Postconditioned Symbolic Execution

Qiuping Yi[1,2], Zijiang Yang[3], Shengjian Guo[4], Chao Wang[4], Jian Liu[1], Chen Zhao[1]

[1] National Engineering Research Center for Fundamental Software, Institute of Software, Beijing, China
[2] University of Chinese Academy of Sciences, Beijing, China
[3] Department of Computer Science, Western Michigan University, Kalamazoo, Michigan, USA
[4] Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, Virginia, USA

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. We have implemented our method in the symbolic execution engine KLEE and conducted experiments on a large set 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.

I. 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 [1], [2], [3], [4], [5], [6]. 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. 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], where a predefined constraint Π_{prec} is passed 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. In contrast, our work aims at eliminating redundant paths without reducing the search space.

We propose a new method called postconditioned symbolic execution to mitigate path explosion, by identifying and then eliminating redundant path suffixes encountered during symbolic execution. Our method is based on the observation that many common path suffixes are shared among different test runs, and repeatedly exploring these common path suffixes is a main reason for path explosion. 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

Corresponding author: Zijiang Yang (email: zijiang.yang@wmich.edu; web: www.cs.wmich.edu/∼zijiang).

symbolic virtual machine [6] and evaluated it using a large set of C programs from the GNU Coreutils suite that 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 postconditioned symbolic execution method for identifying and eliminating common path suffixes to mitigate the path explosion problem.
- We implement a prototype software tool based on KLEE [6] 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 path as well as the execution time.

The remainder of this paper is organized as follows. We first establish the notation and review existing techniques in Section II. Then, we present our postconditioned symbolic execution method in Section III, followed by experimental results in Section IV. We review related work in Section V, and finally give our conclusions in Section VI.

II. Preliminaries

In this section, we review the classic algorithm for test case generation based on symbolic execution.

We consider a sequential program P with a set V of program variables and a set *Instr* of instructions. Let $V_{in} \subseteq V$ 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 framework [9] where all *detectable* failures are modeled using a special **abort** instruction. The reachability of **abort** indicates the occurrence of a runtime failure. For example, instruction `assert(c)` can be modeled as `if(!c)abort`, 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==null)abort;else t->k=5`. Consider that abort may appear anywhere in a sequential path, in general, detecting the failures requires the effective coverage of all valid execution paths.

Let $instr \in Instr$ be an instruction in the program. An execution instance of $instr$ is called an event, denoted $ev = \langle l, instr, l' \rangle$, where l and l' are the control locations before and after executing the instruction. 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.

An instruction $instr$ may have one of the following types:

- **halt**, representing the normal program termination;
- **abort**, representing the faulty program termination;
- assignment $v := exp$, where $v \in V$ and exp is an expression over the set V of program variables;
- branch $if(c)$, where c is a conditional expression over V and $if(c)$ represents the branch taken by the execution. The else-branch is represented by $if(\neg c)$.

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 [1], CUTE [3], or KLEE [6].

A *concrete* execution of a deterministic sequential program is fully determined by the test input. Let τ be a test input. Let path $\pi = l_0 \xrightarrow{e_1} l_1 \xrightarrow{e_2} l_2 \ldots \xrightarrow{e_n} l_n$ be the sequence of events executed under test input τ. A suffix of π is a subsequence $\pi^i = l_i \xrightarrow{e_i} l_{i+1} \ldots \xrightarrow{e_n} l_n$, where $0 \leq i \leq n$. If we denote the concrete execution by the pair (τ, π), then the corresponding symbolic execution is denoted $(*, \pi)$, where $*$ means that the test input is arbitrary.

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 memory map. For each program variable $v \in V$, its symbolic value is represented by the expression $mem[v]$. The initial state is $\langle true, l_{init}, mem_{init} \rangle$, 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'.

