Boost Symbolic Execution Using Dynamic State Merging and Forking

Chao Zhang
Beijing National Research Center for
Information Science and Technology(BNRist)
School of Software, Tsinghua University, Beijing 100084, China
zhang-chao13@mails.tsinghua.edu.cn

Weiliang Yin* Zhiqiang Lin†
HUAWEI, China
{yinweiliang.ywliang*,linzhiqiang†}@huawei.com

***Abstract*—Symbolic execution has achieved wide application in software testing and analysis. However, path explosion remains the bottleneck limiting scalability of most symbolic execution engines in practice. One of the promising solutions to address this issue is to merge explored states and decrease number of paths. Nevertheless, state merging leads to increase in complexity of path predicates at the same time, especially in the situation where variables with concrete values are turned symbolic and chances of concretely executing some statements are dissipated. As a result, calculating expressions and constraints becomes much more time-consuming and thus, the performance of symbolic execution is weakened in contrast. To resolve the problem, we propose a merge-fork framework enabling states under exploration to switch automatically between merging mode and forking mode. First, active state forking is introduced to enable forking a state into multiple ones as if a certain merging action taken before were eliminated. Second, we perform dynamic merge-fork analysis to cut source code into pieces and continuously evaluate efficiency of different merging strategies for each piece. Our approach dynamically combines paths under exploration to maximize opportunities for concrete execution and ease the burden on underlying solvers. We implement the framework on the foundation of the symbolic execution engine KLEE, and conduct experiments on GNU Coreutils code using our prototype to present the effect of our proposition. Experiments show up to 30% speedup and 80% decrease in queries compared to existing works.**

***Index Terms*—State Merging, Active State Forking, Local Merge-fork Analysis**

I. Introduction

The innovation of symbolic execution has been proposed over four decades ago [1], [2]. It shows great potential in software analysis and testing, and underlies a growing number of tools which are widely applied in practical cases [3]–[7]. For example, in [5], [6], they apply symbolic execution to greatly improve the performance of traditional fuzz testing [8], [9]. In [7], they incorporate symbolic execution as a bridge to reduce the false positive of static analysis [10] and the false negative of dynamic analysis and run-time verification [11]The key idea behind symbolic execution is to replace input values with symbols, update symbolic expressions for program variables, and calculate path predicates along explored execution paths to determine their feasibility [12], [13]. In this way, symbolic execution simulates how a program runs and explore feasible paths in an efficient way.

However on the other hand, path explosion is still the most significant bottleneck which limits scalability of symbolic execution. When traversing a program, each path encodes decisions on all its historical branches and maintains them in its path predicate [14]. Thus, the number of paths to be explored grows exponentially with the number of branches and exhaustively covering all paths could be extremely time-consuming even for a small program [15].

State-of-the-art solutions to resolve path explosion mainly aim at directly decreasing number of paths or optimizing the process within limited time budgets. A number of articles introduce various search heuristics to improve performance towards particular targets. For example, ideas from random testing and evolutionary algorithms are frequently used to guide the search order of path exploration so that coverage gets higher in the same time [16], [17]. The second way is to apply some imprecise techniques from static analysis, such as abstract interpretation and summary. These techniques calculate approximations to simplify analysis of code and decrease produced states [18], [19]. State merging is another way designed to cut down number of paths to be explored. When path exploration encounters a branch statement, there are two succeeding states constructed and in turn the original path is split into to two sub-paths, corresponding to true and false case respectively. If the two states join at the same location later, they can be merged into one again, with path predicate updated using disjunction and variable expressions using *ITE* operation [20]. Fig.1 shows an example where there are two paths in the given program: $1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 6$ corresponding to true case of if-statement and $1 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 6$ to false case. Succeeding states of 3 and 4 are merged at 5, with path predicate set to $n < 0 \vee \neg(n < 0)$ or simply $TRUE$, and expression of ret set to $ITE(n < 0, -n, n)$.

