PAPER

SCSE: Boosting Symbolic Execution via State Concretization

Huibin WANG^{†a)}, Chunqiang LI[†], Jianyi MENG^{††}, Nonmembers, and Xiaoyan XIANG^{††b)}, Member

SUMMARY Symbolic execution is capable of automatically generating tests that achieve high coverage. However, its practical use is limited by the scalability problem. To mitigate it, this paper proposes State Concretization based Symbolic Execution (SCSE). SCSE speeds up symbolic execution via state concretization. Specifically, by introducing a concrete store, our approach avoids invoking the constraint solver to check path feasibility at conditional instructions. Intuitively, there is no need to check the feasibility of a path along a concrete execution since it is always feasible. With state concretization, the number of solver queries greatly decreases, thus improving the efficiency of symbolic execution. Through experimental evaluation on real programs, we show that state concretization helps to speed up symbolic execution significantly.

key words: state concretization, symbolic execution, constraint solving

1. Introduction

Symbolic execution (SE) has emerged as a popular technique for software testing [1]. The method feeds program with symbolic inputs instead of concrete ones, and extends instruction semantic to handle symbolic variables. By systematically walking through the path space, all behaviors of a program without regard to loops or recursions can be covered. SE is thus powerful for generating high-coverage test suites. Higher coverage often implies higher probability to expose security bugs.

Despite the great promise, existing SE approaches are unlikely to scale beyond small applications. The scalability problem mainly stems from the path explosion phenomenon and constraint solving overhead. Path space often explodes because there is an exponential relationship between the number of conditionals and the number of paths. On the other hand, since constraint solving is essentially an NP-complete problem, its cost almost always dominates everything else. These performance issues are major hurdles limiting the practical use of symbolic execution.

To alleviate the constraint solving overhead, the literature has proposed many approaches. For instance, *Counterexample caching* [2] stores constraint solving results as counterexamples for later reuse. *Concretization* in concolic execution [3] reduces complex constraints into simpler ones.

Speculative symbolic execution [4] invokes the constraint solver only when the number of unsolved path conditions is accumulated to a specified number. Despite these optimizations, constraint solving is still the most dominant in SE. As is shown in prior work [2], constraint solving consumes 40% ~ 90% of the whole runtime. The experiments of *speculative symbolic execution* [4] also shows that constraint solving dominates most of the runtime.

This paper proposes a new form of symbolic execution named State Concretization based Symbolic Execution (SCSE). SCSE speeds up standard symbolic execution by eliminating redundant constraint solver queries, and hence alleviates the scalability problem. The key idea to achieve this goal is *state concretization*. That is, a state in standard symbolic execution is extended by introducing a concrete store, thus creating a new form of program state representation. In standard symbolic execution, when a conditional instruction is encountered, the constraint solver should be invoked to check the feasibility of two branches. But in SCSE, such feasibility checking can be eliminated. With state concretization, we can achieve the same effect by consistency checking, i.e. checking whether there is a concrete store that is consistent with the path condition. Our idea comes from an intuition: if there is a concrete execution along the path, the path must be feasible. In addition, by strategically updating the concrete store, the overhead of consistency checking is kept low. In general, state concretization allows SCSE to eliminate the constraint solver queries for feasibility checking, and hence speeds up symbolic execution significantly.

Even though SCSE adopts the similar idea of *concretization* as in concolic execution, there are major differences between them. Concolic execution reasons a single path at a time. But every run needs to restart from the very beginning, and hence the same instructions are executed repeatedly. Our experimental results show that this re-execution could be very expensive in practice. However, SCSE clones the execution state at each conditional instruction and reasons multiple paths simultaneously. SCSE never re-executes previous instructions, thus avoiding redundant work.

As to *counterexample caching* [2], one of the most popular constraint solving optimizations, SCSE also has a great advantage. *Counterexample caching* uses a sophisticated data structure derived from UBTree to map sets of constraints to concrete variable assignments (or a special sentinel if the constraint set is unsatisfiable). The results of previous similar queries can thus be reused. However, the

Manuscript received August 25, 2018.

Manuscript revised January 18, 2019.

Manuscript publicized April 26, 2019.

[†]The authors are with Institute of VLSI Design, Zhejiang University, Hangzhou, China.

^{††}The authors are with State Key Laboratory of ASIC & System, Fudan University, Shanghai, China.

a) E-mail: wanghuibin@zju.edu.cn

b) E-mail: xiangxy@fudan.edu.cn

DOI: 10.1587/transinf.2018EDP7298

sophistication of its internal data structure brings overhead. Although much faster than querying the solver, searching in the cache still takes a lot of time. Besides, the cache consumes much memory, which may become a heavy burden when a large program is handled. However, in SCSE, the concrete store is so simple that both its computational overhead and memory consumption are very low. These observations have been confirmed in our experimental analysis.

In this paper, we describe the details of SCSE algorithm and discuss its effectiveness. We also evaluate it using benchmarks from the 2018 Competition on Software Verification and the GNU Coreutils suite. The former are verification tasks with complex control-flow structures. The latter are representatives of the system code in Unix/Linux which are challenging for SE. Results show that SCSE can have a great speedup over the state-of-the-art approaches. Overall, this paper makes the following contributions:

- We propose a new symbolic execution method called State Concretization based Symbolic Execution (SCSE), which mitigates the scalability problem by reducing constraint solving overhead.
- We precisely define state concretization and prove an important property related, which is crucial to the efficiency of our method.
- We have implemented SCSE on top of KLEE to extend the scalability of this symbolic executor.
- We have experimentally and empirically compared our new method with the state-of-the-art approaches. We show that our method is effective to enhance symbolic execution.

The remainder of this paper is organized as follows. Section 2 reviews standard symbolic execution. Section 3 elaborates the details of SCSE. Section 4 reports the experimental results and discusses the effectiveness of SCSE. Finally, Sects. 5 and 6 review the related work and conclude.

2. Background of Symbolic Execution

We begin with some background knowledge of symbolic execution (SE) to make sense of what follows. A program P consists of a set V of variables and a set $Insns$ of instructions. Let $V_{in} \subseteq V$ be marked as symbolic, and Val be the value space. A test input is a mapping $t : V_{in} \rightarrow Val$. A test suite $\mathcal{T}$ is a set of test inputs. The aim of SE is to compute a test suite $\mathcal{T}$, which could cover the path space as much as possible.

An instruction $insn \in Insns$ can be a *conditional* **if**(e) **then** **goto** ℓ , an *assignment* $v := e$, **abort**, which represents the faulty termination, or **halt**, which represents the normal termination. With proper code transformations, the above instruction types are sufficient to represent the semantics of arbitrary codes.

Essentially, SE feeds programs with symbolic values as input and outputs the results as functions of symbolic values. Standard SE traverses the path space of a program

```

1  int foo(int x, int y) {
2      if (x < 0){
3          x = -x;
4      }
5      x = x + y;
6      if (x > 2){
7          ret = x;
8      } else {
9          ret = y;
10     }
11     return ret;
12 }

```

Fig. 1 An example program.