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:                  return ∅;   //BUG_FOUND;
9:              else if ( instr is halt )
10:                 τ ← solve (pcon, mem);
11:                 T := T ∪ {τ};
12:             else if ( instr is if(c) )
13:                 next_state ← ⟨pcon ∧ c, l', mem⟩;
14:                 stack.push( next_state );
15:             else if ( instr is v := exp )
16:                 next_state ← ⟨pcon, l', mem[v ← exp]⟩;
17:                 stack.push( next_state );
18:             end if
19:         end for
20:     end if
21: end while
22: return T;
```

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. Since the goal of software testing is to uncover bugs, once a path suffix is tested, there is no need to generate new test inputs to cover the path suffix again in the future.

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

III. Eliminating Redundancy Using Postconditioned Symbolic Execution

A. A Motivating Example

In this section, we illustrate the main idea behind postconditioned symbolic execution using an example. Consider the program in Figure 1, which has three input variables `a`, `b`, `c` and three consecutive `if-else` statements. The goal is 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. Since the three branching statements are independent from each other, there are $2^3 = 8$ distinct execution paths. Classic symbolic execution tools 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.

```
1:  if (a<=0) res = res+1;
2:  else      res = res-1;
    ...
3:  if (b<=0) res = res+2;
4:  else      res = res-2;
    ...
5:  if (c<=0) res = res+3;
6:  else      res = res-3;
     ...
```

	P1	P2	P3	P4	P5	P6	P7	P8
	1	1	1	1	2	2	2	2
	|	|	|	|	|	|	|	|
	3	3	4	4	3	3	4	4
	|	|	|	|	|	|	|	|
	5	6	5	6	5	6	5	6
	|	|	|	|	|	|	|	|

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

Clearly, the number of paths of a program, like the one in Figure 1, can be exponential in the number of branch conditions—the worst case is when the branch conditions are completely independent of each other. However, 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. Since the goal of testing is to uncover bugs, once a path suffix has been tested, we should not explore it again in the future.

Using the information shown in Table I, we first show how standard symbolic execution works. Then, we show how postconditioned symbolic execution works on the same example. This would explain the reason why our new method can achieve the same exhaustive path coverage, but needs only 4 out of the 8 test inputs computed by the standard method.

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—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 $(a \leq 0)$, $(a \leq 0) \wedge (b \leq 0)$ and $(a \leq 0) \wedge (b \leq 0) \wedge (c \leq 0)$, respectively. A test input such as $a = 0, b = 0, c = 0$ can be computed by solving the path condition $(a \leq 0) \wedge (b \leq 0) \wedge (c \leq 0)$ 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 $(c \leq 0)$. Similarly, for the branch at Line 3, we record the summary constraint $(b \leq 0) \wedge (c \leq 0)$, 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 $(a \leq 0) \wedge (b \leq 0) \wedge (c \leq 0)$, 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 $\phi = (a \leq 0) \wedge (b \leq 0) \wedge (c > 0)$ at Line 5. In our new method, the path constraint should be $\phi' = \phi \wedge \neg(c \leq 0)$, where $c \leq 0$ is the summary of the already explored path suffix. Since $\phi' \equiv \phi$, we do not gain anything by applying this reduction. A test input such as $a = 0, b = 0, c = 1$ 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 $(c \leq 0)$ has been explored. Furthermore, the branch characterized by $(c > 0)$ has been explored. Therefore, the combined postcondition at Line 5 becomes $(c \leq 0) \vee (c > 0) \equiv \mathsf{true}$. We propagate the result backward to Line 3, where the postcondition is $(b \leq 0) \wedge \mathsf{true}$, which is the same as $(b \leq 0)$. Combining it with the previously computed postcondition, we have $((b \leq 0) \wedge (c \leq 0)) \vee (b \leq 0)$, which is the same as $(b \leq 0)$ at Line 3. Similarly, the postcondition at Line 1 is updated to $(a \leq 0) \wedge (b \leq 0)$.