State merging is intended to decrease the number of paths to be explored while no imprecision is introduced. Meanwhile, it faces the challenge that path predicates and variable expressions may get more complex. Constraints involving disjunction and ITE (actually another form of disjunction) are extremely unfriendly to underlying solvers [21]. Such negative impact on performance is rather serious when expressions which can be evaluated concretely turn symbolic after merging. Modern symbolic execution engines commonly adopt concrete execu-

```
int abs(int n) {
  int ret = 0;
  if (n < 0)
    ret = -n;
  else
    ret = n;
  return ret;
}
```

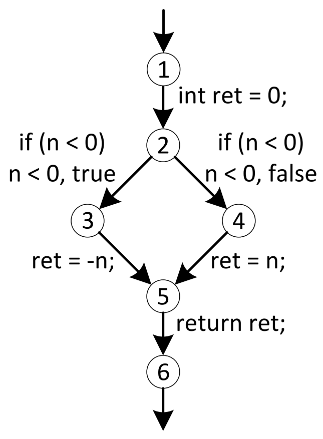

Fig. 1: An example of state merging.

tion when code to be interpreted only involves concrete values. Improper merging actions may dismiss chances of concrete execution and lead to extra solving cost, which eliminates the benefit of decrease in path scale in reverse [20]. Actually, a merging action will speed up symbolic execution or not, or even speed down it, is determined by the future code which is still unexplored. Kuznetsov et al. have proposed an estimation algorithm in [14] to statically find variables frequently used in branch conditions and prevent any merging action erasing concrete values of these variables. In this way, state merging is selectively performed to exclude inefficient cases. We are inspired by this work and try to improve merging strategies to handle more complicated situations. We build a framework dynamically merging and forking states according to the code under execution. During execution, combinations of paths vary to keep chances of concrete execution while exploration of states and paths are joint as much as possible.

Contributions. In this paper, we present a framework boosting symbolic execution by automatically switching between merging states together and forking them separately throughout path exploration. We implement a prototype on the foundation of KLEE [22] and conduct experiments on GNU Coreutils code to evaluate the method. Our main contributions are:

- We introduce active state forking to eliminate a historical merging action and restore the state into original separate ones. By integrating symbolic execution with state merging and forking, we can change the way how states are combined and paths are explored at any moment during execution.
- We present dynamic merge-fork analysis to evaluate performance trade-offs for different path combinations. Decisions are made by estimating chances of concrete execution and efforts in reasoning future statements. The source code is cut into pieces so analysis is simplified and conclusions can be drawn in each piece locally.

The remainder of this paper is organized as follows. In Section II we have an overview of symbolic execution and state merging. Section III introduces active state forking, followed by description of dynamic merge-fork analysis in Section IV. Next we display experimental evaluation of a prototype implementing our merge-fork framework in Section V. Finally we give our conclusion in Section VI.

II. Symbolic Execution and State Merging

In this section, we first present a general process of symbolic execution and then have a discussion on state merging.

A. Classical Symbolic Execution

Symbolic execution starts from the program entry and keeps updating variable expressions and path predicates to systematically explore feasible paths. In this way the executor can reveal code reachability, find bugs or generate test inputs. Descriptions of a general work-flow of classical symbolic execution techniques are given in many works [14], [15]. We refer to these works and present a general algorithm in Algorithm 1. To be concise but without loss of generality, we consider a language made up of three kinds of statements, i.e. assignments, conditional branches and exit statements with exit code to indicate normal termination or failed assertion/error. By unwinding loops and inlining function calls we can transform a program to this form. In the paper, all programs used to describe our method are in such form if not explained particularly.

In symbolic execution, a state of the program is usually encoded as a triple (l, P, M) where l refers to the current program location, P to path predicate corresponding to the path from the entry to l, and M to symbolic store mapping from variables to their current expressions. The algorithm keeps taking out a state from $Worklist$, extending the state and pushing newly found ones back to $Worklist$. Call to $selectState$ at line 3 determines which state is chosen to be visited next, and thus implies the searching strategies of path exploration. The chosen state is then updated according to the next statement just after its location (line 5-21). Call to $eval$ and $checkCond$ in this procedure symbolically calculates expressions and checks satisfiability.

