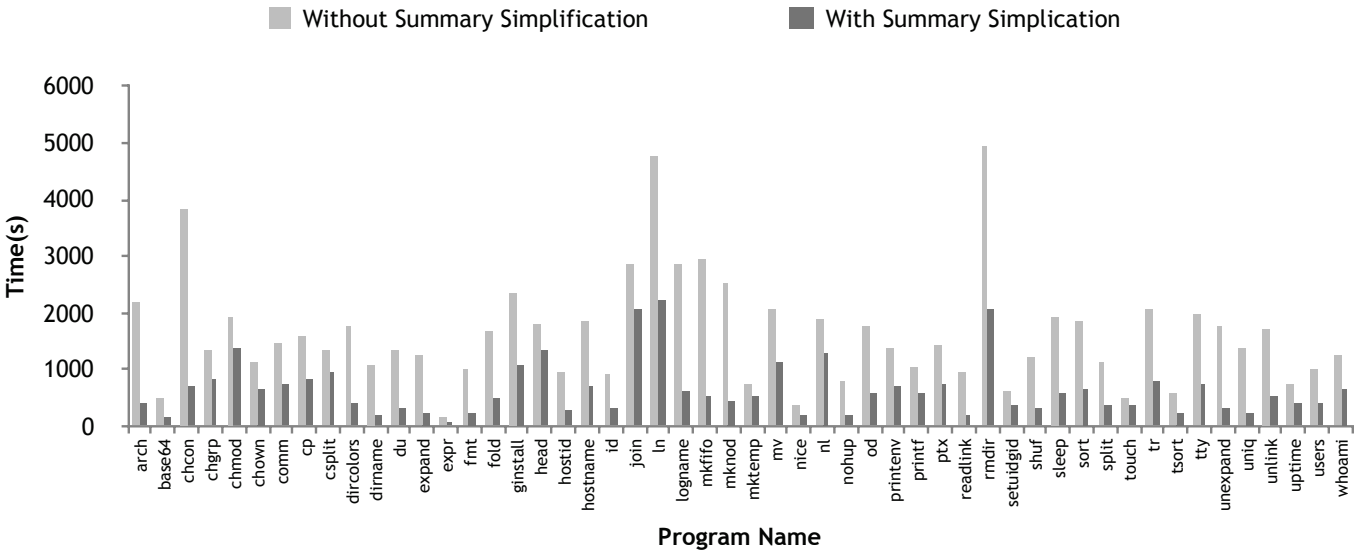

Fig. 10: Time usage for subsumption check with and without summary simplification.