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, because continuing the execution would not lead to any new path. In this case, the test input computed by the solver by solving the path condition at Line 4 can be $a = 0, b = 1, c = *$, meaning it is immaterial whether Line 5 or Line 6 is executed (both branches have been explored before).

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

In our method, path No. 4 will be skipped.

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 $\phi' = \phi \wedge \neg\mathsf{true}$ 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 (all of them have been explored before).

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 because there are no data or control dependency between the conditional expressions in the branches. 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.

B. 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 $\Pi_{post}[\,]$, which maps a control location l in the execution path to the summary $\Pi_{post}[l]$ of all explored path suffixes originating from the location l. Such summary table enables early termination that leads to pruning of partial or whole paths. In additional to the case where *instr* is **halt** at Line 9, we also terminate the forward symbolic execution when $(pcon \wedge c) \rightarrow \Pi_{post}[l']$ holds under the current memory map at Line 14. 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 12 and 17. At the end of each execution path, when the current *instr* is **halt**, we invoke **UpdatePostcondition(**$\bot, \mathsf{true}$**)** at Line 12 so a weakest precondition computation can start from terminal node $\bot$ with the initial logic formula true. The second procedure call happens at Line 17 when the current path condition is subsumed by the postcondition at l'. In this case we invoke **UpdatePostcondition(**$l', \Pi_{post}[l']$**)** so a weakest precondition computation can start from l' with the initial logic formula being the current postcondition at l'. This new procedure, to be discussed in Section III-C, updates the summaries for all control locations along the current path.

C. Summarizing Common Path Suffixes

We construct the summaries for visited control locations incrementally. Initially $\Pi_{post}[l] = \mathsf{false}$ 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

TABLE I
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	$(a \leq 0)$		$(a \leq 0)$		$(a \leq 0) \wedge (b \leq 0) \wedge (c \leq 0)$
	3	$(a \leq 0) \wedge (b \leq 0)$		$(a \leq 0) \wedge (b \leq 0)$		$(b \leq 0) \wedge (c \leq 0)$
	5	$(a \leq 0) \wedge (b \leq 0) \wedge (c \leq 0)$	SAT	$(a \leq 0) \wedge (b \leq 0) \wedge (c \leq 0)$	SAT	$(c \leq 0)$
2	1	$(a \leq 0)$		$(a \leq 0)$		$(a \leq 0) \wedge (b \leq 0)$
	3	$(a \leq 0) \wedge (b \leq 0)$		$(a \leq 0) \wedge (b \leq 0)$		$(b \leq 0)$
	6	$(a \leq 0) \wedge (b \leq 0) \wedge (c > 0)$	SAT	$(a \leq 0) \wedge (b \leq 0) \wedge (c > 0) \wedge \neg(c \leq 0)$	SAT	true
3	1	$(a \leq 0)$		$(a \leq 0)$		$(a \leq 0)$
	4	$(a \leq 0) \wedge (b > 0)$		$(a \leq 0) \wedge (b > 0)$	SAT	true
	5	$(a \leq 0) \wedge (b > 0) \wedge (c \leq 0)$	SAT	$(a \leq 0) \wedge (b > 0) \wedge (c \leq 0) \wedge \neg$true		true
4	1	$(a \leq 0)$			(skipped)	
	4	$(a \leq 0) \wedge (b > 0)$				
	6	$(a \leq 0) \wedge (b > 0) \wedge (c > 0)$	SAT			
5	2	$(a > 0)$		$(a > 0)$	SAT	true
	3	$(a > 0) \wedge (b \leq 0)$		$(a > 0) \wedge (b \leq 0) \wedge \neg$true		true
	5	$(a > 0) \wedge (b \leq 0) \wedge (c \leq 0)$	SAT			true
6	2	$(a > 0)$			(skipped)	
	3	$(a > 0) \wedge (b \leq 0)$				
	6	$(a > 0) \wedge (b \leq 0) \wedge (c > 0)$	SAT			
7	2	$(a > 0)$			(skipped)	
	4	$(a > 0) \wedge (b > 0)$				
	5	$(a > 0) \wedge (b > 0) \wedge (c \leq 0)$	SAT			
8	2	$(a > 0)$			(skipped)	
	4	$(a > 0) \wedge (b > 0)$				
	6	$(a > 0) \wedge (b > 0) \wedge (c > 0)$	SAT			

Algorithm 2 PostconditionedSymbolicExecution( )