Fig. 2 Execution tree of the example program.

by maintaining symbolic program states. A symbolic state contains an instruction location ℓ , a symbolic store Δ and a path condition Π . The symbolic store represents the symbolic values of program variables. The path condition is a boolean formula whose satisfying assignment can drive the concrete execution along the current path. When an assignment instruction is encountered, the symbolic store is updated. When a conditional instruction is encountered, both branches are taken and the state is forked. For each branch, the corresponding condition is added into the path condition and the constraint solver is invoked to check the feasibility. When the end of a path is reached, the constraint solver is invoked again to generate a satisfying test input. Eventually, the symbolic states generated form a *symbolic execution tree*.

Consider the program in Fig. 1, which has two input variables x, y . Figure 2 shows its symbolic execution tree. We choose the commonly used *depth first search* (DFS) to explore the tree. In the initial state s_0 , the symbolic store is initialized as $\Delta[x \rightarrow X, y \rightarrow Y]$, where symbols X and Y represent the input. The path condition Π is initialized as $\langle true \rangle$. Execution forks when the conditional **if** ($x < 0$) is encountered. The constraints, $\langle X < 0 \rangle$ and $\langle X \geq 0 \rangle$, are added to the path conditions of the two branches, respectively. The feasibility is checked by querying the solver. Both branches are feasible here. Two successive states s_1 and s_8 thus re-join the worklist. Next, we explore s_1 , whose corresponding instruction is an assignment $x = -x$. The symbolic store is then updated to $\Delta[x \rightarrow -X, y \rightarrow Y]$, generating state s_2 . The following steps are similar. When a leaf (e.g. s_5) is ex-

explored, the solver is invoked to generate a test input. Clearly, there are ten queries in total, six (the corresponding nodes in Fig. 2 are marked blue) of which are to check feasibility and the other four (marked red) are to generate test inputs.

In the remainder of this paper, we shall present our new approach, State Concretization based Symbolic Execution, which can reduce the number of constraint solver queries significantly.

3. State Concretization Based Symbolic Execution

3.1 A Motivating Example

State Concretization based Symbolic Execution (SCSE) attempts to speed up standard symbolic execution by eliminating redundant constraint solver queries. During the exploration, SCSE maintains concretized symbolic states instead of the pure symbolic states in standard symbolic execution. In addition to an instruction location ℓ , a symbolic store Δ_s and a path condition Π , each concretized state also contains a concrete store Δ_c , which represents the concrete values of program variables. During SCSE, the concrete store Δ_c may be inconsistent with the path condition Π . That is, under the constraint of Π , the concrete execution of the program can not reach a state where all concrete values of program variables is the same as Δ_c . By contrast, if Δ_c is consistent with Π , the path must be feasible since there is already a concrete execution along this path. With the help of the concrete store, the feasibility checking at conditionals can be eliminated. Instead the consistency of a state should be checked. Besides, by strategically updating the concrete store, the overhead of consistency checking is kept low. Overall, consistency checking is much cheaper than feasibility checking.

In this subsection, we illustrate the main idea behind SCSE using an example. Consider the program in Fig. 1 again. Its corresponding symbolic execution tree is shown in Fig. 2. At the initial state s_0 , a concrete store, say $\Delta_c[x : 5, y : 8]$, is attached. When the conditional “if($x < 0$)” is being executed, s_0 is forked into $s_1 = (\ell_2, \langle X < 0 \rangle, \Delta_s[x : X, y : Y], \Delta_c[x : 5, y : 8])$ and $s_8 = (\ell_3, \langle X \geq 0 \rangle, \Delta_s[x : X, y : Y], \Delta_c[x : 5, y : 8])$. Since $x = 5$ steers the concrete execution into the “else” branch but the concrete stores of both branches stay unchanged, we immediately know that s_1 is inconsistent and s_8 is consistent. For s_8 , it is unnecessary to invoke the constraint solver because the “else” branch must be feasible, or the consistent concrete store cannot exist. For the inconsistent state s_1 , which represents the “then” branch, the feasibility is not yet known. Therefore, s_8 can continue execution without additional work, while s_1 should be made consistent before continuing execution. To make s_1 consistent, we attempt to reconstruct a new consistent concrete store to replace the old one. To do so, we invoke the constraint solver to get a satisfying assignment of the path condition $\langle X < 0 \rangle$. We assume getting $\{X = -5, Y = 8\}$. This assignment is used to compute a concrete value for each variable in $\Delta_s[x : X, y : Y]$, thus obtaining a concrete store $\Delta_c[x : -5, y : 8]$. Replacing the old

Algorithm 1: SCSE Algorithm

Input: Initial location ℓ_0 , instruction decoder *instrAt*
Output: Test suite $\mathcal{T}$
Data: Worklist $\mathcal{W}$, instruction location ℓ , path condition Π , symbolic store Δ_s , concrete store Δ_c

```

1  $\mathcal{T} \leftarrow \emptyset$ ;
2  $\mathcal{W} \leftarrow \{(\ell_0, \text{true}, \emptyset, \emptyset)\}$ ;          /* initial worklist */
3 while  $\mathcal{W} \neq \emptyset$  do
4    $((\ell, \Pi, \Delta_s, \Delta_c), \mathcal{W}) \leftarrow \text{select}(\mathcal{W})$ ;
5   if  $\Delta_c$  is not consistent with  $\Pi$  then
6      $\Delta_c \leftarrow \text{reconcretize}(\Pi, \Delta_s)$ ;
7   /* The state must be consistent here */
8   /* Interpret the instruction */
9   switch instrAt( $\ell$ ) do
10    case abort
11      reportBug( $\ell$ ); continue; /* Bug Found!!! */
12    case halt
13      continue; /* end of path */
14    case  $v := e$ 
15       $e_s \leftarrow \text{evalSymbolic}(\Delta_s, e)$ ;
16       $e_c \leftarrow \text{evalConcrete}(\Delta_c, e)$ ;
17       $S \leftarrow \{(\text{succ}(\ell), \Pi, \Delta_s[v \rightarrow e_s], \Delta_c[v \rightarrow e_c])\}$ ;
18    case if( $e$ ) then goto  $\ell'$ 
19       $e_s \leftarrow \text{evalSymbolic}(\Delta_s, e)$ ;
20       $e_c \leftarrow \text{evalConcrete}(\Delta_c, e)$ ;
21       $S \leftarrow \{(\ell', \Pi \wedge e_s, \Delta_s, \Delta_c)\}$ ; /* "then" */
22       $S \leftarrow S \cup \{(\text{succ}(\ell), \Pi \wedge \neg e_s, \Delta_s, \Delta_c)\}$ ; /* "else" */
23   $\mathcal{W} \leftarrow \mathcal{W} \cup S$ ;
24 return  $\mathcal{T}$ ;

```

concrete store in s_1 with this new one can generate a new state $s_1 = (\ell_2, \langle X < 0 \rangle, \Delta_s[x : X, y : Y], \Delta_c[x : -5, y : 8])$. s_1 is then consistent and can continue execution. Besides, in the process of reconstruction, a test input replaying this path is generated simultaneously. Hence there is no need to invoke the constraint solver at the end of the path.