Code at line 22-30 of Algorithm 1 is designed for merging states. For a newly found state s, if there is another state s' in $Worklist$ sharing the same program location, s could be merged with s' rather than directly added to $Worklist$. Definition of $shouldMerge$ makes decisions on whether to merge and thus realizes strategies of state merging.

B. State Merging: SSE vs. DSE

By defining $selectState$ and $shouldMerge$ differently, the classical symbolic execution algorithm presented in Section II-A acts as two extremes in regard to design space of state merging. Dynamic symbolic execution (DSE, adopted in [3], [22]–[24] etc.) skips merging and every execution state encodes a unique path from the entry to its corresponding location. Static symbolic execution (SSE) ensures all paths joining at each junction node in the control flow structure

Algorithm 1 Classical symbolic execution

```
1: $Worklist = \{(l_0, true, M_0)\}$
2: while $Worklist \neq \emptyset$ do
3:   $(l, P, M) = selectState(Worklist)$
4:   $Worklist = Worklist \setminus \{(l, P, M)\}$
5:   switch $nextInst(l)$ do
6:     case $v := e$:
7:       $M' = M \setminus \{(v, M[v])\} \cup \{(v, eval(e, M))\}$
8:       $S = \{(succ(l), P, M')\}$
9:     case $if\ cond\ goto\ l'$:
10:      $sat = checkCond(cond, P, M)$
11:      if $sat == CONST\_TRUE$ then
12:        $S = \{(l', P, M)\}$
13:      else if $sat == CONST\_FALSE$ then
14:        $S = \{(succ(l), P, M)\}$
15:      else if $sat == NON\_CONST$ then
16:        $S = \{(l', P \wedge eval(cond, M), M), (succ(l), P \wedge \neg eval(cond, M), M)\}$
17:      end if
18:    case $exit$:
19:      $S = \emptyset$
20:      $printPath()$
21:  end switch
22:  for all $s = (l, P, M) \in S$ do
23:    for all $s' = (l', P', M') \in Worklist$ do
24:      if $l = l' \wedge shouldMerge(s, s')$ then
25:        $Worklist = Worklist \setminus \{s'\}$
26:        $M'' = \{(v, ITE(P, M[v], M'[v])) | (v, M[v]) \in M\}$
27:        $S = S \setminus \{s\} \cup \{(l, P \vee P', M'')\}$
28:      end if
29:    end for
30:  end for
31:  $Worklist = Worklist \cup S$
32: end while
```

are merged completely, which is common in symbolic model checking and verification condition generation [25], [26].

SSE systematically explores all feasible control paths with all possible input values taken into consideration [20]. It visits states in topological order to ensure that all corresponding states are merged at every junction node in control flow of the program. However, its major trade-offs are great difficulties in calculating path predicates and strict restrictions on searching strategies. DSE techniques are then proposed to interleave concrete and symbolic execution and focus on exploring possible execution paths one by one. A statement can be executed concretely rather than symbolically if all involving variables are concrete. For an assignment, the result is calculated directly if possible. For a conditional branch $if\ cond\ goto\ l'$, concrete execution is performed if $cond$ can be determined concretely. As a result, interpreting the statement does not need to split the current path. DSE techniques such as [3], [23], [24] introduce concrete inputs to make analysis benefit from concrete execution.

Taking both SSE and DSE into consideration, it requires a compromise of the two extremes to use state merging to deal with path explosion in real code. While dynamic techniques remain to keep symbolic execution benefiting from concrete execution and heuristic search order, states are selectively merged to eliminate duplicated efforts exploring paths joining at a certain location. The work of [14] tries to introduce state merging in DSE and switch between different searching strategies. The executor keeps discovering merging opportunities with positive effects on performance and prioritizes visit of involving states. Veritesting [27] proposes combination of SSE and DSE by changing execution mode in process. SSE makes code analyzed one-pass instead of visiting all feasible paths separately in DSE. In this way the executor is able to benefit from solvers and boost path exploration.