```
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' at location l )
7:              if ( instr is abort )
8:                  return ∅;   //BUG_FOUND;
9:              else if ( instr is halt )
10:                 τ ← Solve ( pcon, mem );
11:                 T := T ∪ {τ};
12:                 UpdatePostcondition ( ⊥, true);
13:             else if (instr is if(c) )
14:                 if (pcon ∧ c → Π_post[l'])
15:                     τ ← Solve ( pcon, mem );
16:                     T := T ∪ {τ};
17:                     UpdatePostcondition (l', Π_post[l']);
18:                 else
19:                     next_state ← ⟨pcon ∧ c, l'⟩;
20:                     stack.push( next_state );
21:                 end if
22:             else if ( instr is v := exp )
23:                 next_state ← ⟨pcon, l', mem[v ← exp]⟩;
24:                 stack.push( next_state );
25:             end if
26:         end for
27:     end if
28: end while
29: return T;
```

a logical constraint that characterizes the suffix starting from a location l of the current execution path. If l is a terminal node $\perp$, initially $wp[\perp] = \mathsf{true}$. If l is an internal node with an existing postcondition 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 node 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, $wp[l] = wp[l'][exp/v]$.
- For a node 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 III-A. At the end of executing path No. 1, we invoke ***UpdatePostcondition()***, which carries out the summary computation as follows:

location	instruction	weakest precondition	rule applied
l_0	$\xrightarrow{if(a \leq 0)}$	$(a \leq 0) \wedge (b \leq 0) \wedge (c \leq 0)$	$wp[l_1] \wedge c$
l_1	$\xrightarrow{res:=res+1}$	$(b \leq 0) \wedge (c \leq 0)$	$wp[l_2][exp/v]$
l_2	$\xrightarrow{if(b \leq 0)}$	$(b \leq 0) \wedge (c \leq 0)$	$wp[l_3] \wedge c$
l_3	$\xrightarrow{res:=res+2}$	$(c \leq 0)$	$wp[l_4][exp/v]$
l_4	$\xrightarrow{if(c \leq 0)}$	$(c \leq 0)$	$wp[l_5] \wedge c$
l_5	$\xrightarrow{res:=res+3}$	true	$wp[l_6][exp/v]$
l_6		true	terminal

Algorithm 3 UpdatePostcondition(l,ϕ)