Repeating this process, we can explore the whole symbolic execution tree. Note that the inconsistent state only exactly appears at one branch after a conditional is executed. As a result, the number of solver queries is equal to the number of conditionals. Therefore, a total of *three* queries are generated during SCSE. In contrast, the number is *ten* during standard symbolic execution, as is shown in Sect. 2.

From this example, we can see how state concretization helps to eliminate redundant constraint solver queries. Generally speaking, in standard symbolic execution, solver queries are mainly for feasibility checking and test input generation. But in SCSE, solver queries for feasibility checking are eliminated, while solver queries for test input generation are shifted to the reconstruction of concrete stores. Consequently, the number of solver queries greatly decreases.

3.2 Overall Algorithm

The pseudocode of SCSE is presented in Algorithm 1. At a high level, SCSE functions as an engine to traverse the sym-

bolic execution tree of a program. Given a program P and an initial state, SCSE keeps exploring new program paths and generating new test inputs. That is, it should always be able to generate a test input that replays a feasible path.

In this algorithm, a concretized state is represented as a quadruple $(\ell, \Pi, \Delta_s, \Delta_c)$, where ℓ is an instruction location, Π is the path condition, Δ_s is the symbolic store and Δ_c is the concrete store. A concretized state is said to be consistent if its concrete store Δ_c is consistent with its path condition Π . That is, under the constraint of Π , the concrete execution of the program can reach a state where all concrete values of program variables are the same as Δ_c . A formal definition of state consistency will be given in the next subsection.

In line 2, the algorithm initializes the worklist with a state representing the start of the program. In each iteration of the **while** loop, the work can be roughly divided into three parts: state selection, consistency checking, and state execution.

State Selection. The **select** function selects the next state to process, and removes it from the worklist W . There are a variety of strategies for selecting the next state, such as DFS, random, generational search or other search heuristics.

Consistency checking. Before executing the selected state, we check its consistency in Line 5. If it is inconsistent, the subprocedure **reconcretize** is invoked in Line 6 to reconstruct a new consistent concrete store. This reconstruction process is called *state reconcretization*, of which more details will be given below.

State execution. After ensuring that the current state is consistent, SCSE switches over the instruction types in Line 7.

- At a bug location, when an **abort** is encountered, we report the bug, terminate the state, and go on for next iteration. (see Line 8 to 9)
- At the end of a normal path, when a **halt** is encountered, we terminate the state and go on for next iteration. (see Line 10 to 11)
- For an assignment $v := e$, e is evaluated symbolically to e_s and concretely to e_c . Then the symbolic store Δ_s and the concrete store Δ_c are both updated. (see Line 12 to 15)
- For a conditional **if**(e) **then goto** ℓ' , we evaluate the condition e both symbolically and concretely. This results in a symbolic condition e_s and a concrete condition e_c . The current state is then forked into two concurrent executions: one following the “then” branch, the other, the “else”. The path constraints are updated by doing the assignment $\Pi \leftarrow \Pi \wedge e_s$ and $\Pi \leftarrow \Pi \wedge \neg e_s$, respectively. The symbolic store and concrete store both stay unchanged. (see Line 16 to 20)

3.3 Consistency Checking

Each time a state is selected to be executed, its consistency should be checked first. Consistency checking is crucial to the overall performance since every loop should do this

work. To this end, we formally define state consistency as follows to study this concept further:

Definition 3.1. State consistency. Given a state $S = (\ell, \Pi, \Delta_s, \Delta_c)$, the set of input variables V_{in} , S is consistent if and only if there is a concrete test input t which satisfies both the following properties: 1) for each $v \in V_{in}$, $t[v] = \Delta_c[v]$; 2) t is an satisfying assignment of the path condition Π .

Theorem 3.1. Suppose we have three concretized state, $S_0 = (\ell, \Pi, \Delta_s, \Delta_c)$, $S_1 = (\ell', \Pi \wedge c, \Delta_s, \Delta_c)$, $S_2 = (\ell', \Pi \wedge \neg c, \Delta_s, \Delta_c)$. If S_0 is consistent, only one of S_1 and S_2 is consistent.

Proof. Let t be a test input such that for each $v \in V_{in}$, $t[v] = \Delta_c[v]$. According to Definition 3.1, t is also a satisfiable assignment for Π since S_0 is consistent. Moreover, in c and $\neg c$ there is only one that is *true* over t . If c is *true*, t satisfies $\Pi \wedge c$ of S_1 and does not satisfy $\Pi \wedge \neg c$ of S_2 . As a result, S_1 is consistent and S_2 is inconsistent. If c is *false*, t satisfies $\Pi \wedge \neg c$ of S_2 and does not satisfy $\Pi \wedge c$ of S_1 . Hence S_1 is inconsistent and S_2 is consistent. To sum up, only one of S_1 and S_2 is consistent. This completes the proof. $\square$

According to Theorem 3.1, we directly obtain an efficient method to check the consistency of a state. As stated in the proof, only one of the two states generated is consistent when a conditional instruction is executed. In addition, which one is consistent is determined by the concrete value of the condition c . That is, if $S_{orig}.\Delta_c[c] = \text{true}$, the state for the “then” branch is consistent; otherwise, the state for the “else” branch is consistent. This information is attached to each state as a flag for later inspection. More specifically, we add a flag variable *consistent* to each state to indicate its consistency. The rule of updating *consistent* is as follows: for conditionals, $S_{then}.\text{consistent} = S_{orig}.\Delta_c[c]$ and $S_{else}.\text{consistent} = \neg S_{orig}.\Delta_c[c]$; for other instruction types, the flag stays unchanged. As a result, the task of consistency checking can be efficiently accomplished by simply checking the flag.

3.4 State Reconcretization

When found to be inconsistent, the selected state should be made consistent by *state reconcretization*. Specifically, the aim is to produce a concrete store that is consistent with the given path condition Π . The pseudocode is shown in Algorithm 2.