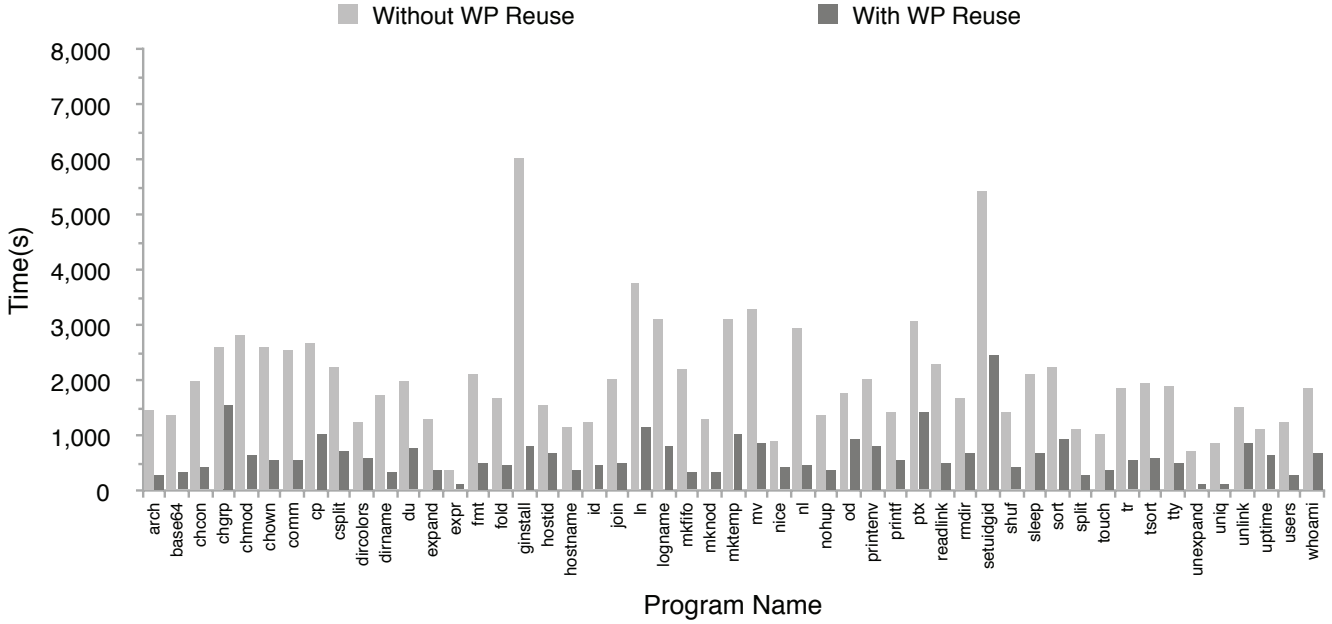

Fig. 11: Time usage for weakest precondition computation with and without reusing the result between two adjacent paths.

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 effects 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.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2659751, IEEE Transactions on Software Engineering

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($\mathcal{R}$) [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 memorization 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

[1] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *USENIX Symposium on Operating Systems Design and Implementation*, 2008, pp. 209–224.

[2] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing." in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jun. 2005, pp. 213–223.

[3] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *USENIX Symposium on Network and Distributed System Security*, 2008.

[4] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2005, pp. 263–272.

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

[6] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *ASE*, 2008, pp. 443–446.

[7] C. S. Pasareanu and W. Visser, "A survey of new trends in symbolic execution for software testing and analysis," *International Journal on Software Tools for Technology Transfer*, vol. 11, no. 4, pp. 339–353, 2009.

[8] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, "AEG: Automatic exploit generation," in *USENIX Symposium on Network and Distributed System Security*, Feb. 2011.

[9] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Active property checking," in *International Conference on Embedded Software*, 2008, pp. 207–216.

[10] E. Bounimova, P. Godefroid, and D. A. Molnar, "Billions and billions of constraints: whitebox fuzz testing in production," in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, 2013, pp. 122–131.

[11] D. A. Ramos and D. R. Engler, "Under-constrained symbolic execution: Correctness checking for real code," in *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, 2015, pp. 49–64.

[12] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: a platform for in-vivo multi-path analysis of software systems," in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011*, 2011, pp. 265–278.

[13] V. Pham, W. B. Ng, K. Rubinov, and A. Roychoudhury, "Hercules: Reproducing crashes in real-world application binaries," in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, 2015, pp. 891–901.

[14] P. Godefroid, "Compositional dynamic test generation," in *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2007, pp. 47–54.

[15] R. Majumdar and K. Sen, "Hybrid concolic testing," in *International Conference on Software Engineering*, 2007, pp. 416–426.

[16] K. L. McMillan, "Lazy annotation for program testing and verification," in *International Conference on Computer Aided Verification*, 2010, pp. 104–118.

[17] J. Jaffar, A. E. Santosa, and R. Voicu, "Efficient memoization for dynamic programming with ad-hoc constraints," in *AAAI*, 2008, pp. 297–303.

[18] J. Jaffar, A. Santosa, and R. Voicu, "An interpolation method for CLP traversal," in *International Conference on Principles and Practice of Constraint Programming*, 2009, pp. 454–469.

[19] D.-H. Chu and J. Jaffar, "A complete method for symmetry reduction in safety verification," in *International Conference on Computer Aided Verification*, 2012, pp. 616–633.

[20] C. S. Pasareanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. R. Lowry, S. Person, and M. Pape, "Combining unit-level symbolic execution and system-level concrete execution for testing NASA software," in *International Symposium on Software Testing and Analysis*, 2008, pp. 15–26.

[21] L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, and G. Candea, "Cloud9: a software testing service," *Operating Systems Review*, vol. 43, no. 4, pp. 5–10, 2009.