```
1:  Let path be the stack 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
```

D. 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 14, 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.

Algorithm 2 revises Algorithm 1 to enable common 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 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 III-A. When executing path No. 3 in Table I symbolically up to Line 4, we have path condition $pcon = (a \leq 0) \wedge (b > 0)$. The next instruction is $if(c \leq 0)$. At the next control location l', the summary of explored path suffixes is $\Pi_{post}[l'] = \mathsf{true}$. To check whether $(a \leq 0) \wedge (b > 0) \wedge (c \leq 0) \rightarrow \mathsf{true}$ holds, we check the satisfiability of its negation: $(a \leq 0) \wedge (b > 0) \wedge (c \leq 0) \wedge \neg\mathsf{true}$. Obviously it is unsatisfiable as the formula is equivalent to false.

While running Algorithm 2, we would go inside the if-branch at Line 14. By invoking the constraint solver on $pcon = (a \leq 0) \wedge (b > 0)$, we can compute a test input such as $a = 0, b = 1, c = *$, 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.

E. 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 common 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 common path 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 III-A.

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.

F. Controlling the Cost of Pruning

In our implementation, the size of the table $\Pi_{post}[\ ]$ as well as the size of each logic constraint $\Pi_{post}[l]$ may be large, e.g., when the execution paths are long and many. 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.

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 $(pcon \wedge c) \rightarrow \Pi_{post}[l]$.

In practice, we make two types of under-approximations.

- We use a hash table with a fixed number of entries to limit the storage cost for $\Pi_{post}[l]$. With a bounded table, however, two global control locations l and l' may be mapped to the same hash table entry. In this case, we 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).
- We use a fixed threshold to bound the size of the individual logic formulas of $\Pi_{post}[l]$. That is, we replace Line 9 in Algorithm 3 by the following statement:
  if ($|\Pi_{post}[l]| <$ threshold) $\Pi_{post}[l] \leftarrow \Pi_{post}[l] \vee wp[l]$;

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.

IV. Experiments

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

- How much redundancy is there in real-world applications due to the common path suffixes?
- How much computational overhead does the pruning method have?
- With such computational overhead, are there net benefits for applying the 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.

A. Subjects and Methodology

We have conducted experiments on a large set of C programs from the GNU Coreutils suite, which implements 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.

B. Effectiveness of Pruning

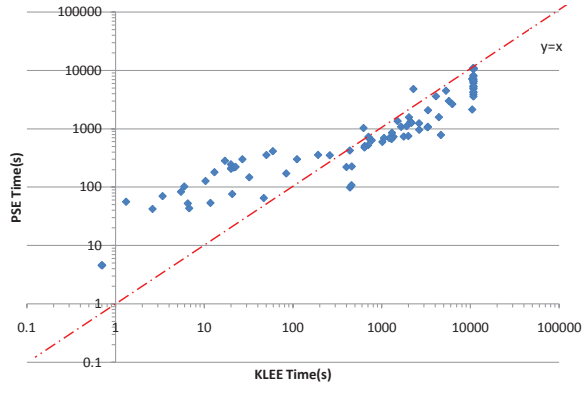

Fig. 2. KLEE v.s. postconditioned symbolic execution (PSE).

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 3 hour (10800 seconds).

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 2 shows the scatter diagram that compares the performance of postconditioned symbolic execution (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. This figure clearly indicates that the standard symbolic execution is more efficient when a benchmark is small, and postconditioned symbolic execution starts to win where 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 IV-C. Due to page limit and the interest in the cases when applying standard symbolic execution is challenging, for the rest of the section we only gives the experimental results of the 55 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 II. Column 1 lists the names of the benchmarks. Columns 2 to 4 show the number of paths explored, the number of instructions executed, and the time usage in seconds by KLEE. Columns 5 to 7 show the corresponding types