First, we attempt to get a satisfying assignment of the path condition Π by invoking the constraint solver. If we fail, we know that the current path is infeasible and terminate the corresponding state. If we successfully obtain an assignment t , we add it into the test suite $\mathcal{T}$ in Line 2. Then we use the assignment t and the current symbolic store Δ_s to reconstruct a new concrete store Δ_c . In fact, for any state in SCSE, variables in a symbolic store and those in a concrete store have a one-to-one correspondence relationship with each other. Each variable in the symbolic store is either a symbolic expression (input-dependent) or a concrete

Algorithm 2: reconcretize(Π, Δ_s)

Input: Path condition Π , Symbolic store Δ_s
Output: an assignment t , concrete store Δ_c
Data: An auxiliary subprogram *evalAssignment* to compute the value of a symbolic expression over a particular assignment, Test suite $\mathcal{T}$

```

1 if  $\Pi$  has a solution  $t$  then
2    $\mathcal{T} \leftarrow \mathcal{T} \cup \{t\};$ 
   /* Compute each variable in the symbolic
   store over  $t$  and generate the concrete
   store */
3    $\Delta_c \leftarrow \emptyset;$ 
4   foreach var in  $\Delta_s$  do
5      $exp \leftarrow \Delta_s[var];$ 
6     if  $exp$  is symbolic then
7        $\Delta_c \leftarrow \Delta_c[var \rightarrow evalAssignment(exp, t)];$ 
8     else
9        $\Delta_c \leftarrow \Delta_c[var \rightarrow exp];$ 
10  return  $\Delta_c;$ 

```

value (input-independent). If it is symbolic, we evaluate the symbolic expression over the assignment t , get a concrete result, and add the result into Δ_c . If it is already concrete, we directly add the value into Δ_c . Line 3 ~ 9 describes this reconstruction process.

For ease of comprehension, we illustrate the algorithm with an example. Suppose we have a symbolic store $\Delta_s = [x : X + 2 \cdot Y, y : Y, k : 10]$ and a path condition $\Pi = \langle X + Y > 4 \rangle$, we want to reconstruct a concrete store that is consistent with them. We first invoke the constraint solver to get an assignment of $\Pi = \langle X + Y > 4 \rangle$, say $\{X = 3, Y = 5\}$. In Δ_s , there are three program variables, i.e. x, y and k . Given $X = 3$ and $Y = 5$, we immediately have $x = X + 2 \cdot Y = 13$ and $y = Y = 5$. k is independent of the symbolic input so $k = 10$. As a result, $\Delta_c = [x : 13, y : 5, k : 10]$ is the concrete store reconstructed.

4. Evaluation

4.1 Experimental Implementation

We have implemented SCSE on top of KLEE, an open-source symbolic executor for LLVM bytecode. KLEE is a typical implementation of standard symbolic execution. KLEE mainly consists of an LLVM virtual machine to interpret LLVM instructions, an adaptive search engine to systematically explore program states, as well as a powerful constraint solver STP to determine path feasibility and compute test inputs. We have extended KLEE to support SCSE as follows.

- **State Concretization.** We introduced a concrete store to extend the state representation. Class *Expr*, the base class of all kinds of variables, was extended with a concrete value.
- **Semantic Extension.** The interpreter was extended to update the concrete store when executing an instruction. Most of this task was delegated to the creation of different

kinds of Exprs.

- **State Reconcretization.** A state should be reconcretized if it was inconsistent. The objects to be reconcretized included global variables, heap objects and local variables.

Besides, we simulated concolic execution by further modifying SCSE. Before presenting the modification, we briefly describe how concolic execution works. Concolic execution starts with some given or random input, gathers symbolic constraints on inputs at conditionals along the execution, and then invokes a constraint solver to infer variants of the previous inputs. Once the current path ends, a test-input variant is strategically selected to steer the next execution toward an alternative feasible path. By systematically repeating this process, all feasible paths can be explored.

Our modification for concolic execution mainly consisted of three parts. Firstly, when a conditional is encountered, execution directly transferred to one of the two branches according to the concrete store and no longer forked. Secondly, a container was used to maintain test-input variants. Finally, when a path ended, a test input was strategically selected from the container to form an initial state and execution restarted.

4.2 Experimental Evaluation

In this section, we evaluate SCSE using multiple benchmarks with respect to four research questions:

- RQ1.** Does state concretization reduce the number of constraint solver queries? We show that symbolic execution with state concretization generates much fewer solver queries than without.
- RQ2.** Since redundant state exploration is the bottleneck of concolic execution, does SCSE overcome this problem? We show that SCSE explore much fewer states than concolic execution, thus no longer suffering from redundant state exploration.
- RQ3.** How about the efficiency of SCSE? We show that SCSE decreases the overall runtime significantly.
- RQ4.** How about the overhead of state concretization? We show that the computational overhead of state concretization is very low. We also show that, although slightly higher than standard SE, the memory consumption of SCSE is much lower than SE with counterexample caching.

4.2.1 Evaluation Setup

We used KLEE 1.4.0 compiled with LLVM 3.4.2 and STP 2.1.2 as the constraint solver. Our implementations for SCSE and concolic execution were based on this version too. All experiments were conducted on a server running Ubuntu 16.04, equipped with a dual-core Intel processor at 2.7GHz and 4GB of RAM.

We created two benchmarks: the 2018 Competition on

Table 1 States explored(n_s) and queries made(n_q) in standard symbolic execution, concolic execution and SCSE.

Program Name	States n_s			Queries n_q		
	SE	concolic	SCSE	SE	concolic	SCSE
kbfiltr2t	7691	163996	7691	208	71	71
svr6t	13857	417334	13857	883	389	389
floppy4t	44101	1144005	44101	1943	756	756
diskperf1t	31425	689174	31425	1375	616	616
cdaudio1t	43591	1022507	43591	1361	486	486
clnt2t	7158	194677	7158	678	337	337

Software Verification (sv-comp18) [5] and the GNU Coreutils suite. Each program was first transformed into the LLVM bitcode using the standard Clang/LLVM tool-set. Symbolic executors took the LLVM bitcode program as input. Symbolic variables were either imported by user annotation or from the command-line arguments. For all analyses including concolic execution, Depth First Search (DFS) was chosen as the default exploration strategy (For *concolic execution*, this “*exploration strategy*” determines how to select a test input from the container of test-input variants).

Benchmark 1: sv-comp18. We chose six programs from **ntdrivers-simplified** and **ssh-simplified** instances. These programs are artificial verification tasks with complex control-flow structures. By limiting the maximum loop iterations to 10, we ensure that the path space of each program is finite and hence can be entirely explored. To comply with KLEE’s standards, they were slightly modified, for example, using KLEE’s API `klee_make_symbolic` to import a symbolic value. This benchmark was mainly used to answer the first two research questions.

Benchmark 2: GNU Coreutils. We selected 33 programs without random behaviour from the GNU Coreutils suite. The details of selection will be described below. We used Coreutils to validate the effectiveness of SCSE and to evaluate the overhead since the Coreutils suite was originally used by KLEE and other researchers to evaluate their systems. These programs are real medium-size programs in the Unix/Linux operating system. They are challenging for symbolic execution because they extensively use error checking code as well as complicated data structures such as trees and lists. Unless otherwise specified, the argument configuration of each program was the same as in prior work [2].