[22] M. Staats and C. S. Pasareanu, "Parallel symbolic execution for structural test generation," in *International Symposium on Software Testing and Analysis*, 2010, pp. 183–194.

[23] J. H. Siddiqui and S. Khurshid, "Scaling symbolic execution using ranged analysis," in *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, 2012, pp. 523–536.

[24] M. Kim, Y. Kim, and G. Rothermel, "A scalable distributed concolic testing approach: An empirical evaluation," in *ICST*, 2012, pp. 340–349.

[25] K. L. McMillan, "Lazy abstraction with interpolants." in *International Conference on Computer Aided Verification*. Springer, 2006, pp. 123–136, LNCS 4144.

[26] A. V. Nori, S. K. Rajamani, S. Tetali, and A. V. Thakur, "The yogi project: Software property checking via static analysis and testing," in *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, York, UK, March 22-29, 2009. Proceedings*, 2009, pp. 178–181.

[27] B. S. Gulavani, S. Chakraborty, A. V. Nori, and S. K. Rajamani, "Automatically refining abstract interpretations," in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, 2008, pp. 443–458.

[28] J. Jaffar, V. Murali, J. A. Navas, and A. E. Santosa, "TRACER: A symbolic execution tool for verification," in *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, 2012, pp. 758–766.

[29] D. Chu, J. Jaffar, and V. Murali, "Lazy symbolic execution for enhanced learning," in *Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings*, 2014, pp. 323–339.

[30] J. Jaffar, V. Murali, and J. A. Navas, "Boosting concolic testing via interpolation," in *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, 2013, pp. 48–58.

[31] J. Jaffar, J. A. Navas, and A. E. Santosa, "Unbounded symbolic execution for program verification," in *Runtime Verification - Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers*, 2011, pp. 396–411.

[32] D. Beyer and P. Wendler, "Algorithms for software model checking: Predicate abstraction vs. impact," in *Formal Methods in Computer-Aided Design, FMCAD 2012, Cambridge, UK, October 22-25, 2012*, 2012, pp. 106–113.

[33] D. Kroening and G. Weissenbacher, "Interpolation-based software verification with wolverine," in *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, 2011, pp. 573–578.

[34] G. Weissenbacher, D. Kroening, and S. Malik, "Wolverine: Battling bugs with interpolants - (competition contribution)," in *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, 2012, pp. 556–558.

[35] V. D'Silva, D. Kroening, M. Purandare, and G. Weissenbacher, "Interpolant strength," in *Verification, Model Checking, and Abstract Interpretation, 11th International Conference, VMCAI 2010, Madrid, Spain, January 17-19, 2010. Proceedings*, 2010, pp. 129–145.

[36] A. Albarghouthi, A. Gurfinkel, and M. Chechik, "Whale: An interpolation-based algorithm for inter-procedural verification," in *Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings*, 2012, pp. 39–55.

[37] A. Albarghouthi, Y. Li, A. Gurfinkel, and M. Chechik, "Ufo: A framework for abstraction- and interpolation-based software verification," in *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, 2012, pp. 672–678.

[38] J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. C. Yap, "The CLP(R) language and system," *ACM Trans. Program. Lang. Syst.*, vol. 14, no. 3, pp. 339–395, 1992.

[39] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, 2007, pp. 519–531.

[40] S. Anand, P. Godefroid, and N. Tillmann, "Demand-driven compositional symbolic execution," in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, 2008, pp. 367–381.

[41] P. Godefroid, A. V. Nori, S. K. Rajamani, and S. Tetali, "Compositional may-must program analysis: unleashing the power of alternation," in *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2010, pp. 43–56.

[42] J. M. Rojas and C. Păsăreanu, "Compositional symbolic execution through program specialization," in *8th Workshop on Bytecode Semantics, Verification, Analysis and Transformation, BYTECODE*, 2013.

[43] R. Qiu, G. Yang, C. S. Pasareanu, and S. Khurshid, "Compositional symbolic execution with memoized replay," in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, 2015, pp. 632–642.

[44] W. Visser, J. Geldenhuys, and M. B. Dwyer, "Green: reducing, reusing and recycling constraints in program analysis," in *ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2012, p. 58.