of data for postconditioned symbolic execution. The last three columns show 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 postconditioned symbolic execution over KLEE. With pruning, postconditioned symbolic execution, the number of paths required to achieve exhaustive coverage is reduced by about 4.3X. That is, on average more than 70% 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, postconditioned symbolic execution reduces the executed instructions by about 14.5X. The results confirm our conjecture that redundancy due to common sub paths 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 15 benchmarks that KLEE cannot complete within the three hour time whereas our method can, the average speedup achieved by postconditioned symbolic execution is 2.0X. This is in contrast with 4.3X and 14.5X in the path and instruction reductions. In Section IV-C, we give the reasons.

C. 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 computa-

TABLE II
COMPARING KLEE WITH POSTCONDITIONED SYMBOLIC EXECUTION.

Test Program	Standard Symbolic Execution (KLEE)			Postconditioned Symbolic Execution			Performance Improvement		
Name	Explored paths	Explored insts	Time (sec)	Explored paths	Explored insts	Time (sec)	Path ratio	Inst ratio	Speedup in time
arch	1,375	12,230,344	1994.71	246	1,855,846	752.15	5.6X	6.6X	2.7X
base64	1,058	941,7607	1511.49	540	4,019,057	1354.56	2.0X	2.3X	1.1X
chcon	-	-	>3h	1,227	7,915,005	5336.64	-	-	2.0X
chmod	-	-	>3h	1,045	4,233,560	8198.14	-	-	1.3X
comm	2,522	22,936,623	4087.59	1,217	10,112,737	3607.04	2.1X	2.3X	1.1X
cp	-	-	>3h	1,236	11,813,098	4245.87	-	-	2.5X
csplit	3,235	31,998,354	10444.05	1,763	13,899,960	7154.76	1.8X	2.3X	1.5X
dircolors	1,178	13,579,980	1655.75	237	1,059,862	1081.14	5.0X	12.8X	1.5X
dirname	564	4,923,714	439.13	124	80,772	98.58	4.5X	61.0X	4.5X
du	949	628,662,146	2646.26	166	12,362,212	962.24	5.7X	50.9X	2.8X
expand	762	7,820,299	456.87	83	672,604	107.92	9.2X	11.6X	4.2X
expr	651	3,919,884	400.52	136	62,346	221.13	4.8X	62.9X	1.8X
factor	-	-	>3h	1,260	35,709,256	3878.14	-	-	2.8X
fmt	792	6,489,860	1770.91	306	2,777,479	736.24	2.6X	2.3X	2.4X
fold	967	10,143,108	1063.06	115	1,397,060	697.08	8.4X	7.3X	1.5X
ginstall	4,911	48,857,663	10467.79	407	710,212	2141.11	12.1X	68.8X	4.9X
head	-	-	>3h	1,285	42,104,790	5019.37	-	-	2.2X
hostid	1,375	12,278,096	1360.31	255	2,824,828	731.23	5.4X	4.3X	1.9X
hostname	1,375	13,029,976	1292.04	238	2,986,780	667.59	5.8X	4.4X	1.9X
id	1,340	54,103,365	1218.79	214	4,163,970	691.93	6.3X	13.0X	1.8X
join	-	-	>3h	1,256	31,915,817	6445.28	-	-	1.7X
link	-	-	>3h	1,328	42,390,898	6845.52	-	-	1.6X
ln	1,510	19,212,957	5714.31	238	894,724	2998.6	6.3	21.5X	1.9X
logname	1,375	12,247,184	2029.9	246	1,847,189	1576.5	5.6X	6.6X	1.3X
ls	-	-	>3h	735	121,864,375	3583.25	-	-	3.0X
mkdir	-	-	>3h	903	7,388,596	4855.76	-	-	2.2X
mkfifo	-	-	>3h	967	8,538,955	4252.86	-	-	2.5X
mknod	1,756	15,913,533	2275.7	824	7,958,409	4814.71	2.1X	2.0X	0.5X
mktemp	3,279	31,495,317	5299.09	1,046	11,084,508	4496.36	3.1X	2.8X	1.2X
mv	-	-	>3h	1,137	21,881,541	6077.27	-	-	1.8X
nice	637	4,894,266	1017.04	144	420,973	597.13	4.4X	11.6X	1.7X
nl	-	-	>3h	1,164	9,486,519	7914.07	-	-	1.4X
nohup	924	9,846,460	624.17	822	8,517,887	1030.96	1.1X	1.2X	0.6X
od	2,851	43,443,017	4409.97	745	7,691,252	1584.71	3.8X	5.6X	2.8X
printenv	3,663	8,047,843	2637.18	924	2,118,129	1243.32	4.0X	3.8X	2.1X
printf	-	-	>3h	1,347	11,438,582	7119.41	-	-	1.5X
ptx	1,914	46,610,662	3312.43	502	15,765,823	1081.34	3.8X	3.0X	3.1X
readlink	745	5,598,617	643.47	585	3,963,112	506.84	1.3X	1.4X	1.3X
rm	482	7,340,679	702.99	138	267,419	524.23	3.5X	27.5X	1.3X
setuidgid	1,848	32,832,492	4646.58	363	661,059	784.72	5.1X	49.7X	5.9X
shuf	1,611	14,473,864	6240.45	580	4,152,250	2653.32	2.8X	3.5X	2.4X
sleep	-	-	>3h	992	12,346,738	6474.45	-	-	1.7X
sort	1,801	22,398,578	2114.36	537	2,703,899	1326.72	3.4X	8.3X	1.6X
split	738	4,421,955	460.14	228	108,056	226.37	3.2X	40.9X	2.0X
touch	1,288	2,337,850	1303.89	383	1,559,711	854.31	3.4X	1.5X	1.5X
tr	1,690	16,302,848	3323.09	789	4,855,035	2080.26	2.1X	3.4X	1.6X
tsort	580	5,176,989	640.34	116	932,539	478.01	5.0X	5.6X	1.3X