4.2.2 Solver Query and State Exploration

To answer the first two research questions, we conducted an experiment on the six programs from sv-comp18. For each program, we ran three kinds of analysis: standard symbolic execution (abbreviated as *SE*), concolic execution (abbreviated as *concolic*), and SCSE. We recorded the number of states explored (n_s) and the number of queries generated (n_q) after exploring the whole path space.

Table 1 presents the results. The relationship of the three analyses with respect to n_q and n_s is particularly interesting: 1) for n_q , SCSE is the same as concolic execution, but much less than standard symbolic execution; 2) for n_s ,

SCSE is the same as standard symbolic execution, but much less than concolic execution.

4.2.3 Randomness Elimination

To answer the remaining two research questions, we conducted experiments on real programs from the Coreutils suite. To ensure that results across runs were comparable, we first made KLEE behave deterministically. KLEE has random behaviour mainly because it relies on a wide variety of timeouts, concrete memory addresses and other environmental values. To eliminate the randomness as much as possible, we disabled Address Space Layout Randomization (ASLR) in Ubuntu, used depth-first search strategy, turned on the “`allocate-determ`” command line argument and relied on TCMalloc [6] to deterministically allocate memory. Originally, there are 100 real programs in the Coreutils suite. From these programs, we selected a subset that could behave deterministically.

- (1) We ran KLEE (standard) with a time limit of 30 minutes. Then the number of instructions executed and the execution trace were recorded.
- (2) We ran KLEE again, but stopped when the number of instructions executed reached the value obtained in step (1).
- (3) We compared the execution traces between steps (1) and (2). If there was any mismatch, the corresponding program was discarded.

After these steps, we discarded 67 programs and had 33 left. Table 2 lists the 33 programs for which execution was deterministic. The programs left were considered for further analysis.

4.2.4 Effectiveness

We evaluate the effectiveness of SCSE by assessing its ability to speed up symbolic execution. For each program, we limited the number of instructions executed to the value obtained from step (1) in Sect. 4.2.3, and ran four kinds of analysis: concolic execution, pure SE, SE with counterexample caching (abbreviated as *SE-cex*), and SCSE. We configured SE-cex by adding the argument “`use-cex-cache=true`” to the command line of KLEE. To ensure that the same part of the execution tree is explored, we modified the instruction counting mechanism in concolic execution to exclude repeated instructions. Table 2 summarizes the results. For each analysis, we report the runtime in seconds. Then we compute the speed-up of SCSE over other three analyses. Programs are sorted by the speed-up of SCSE over pure SE (the column marked grey).

The results in Table 2 show that SCSE indeed improves performance significantly in many cases. Specifically, there are 22 programs for which SCSE speeds up pure SE by more than 2.0x. The largest improvement occurs at `fmt` with speed-up of 15.88x. Besides, SCSE never negatively impacts performance compared with pure SE. Even the small-

Table 2 Runtime of different SE approaches running the same instructions: standard symbolic execution (SE), concolic execution (concolic), symbolic execution with counterexample caching (SE-cex) and SCSE. The first three constitute the Baselines, while SCSE is our proposed approach. The middle three columns present the speed-up of SCSE over the three Baselines. The last two columns show state reconcretization time of SCSE and its percentage in total runtime. The table is sorted by the speed-up of SCSE over standard SE.

Program	Time (seconds)				Speedup of SCSE over			Reconcretization	
	SE	concolic	SE-cex	SCSE	SE	concolic	SE-cex	Time (seconds)	Percentage
fmt	2756.84	1074.98	625.34	173.62	15.88	6.19	3.60	0.53	0.31%
expr	2765.09	253.39	1595.00	175.22	15.78	1.45	9.10	0.05	0.03%
unexpand	1784.52	692.09	861.13	214.53	8.32	3.23	4.01	0.38	0.18%
link	1778.42	1865.44	1851.53	447.31	3.98	4.17	4.14	0.96	0.21%
tr	1658.98	1165.58	609.21	474.09	3.50	2.46	1.29	1.24	0.26%
printenv	1823.22	1657.17	279.55	569.39	3.20	2.91	0.49	0.19	0.03%
wc	1631.85	1240.81	485.61	522.90	3.12	2.37	0.93	0.83	0.16%
dircolors	2767.50	6706.92	1420.04	916.45	3.02	7.32	1.55	1.74	0.19%
echo	1613.96	3425.58	1231.49	571.21	2.83	6.00	2.16	2.10	0.37%
comm	1675.29	5001.29	853.86	656.05	2.55	7.62	1.30	3.65	0.56%
csplit	4626.73	4676.23	860.47	1877.37	2.46	2.49	0.46	1.60	0.09%
rmdir	1785.56	3730.63	1114.56	736.62	2.42	5.06	1.51	1.68	0.23%
runcon	1766.41	3148.43	684.11	758.73	2.33	4.15	0.90	1.79	0.24%
ln	1705.52	1159.75	409.85	735.47	2.32	1.58	0.56	0.45	0.06%
mknod	1748.90	3272.76	1391.26	771.15	2.27	4.24	1.80	1.33	0.17%
mkfifo	1689.03	4545.43	1477.00	751.71	2.25	6.05	1.96	1.62	0.22%
nice	1815.40	4212.86	902.36	811.28	2.24	5.19	1.11	1.20	0.15%
[	1678.32	7882.28	1073.57	772.07	2.17	10.21	1.39	2.04	0.26%
expand	1752.87	6099.28	678.52	809.41	2.17	7.54	0.84	1.81	0.22%
env	1617.56	1436.80	337.61	771.99	2.10	1.86	0.44	0.27	0.03%
basename	1839.20	40990.05	1702.97	879.45	2.09	46.61	1.94	1.95	0.22%
base64	1810.85	18753.50	1645.28	877.37	2.06	21.37	1.88	2.16	0.25%
id	1714.41	4654.53	1461.01	861.81	1.99	5.40	1.70	2.11	0.24%
logname	1702.22	5590.43	1514.18	876.91	1.94	6.38	1.73	1.80	0.21%
seq	1580.38	7672.75	850.37	821.55	1.92	9.34	1.04	1.83	0.22%
users	1726.64	6113.06	1650.29	906.45	1.90	6.74	1.82	1.91	0.21%
whoami	1497.69	3752.93	1296.54	789.74	1.90	4.75	1.64	1.57	0.20%
tsort	1721.05	6440.61	1406.74	909.92	1.89	7.08	1.55	1.75	0.19%
dirname	1671.02	5977.69	1532.18	886.89	1.88	6.74	1.73	1.71	0.19%
readlink	1721.11	3636.41	1360.09	921.88	1.87	3.94	1.48	1.82	0.20%
setuidgid	1756.00	4960.50	720.15	972.10	1.81	5.10	0.74	0.96	0.10%
cksum	2253.66	1516.83	1317.54	1364.81	1.65	1.11	0.97	0.37	0.03%
tty	1777.74	11091.71	1276.67	1141.99	1.56	9.71	1.12	4.20	0.37%
Avg.	1915.57	5587.84	1105.34	779.62	2.46	7.17	1.42	1.50	0.19%

est speed-up is up to 1.56x. On average, the speed-up of SCSE over pure SE reaches 2.46x.