Our proposition tries to mix SSE and DSE on the foundation of [14]. In this article, Kuznetsov et al. have constructed a framework to estimate whether it is worth driving path exploration towards a potential merging action. They present a static bottom-up analysis to find variables which may be frequently used later and suggest merging actions with no impact on values of these variables. Path exploration is then under control of an SSE-like module temporally instead of a normal DSE process. We take a more complicated case into consideration where evaluation on a merging action varies across the code. A static decision - either merging or not - improves efficiency for some pieces of code but burdens performance for others. Actually, an optimal solution usually consists of multiple parts which are drawn in different pieces of code locally and combined together dynamically.

III. Active State Forking

In this section, we depict our active state forking technique. First, we use a series of examples to explain motivation for forking and then describe definitions and method of it.

A. Motivating Examples

As is discussed in last chapter, influence of a certain merging action varies throughout different pieces of code. We use some example code in Fig.2 to explain the issue in detail and present effect of active state forking. To be concise, some function calls are not inlined.

In function $f1$ state merging dismisses the opportunity of concrete execution. There are two if-blocks in $f1$, namely line 3 and line 4-5. State forks and path splits when reaching the first if-block (this is passive state forking caused by non-concrete if-condition), and then two sub-paths join after it. Expression of $flag$ gets different along two sub-paths, and is instantly used in if-condition at line 4. Expression of $flag$ is 1 along true case of the first if-block, 0 along false case, and $ITE(a == b, 1, 0)$ if states are merged before line 4. It can be seen that both the two separate states lead concrete execution on the second if-block since its condition can be decided concretely, but the merged one erases concrete value of $flag$ and results in passive state forking. Therefore, the

```
1  void f1(int a, int b) {
2    int flag = 0;
3    if (a == b) flag = 1;
4    if (flag) g1();
5    else g2();
6    exit(0);
7  }
8
9  void f2(int a, int b) {
10   int flag = 0;
11   if (a == b) flag = 1;
12   if (flag) g1();
13   else g2();
14   g3(a, b);
15 }
16
17 void f3(int a, int b) {
18   int flag = 0;
19   if (a == b) flag = 1;
20   g3(a, b);
21   if (flag) g1();
22   else g2();
23 }
```

Fig. 2: Motivating examples.

merging action after the first if-block just makes expressions and path predicate more complex while number of paths to explore is not changed.

In function $f2$, $exit(0)$ at line 6 in $f1$ is replaced with another function call $g3(a, b)$. While things go the same as for $f1$, it is required to reason about call to $g3$ on each path. If paths are merged before line 14, call to $g3$ will be analyzed only once rather than once along each path. Hence, it is expected that states forked in the first if-block should be merged at the exit of the second if-block.

For function $f3$, call to $g3$ goes before the second if-block. As is discussed for $f1$ and $f2$, we can make a similar conclusion that states forked in the first if-block should be merged immediately to avoid repeated efforts reasoning about call to $g3$, but later "restored" to two separate paths with concrete value for $flag$ along each. Common methods try to compare advantages and disadvantages to reveal a static decision on state merging, and our approach, in another way, makes it possible to fork a merged state as if a historical merging action were not taken. In $f3$, paths are made to join after line 19 and fork actively before line 21 with $flag$ restored to concrete along each forked path. In this way, the benefit of state merging is maximized while its negative impact on concrete execution is avoid as far as possible.

B. Merge and Fork States with Extended Execution Graph

The motivating examples figure out that an optimal merging strategy may promote contrast decisions for different pieces of code. Active state forking together with state merging makes it possible to change how paths are combined during exploration. To maintain information of historical merging actions and support forking states, we make extensions to the formalism in classical symbolic execution.

A DSE method as shown in Algorithm 1 conducts path exploration by visiting execution states and creating new ones. This process can be efficiently represented as a graph in which nodes are execution states and edges indicate transitions between them. Since DSE does not merge states, such a graph would be a tree, which is so called symbolic execution tree. Definitions of execution state and symbolic execution tree are given in Definition 1 and 2.

***Definition** 1:* An execution state is defined as a tuple $es = (l, P, M)$ where

- $l \in L$ is the current program location.

- $P \in \mathcal{F}(\Sigma, Const)$ is the path predicate, which is a formula over symbolic inputs Σ and constants $Const$ encoding all decisions of branches along the path from entry to l.