[45] X. Jia, C. Ghezzi, and S. Ying, "Enhancing reuse of constraint solutions to improve symbolic execution," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, 2015, pp. 177–187.

[46] A. Aquino, F. A. Bianchi, C. Meixian, G. Denaro, and M. Pezzè, "Reusing constraint proofs in program analysis," in *Proceedings of the International Symposium on Software Testing and Analysis*, ser. ISSTA '15. ACM, 2015, pp. 305–315.

[47] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: automatically generating inputs of death," in *Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006, Alexandria, VA, USA, Ioctober 30 - November 3, 2006*, 2006, pp. 322–335.

[48] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2012, pp. 193–204.

[49] S. Bugrara and D. R. Engler, "Redundant state detection for dynamic symbolic execution," in *2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013*, 2013, pp. 199–211.

[50] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, "Enhancing symbolic execution with veritesting," in *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, 2014, pp. 1083–1094.

[51] P. Collingbourne, C. Cadar, and P. H. J. Kelly, "Symbolic crosschecking of floating-point and SIMD code," in *Proceedings of the Sixth European conference on Computer systems, EuroSys 2011, Salzburg, Austria, April 10-13, 2011*, 2011, pp. 315–328.

[52] K. Sen, G. C. Necula, L. Gong, and W. Choi, "Multise: multi-path symbolic execution using value summaries," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, 2015, pp. 842–853.

[53] R. Pandita, T. Xie, N. Tillmann, and J. de Halleux, "Guided test generation for coverage criteria," in *IEEE International Confer-*

ence on Software Maintenance (ICSM 2010), September 12-18, 2010, Timisoara, Romania, 2010, pp. 1–10.

[54] X. Ge, K. Taneja, T. Xie, and N. Tillmann, "DyTa: dynamic symbolic execution guided with static verification results," in *International Conference on Software Engineering*, 2011, pp. 992–994.

[55] X. Xiao, S. Li, T. Xie, and N. Tillmann, "Characteristic studies of loop problems for structural test generation via symbolic execution," in *IEEE/ACM International Conference On Automated Software Engineering*, 2013, pp. 246–256.

[56] M. Baluda, G. Denaro, and M. Pezzè, "Bidirectional symbolic analysis for effective branch testing," *IEEE Transactions on Software Engineering*, vol. PP, no. 99, p. to appear, 2015.

[57] Y. Li, Z. Su, L. Wang, and X. Li, "Steering symbolic execution to less traveled paths," in *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, 2013, pp. 19–32.

[58] H. Seo and S. Kim, "How we get there: a context-guided search strategy in concolic testing," in *ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2014, pp. 413–424.

[59] K. Ma, Y. P. Khoo, J. S. Foster, and M. Hicks, "Directed symbolic execution," in *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings*, 2011, pp. 95–111.

[60] D. Babic, L. Martignoni, S. McCamant, and D. Song, "Statically-directed dynamic automated test generation," in *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*, 2011, pp. 12–22.

[61] C. Zamfir and G. Candea, "Execution synthesis: a technique for automated software debugging," in *Proceedings of the 5th European conference on Computer systems, EuroSys 2010, Paris, France, April 13-16, 2010*, 2010, pp. 321–334.

[62] H. Cui, G. Hu, J. Wu, and J. Yang, "Verifying systems rules using rule-directed symbolic execution," in *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, 2013, pp. 329–342.

[63] Y. Zhang, Z. Chen, J. Wang, W. Dong, and Z. Liu, "Regular property guided dynamic symbolic execution," in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, 2015, pp. 643–653.

[64] S. Person, G. Yang, N. Rungta, and S. Khurshid, "Directed incremental symbolic execution," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, 2011, pp. 504–515.

[65] P. D. Marinescu and C. Cadar, "make test-zesti: A symbolic execution solution for improving regression testing," in *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, 2012, pp. 716–726.

[66] G. Yang, S. Khurshid, and C. S. Pasareanu, "Memoise: a tool for memoized symbolic execution," in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, 2013, pp. 1343–1346.

[67] G. Yang, C. S. Pasareanu, and S. Khurshid, "Memoized symbolic execution," in *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*, 2012, pp. 144–154.