The results also show that SCSE outperforms concolic execution. Although there are cases (e.g. cksum) where no significant speed-up can be seen, there is no case where SCSE causes noticeable performance degradation. A total of 29 programs obtain speed-ups by more than 2.0x. The most amazing improvement is for `basename` with speed-up of 46.61x. On average, the speed-up of SCSE over concolic execution reaches 7.17x.

As to SE-cex, one of the most powerful constraint solving optimizations, SCSE also shows remarkable advantages with an average speed-up by 1.42x. Of the 33 programs, there are only 9 where SCSE runs slower than SE-cex. For the other programs, SCSE always runs faster. The largest improvement can be seen for `expr` with speed-up of 9.10x. In addition, we have found that counterexample caching greatly improves pure SE, which accords to the finding in prior work [2].

4.2.5 Overhead

To evaluate the overhead incurred by SCSE, we measured the time spent on state reconcretization, which does not exist in other SE approaches. The time of reconcretization and its percentage in total runtime is listed in the last two columns in Table 2. Remarkably, the average percentage is 0.19% and even the maximum is only 0.56%.

Memory consumption could be a serious concern when large programs are tested. Hence we also measured the maximal memory used at a point of time during the execution of different approaches. The results in Fig. 3 shows the relationship: `concolic` > `SE-cex` > `SCSE` > `SE`. Specifically, concolic execution and SE-cex consume much more memory than the SCSE and pure SE. In addition, SCSE only consumes slightly more memory than pure SE.

4.3 Discussion

Our experiments so far show that state concretization can

Fig. 3 Maximal memory consumed at a point of time during the execution of different approaches: standard symbolic execution (SE), concolic execution (concolic), symbolic execution with counterexample caching (SE-cex), and SCSE. SCSE is our proposed approach. The other three are the baselines.

effectively reduce the number of constraint solver queries and speed up symbolic execution. In this subsection, we discuss an empirical formula to estimate the performance of symbolic executors. Based on the formula and our collected data, we explain why SCSE works well.

The work of SE can be divided into three parts: state execution, constraint solving, and other auxiliary work. The time of state execution is approximately proportional to the number of generated states, and the time of constraint solving is approximately proportional to the number of solver queries. To empirically estimate the total runtime, we ignore the approximation. Then we have the following formula,

$$T = k_1 \cdot n_s + k_2 \cdot n_q + C \quad (1)$$

where T denotes the total runtime, n_s denotes the number of states generated, n_q denotes the number of solver queries, k_1 and k_2 are two corresponding coefficients, and C is the time of other auxiliary work.

In Formula 1, some complicated details are abstracted away, such as different execution time of different instructions and different solving time of different symbolic formulas. k_1 and k_2 can thus be regarded as constants. Auxiliary work may include initialization, gathering statistics and so on. These are not our focus and C can be regarded as constant too.

Essentially, to symbolically execute a program is to explore its symbolic execution tree. Through analyzing the relationship of n_s and n_q with the tree structure, we can estimate the total runtime of symbolic executors. Let a binary tree T be the execution tree of a given program P , and n_0 be a bound of the maximum number of leaf nodes, n_1 be that of degree-1 nodes, n_2 be that of degree-2 nodes. If P does not contain any loop or recursion, such bounds are guaranteed to exist. If P contains input-dependent loops or recursions, the number of execution paths may be infinite. In practice, however, these bounds can always be enforced by simply limiting the resources used, such as time or memory. $N = n_0 + n_1 + n_2$ is the total number of nodes in T .

Given such bounds, the number of solver queries n_q can be calculated as follows:

- **pure SE.** $n_q = 2 \cdot n_2 + n_0 = 3 \cdot n_0 - 2$. The $2 \cdot n_2$ item means

Table 3 Empirical performance estimate based on Eq. (1): pure SE, concolic execution and SCSE.

Method	Queries(n_q)	States(n_s)	Time spent T (based on Formula 1)
pure SE	$3 \cdot n_0 - 2$	N	$k_1 \cdot N + k_2 \cdot (3 \cdot n_0 - 2) + C$
Concolic execution	$n_0 - 1$	$\frac{(N+1)}{2} \cdot \log(N+1)$	$k_1 \cdot \frac{N+1}{2} \cdot \log(N+1) + k_2 \cdot (n_0 - 1) + C$
SCSE	$n_0 - 1$	N	$k_1 \cdot N + k_2 \cdot (n_0 - 1) + C$

that the solver is invoked twice for the feasibility checking at each branch instruction. The second item n_0 means that the solver is invoked at the end of a path to generate a test input. Note that for a binary tree, $n_0 = n_2 + 1$.

- **Concolic execution.** $n_q = n_0$. Every solver query is used to compute a test input, which replays a unique path.
- **SCSE.** $n_q = n_0$. Similar to concolic execution.

The number of states explored n_s can be calculated as follows:

- **pure SE.** $n_s = N$. All states in T are explored without repetition. (N denotes the total number of nodes in T .)
- **Concolic execution.** $n_s \gg N$. States are repeatedly explored since each run restarts from the very beginning. The exact value of n_s closely relates to the tree structure. We make an estimate here. T is assumed to be a perfect binary tree. The number of nodes from the root to a leaf is thus $\log(N+1)$. There are $\frac{(N+1)}{2}$ leaves in total. Therefore, $n_s = \frac{(N+1)}{2} \cdot \log(N+1)$. Although in practice the tree structure is probably far from perfect, the result has the same order of magnitude in the sense of probability distribution. What's more, things may get worse if there is a lot of degree-1 nodes near the root. It appears that n_s can be much greater than N in practice.
- **SCSE.** $n_s = N$. Similar to pure SE.

Table 3 summarizes the results, which are roughly in line with our collected data. Compared with pure SE, SCSE avoids unnecessary feasibility checking, thus generating much fewer solver queries. Compared with concolic execution, SCSE avoids redundant symbolic re-execution, thus exploring much fewer states. However, we also notice

that the specific values of n_s and n_q do not strictly adhere to the empirical calculation. For example, for `clnt2t`, n_q of pure SE is 678, while n_q of SCSE is 337. There is no 3 times relationship between them, not adhering to the empirical calculation above. This deviation is probably because in empirical calculation we only consider queries for feasibility checking and test input generation, while in practice queries also occur at symbolic pointer concretization, array index validation and so on. In addition, in practice the structure of the symbolic execution tree is far from perfect, thus making n_s deviate.

Despite these abstractions in the empirical calculation, we emphasize that the data (n_s and n_q) of SCSE is the smallest of these approaches. That's why SCSE works well. More directly and clearly, the runtime data in Table 2 shows that SCSE runs faster than pure SE and concolic execution.