tty	1,927	20,813,531	3288.5	747	5,515,712	1062.27	2.6X	3.8X	3.1X
uname	-	-	>3h	1,255	11,873,393	5359.82	-	-	2.0X
unexpand	812	8,682,733	771.77	155	1,144,891	635.85	5.2X	7.6X	1.2X
uniq	939	8,559,681	437.42	355	402,476	424.86	2.6X	21.3X	1.0X
unlink	1,375	12,939,395	2164.11	615	10,270,941	1266.16	2.2X	1.3X	1.7X
uptime	577	5,475,778	709.71	183	908,314	729.13	3.2X	6.0X	1.0X
users	577	5,257,283	712.63	171	921,753	695.57	3.4X	5.7X	1.0X
whoami	1,375	12,437,691	1909.14	152	606,472	1108.31	9.0X	20.5X	1.7X
Average	-	-	-	-	-	-	**4.3X**	**14.5X**	**>2.0X**

tional 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.

Table III shows the pruning overhead. Column 1 lists the benchmark names. Columns 2 to 4 compare the memory usage between KLEE and our approach. On average our approach uses four times more memory than KLEE. The last four columns (Columns 5 to 8) show the time breakdown of postconditioned symbolic execution. Column 5 and 6 show the time spent on subsumption check and weakest precondition computation, respectively. Column 8 shows the percentage of the pruning overhead time against the total time given in Column 7. On average, the majority of the time (60%) 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, postconditioned symbolic execution has led to considerable time speedup for large programs.

V. RELATED WORK

As we have mentioned earlier, there is a large body of work on test input generation based on symbolic execution [1], [2], [3], [4], [5], [6]. 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 [10], abstraction-refinement [11], interpolation [12], [13], [14], [15], and parallelization [16], [17], [18], [19], [20], path explosion remains a bottleneck in scaling symbolic execution to larger applications.

McMillan proposed a redundancy removal method for symbolic execution, called *lazy annotation* [12]. The method computes an interpolant from an unsatisfiable formula due to

TABLE III
PRUNING OVERHEAD IN POSTCONDITIONED SYMBOLIC EXECUTION.

Test Program	Memory			Time Breakdown of Postconditioned Symbolic Execution			
Name	KLEE	Postconditioned SE	Postconditioned SE/KLEE	Check time(CT)	WP time(WT)	All time(AT)	(CT+WT)/AT
arch	64	331	5.2X	33.02	537.2	752.15	0.8
base64	55	216	3.9X	387.22	404.43	1354.56	0.6
chcon	-	618	-	2358.57	654.74	5336.64	0.6
chmod	-	289	-	1515.52	1678.66	8198.14	0.4
comm	95	463	4.9X	523.56	618.24	3607.04	0.3
cp	-	634	-	651.25	826.31	4245.87	0.3
csplit	194	958	4.9X	1072.99	835.48	7154.76	0.3
dircolors	60	583	9.7X	56.26	723.65	1081.14	0.7
dirname	28	103	3.7X	13.21	60.86	98.58	0.8
du	52	451	8.7X	218.25	354.01	962.24	0.6
expand	37	115	3.1X	21.35	59.51	107.92	0.7
expr	66	281	4.3X	66.34	115.43	221.13	0.8
factor	-	176	-	826.31	1186.79	3878.14	0.5
fmt	46	316	6.9X	132.36	309.18	736.24	0.6
fold	45	266	5.9X	64.72	416.26	697.08	0.7
ginstall	170	341	2.0X	882.17	581.87	2141.11	0.7
head	-	765	-	836.28	1795.57	5019.37	0.5
hostid	68	145	2.1X	308.52	113.91	731.23	0.6
hostname	64	141	2.2X	260.43	121.75	667.59	0.6
id	49	267	5.5X	147.81	326.25	691.93	0.7
join	-	811	-	1103.98	1881.47	6445.28	0.5
link	-	902	-	1172.65	1134.12	6845.52	0.3
ln	136	300	2.2X	1246.61	742.01	2998.6	0.7
logname	63	338	5.4X	60.24	1182.93	1576.5	0.8
ls	-	324	-	1362.74	1343.26	3583.25	0.8
mkdir	-	979	-	1497.52	1573.77	4855.76	0.6
mkfifo	-	856	-	1242.23	1206.85	4252.86	0.6
mknod	98	712	7.3X	1481.27	1548.31	4814.71	0.6
mktemp	125	532	4.3X	1123.12	599.01	4496.36	0.4
mv	-	792	-	1138.25	1761.15	6077.27	0.5
nice	45	227	5.0X	101.46	322.46	597.13	0.7
nl	-	881	-	1562.86	2774.95	7914.07	0.5
nohup	46	119	2.6X	146.53	360.87	1030.96	0.5
od	73	451	6.2	625.96	780.19	1584.71	0.9
printenv	63	180	2.9X	157.23	607.24	1243.32	0.6
printf	-	581	-	1391.78	3230.31	7119.41	0.6
ptx	131	428	3.3X	162.74	482.31	1081.34	0.6
readlink	45	267	5.9X	157.81	206.26	506.84	0.7
rm	38	224	5.9X	98.37	272.17	524.23	0.7
setuidgid	83	217	2.6X	116.34	520.59	784.72	0.8
shuf	85	704	8.3X	486.68	1365.03	2653.32	0.7
sleep	-	539	-	1116.35	2631.89	6474.45	0.6
sort	94	750	8.0X	217.54	656.43	1326.72	0.7
split	33	75	2.3X	26.98	121.03	226.37	0.7
touch	67	506	7.6X	43.79	474.87	854.31	0.6
tr	92	767	8.3X	388.64	1063.78	2080.26	0.7