As regards the comparison with counterexample caching, SCSE does perform worse for some programs. This is not surprising because the performance of counterexample caching is closely related to the cache-hit rate. Given DFS as the exploration strategy, the cache-hit rate mainly depends on the specific structure of the tested program. When the constraints generated for a particular program are cache-friendly, counterexample caching behaves very well, as expected.

Nevertheless, according to the data in the eighth column of Table 2, we emphasize that on average SCSE runs much faster than SE-cex. In fact, the running speed of SE-cex may be slowed down by the maintenance of the complicated cache structure and the low cache-hit rate when the constraints generated are not cache-friendly. On the other hand, the computational overhead of state concretization mainly stems from the reconstruction of the concrete store. The data in Table 2 provides compelling evidence that the overhead of state concretization is much lower than counterexample caching. More specifically, from the last two columns, we see that the reconstruction time in SCSE only takes up a very small part of the total runtime (less than 1%).

We also measured the memory consumption of these approaches, as Fig. 3 shows. Pure SE can be treated as a baseline. Basically, the memory consumption of SCSE is at the same level as that of pure SE. Our SCSE implementation directly benefits from KLEE by inheriting its copy-on-write mechanism (similar to sharing portions of memory pages across `fork()`), dramatically reducing per-state memory requirements [2]. What makes SCSE consume slightly more memory could be that SCSE needs to maintain a concrete store for each state whereas pure SE does not. Both pure SE and SCSE do not use complicated data structures like SE-cex, whose counterexample cache needs to store massive sets of constraints; so pure SE and SCSE consume much less memory than SE-cex. This result clearly shows that SCSE has an advantage over SE-cex in terms of memory consumption.

With regard to concolic execution, our results show that it consumes the most memory. This may be surprising, but we must admit that what makes concolic execution consume

massive memory is not its concretization mechanism but the representation of test-input variants in our implementation. As stated in previous sections, concolic execution needs to maintain many test-input variants to explore all paths. For each test input, a memory block is allocated to hold all its fields. Note that for two execution paths sharing a long common prefix, most fields of the test inputs are same. If we take advantage of this fact and use a compact test-input representation that can share the same fields, the memory consumption will be greatly reduced, probably reaching a level similar to that of pure SE. On the other hand, both pure SE and SCSE implement copy-on-write to share similar states and thus consume much less memory. Therefore, it is the representation of key data structures that causes the difference in memory consumption between concolic execution and SCSE, and we do not think this difference is essential. Despite this, we emphasize that SCSE still has an advantage over concolic execution in efficiency in terms of the number of states explored.

Future Work: Our experiments show that state concretization has better spatial performance than counterexample caching. Since memory exhaustion may result in state loss, bad spatial performance could be a serious problem when large programs are handled. The excellent spatial performance of SCSE implies its great potential on larger programs. To explore this potential could be one possible direction. In addition, there are other constraint solving optimizations in literature. State concretization should be able to cooperate with them. Hence another future direction could be to study their cooperation and to find out whether they interfere with each other.

5. Related Work

Symbolic execution was proposed in the 1970s [7] and has attracted the attention of academic research in recent years. The research can be divided into two primary directions: 1) to improve its efficiency; 2) to improve its applicability. SCSE focuses on improving efficiency. We thus mainly review the research in the first direction, followed by a brief discussion of the second.

5.1 Efficiency of Symbolic Execution

The scalability problem in symbolic execution mainly stems from two reasons: path explosion problem and constraint solving overhead. To attack the path explosion problem, various methods have been proposed, including: function summarization [8], heuristic search [9], [10], state merging [11], [12], abstraction [13], combining with static analysis [14], [15] and so on.

On the other hand, plenty of work has attempted to alleviate the constraint solving overhead. A common approach is to reduce constraints into simpler forms. For instance, *constraint independence* optimization divides constraint sets into disjoint independent subsets [16], and *expression rewriting* attempts to simplify constraints by using

techniques from optimizing compilers such as constant folding, strength reduction and linear simplification [2]. In research [17], bit operations are optimized by replacing parts of a symbolic variable with concrete values.

Another approach to unburden the constraint solver is to reuse previously computed solutions. EXE [16] creates a server process that can cache the results of constraint solutions and can answer queries from the execution engine. KLEE [2] implements *counterexample caching*, which uses a data structure derived from UBTREE to map sets of constraints to concrete variable assignments, or to a special sentinel if the constraint set is unsatisfiable. *Memoized symbolic execution* [18] uses a trie-based data structure to record the choices taken when exploring different paths, thus allowing reusing similar queries across runs. In research [19], Cristian suggests transforming original programs to generate friendlier constraints. More recently, researchers in [20] propose a set of semantics-preserving transformations which can generate simpler constraints for array operations.

There are also other ways to alleviate the constraint solving overhead. For example, *concretization* in concolic execution [3] can replace a symbolic operand with a concrete value when the formula is hard for the solver, albeit at the cost of sacrificing soundness in the exploration. *Speculative Symbolic Execution* [4] invokes the constraint solver only when the number of unsolved path conditions is accumulated to a specified number.

SCSE adopts a new idea to attack the constraint solving overhead. Unlike previous approaches that treat all constraint solver queries indiscriminately, SCSE attempts to eliminate a specific type of queries. More specifically, our approach introduces a concrete store to eliminate solver queries for feasibility checking, on the premise of the same exploration effect. Besides, SCSE does not use complicated data structures like counterexample caching and memoized symbolic execution. State concretization is so simple that both its computational overhead and memory consumption are quite low. Although sharing a similarity with *concretization* in concolic execution, our approach avoids re-exploring previous states, thus overcoming the bottleneck of concolic execution.

5.2 Applicability of Symbolic Execution

Symbolic execution has been applied to various software engineering areas, for instance, test input generation [21], [22], program verification [23], [24], vulnerability detection [25], regression testing [14], [26] program debugging [27], [28] and document recovery [29]. The research [22] proposes an approach that combines symbolic execution and symbolic reachability analysis to generate high branch coverage test suite. Research [24] uses annotations to guide dynamic symbolic execution toward unverified program executions. In research [25], symbolic execution is used to detect any violation of security requirements. In regression testing, DiSE [14] uses differences between two related program versions to guide the analysis. The research [27]

presents an interprocedural backwards symbolic analysis to quickly find bugs. DocoverY in research [29] leverages symbolic execution to fix broken documents without any prior knowledge of the file format.

6. Conclusion

We have proposed State Concretization based Symbolic Execution (SCSE), a new approach to enhance symbolic execution via state concretization. SCSE introduces a concrete store to reduce constraint solving time, which is almost always the most dominant in the total runtime of symbolic execution. We have implemented SCSE and evaluated it on real programs. The experimental results show that state concretization reduces the number of constraint solver queries, and hence speeds up symbolic execution significantly. We are confident that our findings can improve knowledge about mitigating the scalability problem of symbolic execution.

Acknowledgements

We would like to thank Hangzhou C-SKY Microsystems CO., LTD for providing experimental environment.

References

- [1] R. Baldoni, E. Coppa, D.C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol.51, no.3, pp.50:1–50:39, May 2018.
- [2] C. Cadar, D. Dunbar, D.R. Engler, et al., "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," *OSDI*, pp.209–224, 2008.
- [3] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," *ACM Sigplan Notices*, pp.213–223, ACM, 2005.
- [4] Y. Zhang, Z. Chen, and J. Wang, "Speculative symbolic execution," *2012 IEEE 23rd International Symposium on Software Reliability Engineering (ISSRE)*, pp.101–110, IEEE, 2012.
- [5] D. Beyer, "Competition on software verification." <https://sv-comp.sosy-lab.org/2018/>, 2018.
- [6] G. Sanjay and M. Paul, "Tcmalloc homepage." <https://code.google.com/p/gperftools/>
- [7] J.C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol.19, no.7, pp.385–394, 1976.
- [8] P. Godefroid, "Compositional dynamic test generation," *ACM Sigplan Notices*, pp.47–54, ACM, 2007.
- [9] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pp.443–446, IEEE, 2008.
- [10] P. Dinges and G. Agha, "Targeted test input generation using symbolic-concrete backward execution," *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pp.31–36, ACM, 2014.
- [11] P. Boonstoppel, C. Cadar, and D. Engler, "Rwset: Attacking path explosion in constraint-based test generation," *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp.351–366, Springer, 2008.
- [12] H. Wang, T. Liu, X. Guan, C. Shen, Q. Zheng, and Z. Yang, "Dependence guided symbolic execution," *IEEE Transactions on Software Engineering*, vol.43, no.3, pp.252–271, 2017.
- [13] S. Anand, C.S. Păsăreanu, and W. Visser, "Symbolic execution with abstraction," *International Journal on Software Tools for Technology Transfer (STTT)*, vol.11, no.1, pp.53–67, 2009.

- [14] S. Person, G. Yang, N. Rungta, and S. Khurshid, "Directed incremental symbolic execution," *ACM SIGPLAN Notices*, pp.504–515, ACM, 2011.
- [15] T. Avgerinos, A. Rebert, S.K. Cha, and D. Brumley, "Enhancing symbolic execution with veritesting," *Proceedings of the 36th International Conference on Software Engineering*, pp.1083–1094, ACM, 2014.
- [16] C. Cadar, V. Ganesh, P.M. Pawlowski, D.L. Dill, and D.R. Engler, "Exe: automatically generating inputs of death," *ACM Transactions on Information and System Security (TISSEC)*, vol.12, no.2, pp.1–38, 2008.
- [17] V. Chipounov, V. Kuznetsov, and G. Candea, "S2e: A platform for in-vivo multi-path analysis of software systems," *ACM SIGPLAN Notices*, vol.46, no.3, pp.265–278, 2011.
- [18] G. Yang, C.S. Păsăreanu, and S. Khurshid, "Memoized symbolic execution," *International Symposium on Software Testing and Analysis*, pp.144–154, 2012.
- [19] C. Cadar, "Targeted program transformations for symbolic execution," *Joint Meeting on Foundations of Software Engineering*, pp.906–909, 2015.
- [20] D.M. Perry, A. Mattavelli, X. Zhang, and C. Cadar, "Accelerating array constraints in symbolic execution," *ACM Sigsoft International Symposium on Software Testing and Analysis*, pp.68–78, 2017.
- [21] S. Anand, E.K. Burke, T.Y. Chen, J. Clark, M.B. Cohen, W. Grieskamp, M. Harman, M.J. Harrold, P. McMinn, A. Bertolino, J.J. Li, and H. Zhu, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol.86, no.8, pp.1978–2001, 2013.
- [22] M. Baluda, G. Denaro, and M. Pezze, "Bidirectional symbolic analysis for effective branch testing," *IEEE Transactions on Software Engineering*, vol.42, no.5, pp.403–426, 2016.
- [23] D. Bucur and M. Kwiatkowska, "On software verification for sensor nodes," *Journal of Systems and Software*, vol.84, no.10, pp.1693–1707, 2011.
- [24] M. Christakis, P. Müller, and V. Wüstholtz, "Guiding dynamic symbolic execution toward unverified program executions," *Proceedings of the 38th International Conference on Software Engineering*, pp.144–155, ACM, 2016.
- [25] H. Li, J. Oh, and H. Lee, "Detecting violations of security requirements for vulnerability discovery in source code," *Ieice Transactions on Information & Systems*, vol.E99-D, no.9, pp.2385–2389, 2016.
- [26] D. Felsing, S. Grebing, V. Klebanov, P. Rümmer, and M. Ulbrich, "Automating regression verification," *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pp.349–360, ACM, 2014.
- [27] S. Chandra, S.J. Fink, and M. Sridharan, "Snugglebug: a powerful approach to weakest preconditions," *ACM Sigplan Notices*, vol.44, no.6, pp.363–374, 2009.
- [28] M. Li, Y. Chen, L. Wang, and G. Xu, "Dynamically validating static memory leak warnings," *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pp.112–122, ACM, 2013.
- [29] T. Kuchta, C. Cadar, M. Castro, and M. Costa, "Docover: Toward generic automatic document recovery," *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pp.563–574, ACM, 2014.

Huibin Wang received the B.S. degree from Zhejiang University, Hangzhou, China, in 2012. He is currently a Ph.D. candidate in Institute of VLSI Design, Zhejiang University, Hangzhou, China. His research interests include program analysis, software verification, software testing and optimizing compiler.

Chunqiang Li received the B.S. degree from Zhejiang University, Hangzhou, China, in 1999, and the M.S. degree from Hangzhou Dianzi University, Hangzhou, China, in 2007. He is currently a Ph.D. candidate in Institute of VLSI Design, Zhejiang University, Hangzhou, China. His research interests include program analysis, optimizing compiler, software debugging and embedded systems.

on it. His current research interests include embedded computer architecture design, the technology and design of ECG processor, and near-threshold and sub-threshold circuit design.

Jianyi Meng received the B.S. and Ph.D. degrees in the Institute of VLSI Design from Zhejiang University, Hangzhou, China, in 2004 and 2009, respectively. He was with the Institute of VLSI Design, Zhejiang University, China, from 2009 to 2013. In 2013, he joined the State Key Lab of ASIC & System, Fudan University, Shanghai, China as a professor. He has proposed the C-SKY (China SKY) ISA in embedded system with several computer scientists, and designed several commercial processors based

Xiaoyan Xiang received the B.S. and Ph.D. degrees from Zhejiang University, Hangzhou, China, in 2008 and 2013, respectively, all in Institute of VLSI Design. She has been a lecturer in the State Key Lab of ASIC & System, Fudan University from 2013. Her research interests include low-voltage low-power circuits for resilient processor and the CPU architecture optimization in IoT field.