tsort	39	83	2.1X	86.14	208.12	478.01	0.6
tty	82	188	2.3X	120.25	518.39	1062.27	0.6
uname	-	764	-	1378.57	1863.06	5359.82	0.6
unexpand	38	249	6.6X	112.45	365.97	635.85	0.8
uniq	34	104	3.1X	157.63	106.24	424.86	0.6
unlink	62	572	9.2X	114.85	798.95	1266.16	0.7
uptime	42	183	4.4X	83.35	422.12	729.13	0.7
users	38	215	5.7X	89.26	356.78	695.57	0.6
whoami	63	184	2.9X	80.12	616.82	1108.3	0.6
Average	-	-	**4.9X**	-	-	-	**0.6**

the unreachability of certain branch conditions in a program. The interpolant can be regarded as an *over-approximated* set of forward reachable states. Jaffar *et al.* [13], [14], [15] proposed a similar method in the context of dynamic programming, for computing resource-constrained shortest paths and analyzing the worst-case execution time. Although interpolant is more general than weakest precondition, it is also more expensive to compute and requires special constraint solvers.

There are also pruning methods based on computing summaries. For example, Godefroid [10] 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. Majumdar and Sen [11] proposed a demand-driven abstraction-refinement style hybrid concolic testing algorithm, which can achieve a similar reduction. Godefroid et al. [21] proposed a compositional may-must program analysis to speed up symbolic execution using the result from over-approximated analysis and vice versa. However, our new method is significantly different in that our common path suffix elimination method is not restricted to the function boundary, and does not need the abstract-refinement loop.

There also exist techniques for quickly achieving structural coverage in symbolic execution [22], [23], [24] or increasing the coverage of less-traveled paths [25], [26]. These techniques differ from ours in that they do not attempt to achieve the complete path coverage. Our method, in contrast, focus on sound pruning techniques for achieving the complete path coverage.

The GREEN tool [27] 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. As such, the reuse is orthogonal to the pruning by our method. Therefore, it would be interesting to see if GREEN can be plugged into our distributed parallel symbolic execution framework to achieve more reduction—we leave this for future work.

The state merging reduction proposed by Kuznetsov *et al.* [28] 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. However, the method differs 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, which computes the summary of path suffixes. In general, these two techniques are orthogonal and may be used together to complement each other.

VI. Conclusions and Future Work

We have presented a new redundancy removal method for symbolic execution, which can identify and eliminate common 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.

VII. Acknowledgments

This work was supported in part by the National Science Foundation of China (NSFC) under grant 61472318, the National Science and Technology Major Project under Grant 2012ZX01039-004, and the National Science Foundation (NSF) under grants CCF-1149454, CCF-1500365, 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] 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.

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

[3] 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.

[4] 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.

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

[6] 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.

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

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

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

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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

[21] 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.

[22] R. Pandita, T. Xie, N. Tillmann, and J. de Halleux, "Guided test generation for coverage criteria," in *IEEE International Conference on Software Maintenance (ICSM 2010), September 12-18, 2010, Timisoara, Romania*, 2010, pp. 1–10.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.