- $M : V \to \mathcal{F}(\Sigma, Const)$ is the symbolic store, which maps a program variable to the formula corresponding to its expression.

***Definition** 2:* A symbolic execution tree is a tree $T = (ES, T)$ where

- ES is the set of execution states.

- $T \subseteq ES \times ES$ is the set of transitions between execution states.

Symbolic execution tree records every explored state and path, and is widely used to characterize the process of symbolic execution [25]. Our extended formalism is built on top of it. While state merging is performed, an execution state could encode multiple nodes in the corresponding symbolic execution tree. Such relations naturally extract information of merging actions, and thus, we introduce virtual states to hold the similar structure. Definition 3 presents how virtual states are defined.

***Definition** 3:* A virtual state is defined as a tuple $vs = (es, CV)$ where

- $es \in ES$ is the corresponding execution state.

- $CV \subseteq V \times Const$ is the set of variables with concrete values together with their values.

When two execution states are merged to be one, we create virtual state for each sub-state respectively and map them to the produced state. Every virtual state tracks its exclusive concrete variables. Our formalism then constructs transitions between execution states and virtual states, resulting in a directed acyclic graph. An execution state contains data of path predicate and symbolic store, and is used to actually execute statement. It appears as a node in the graph only if it never experiences merging or we are not concerned with its historical merging actions anymore. Otherwise, its corresponding virtual states are used instead to make up the graph. Definition 4 describes our extended execution graph formally.

***Definition** 4:* An extended execution graph is a directed acyclic graph $G = (S, T, \psi, \Delta)$ where

- $S \subseteq ES \cup VS$ is the set of states.

- $T \subseteq S \times S$ denotes transitions between states.

- $\psi : ES \rightarrow 2^{VS}$ maps a execution state to the set of its corresponding virtual states.
- $\Delta : VS \rightarrow \mathcal{F}(\Sigma, Const)$ maps a virtual state to its newly added conditions compared to its predecessor's path predicate.

While execution states hold essential data and drive execution to explore paths, virtual states simulate the structure of symbolic execution tree and preserve merging history. If there is no further operation on virtual state, our extended execution graph would degenerate into a tree. However, virtual states are merged and swapped for execution states when updates on variables efface complexity introduced by merging. In the remaining part of this section, we explain in detail how to perform state merging and forking with our extended execution graph.

Merging. When trying to merge $es_1 = (l, P_1, M_1)$ and $es_2 = (l, P_2, M_2)$ into $es_3 = (l, P_1 \vee P_2, M)$ where $M(v) = ITE(P_1, M_1(v), M_2(v))$ for each $v \in V$, we first create virtual states and then update the graph $G = (S, T, \psi, \Delta)$.

If es_1 appears as a node in G (in other word $es_1 \in S$ and $\psi(es_1) = \emptyset$), we create a virtual state directly to represent its corresponding sub-state encoded by es_3: $vs = (es_3, CV)$. CV denotes exclusive concrete variables of this sub-state, so it is made up of variables concrete in es_1 but not in es_2. That is, $CV = \chi(M_1) \setminus \chi(M_2)$ where $\chi(M) = \{(v, M(v)|M(v) \in \mathcal{F}(Const)\}$.

If es_1 encodes multiple sub-states so that $es_1 \notin S$ or $\psi(es_1) \neq \emptyset$, we create a virtual state for every sub-state in $\psi(es_1)$ and add up CV of sub-state and exclusive concrete variables of es_1 to assign to newly created one.

Creating virtual states for es_2 is in the same way, except that exclusive concrete variables of es_2 is calculated as $\chi(M_2) \setminus \chi(M_1)$.

All these generated virtual states are then added to G, together with corresponding transitions. $\psi(es_3)$ is set to be all these new states. Δ of them can be simply set to empty since there is no change between their and their predecessors' path predicates.

Update of Assignment. Updating an execution state es in G can be done immediately. However, if $es \notin S$, set of concrete variables of each virtual state $vs = (es, CV) \in \psi(es)$ requires update. Those not concrete or not exclusively concrete anymore should be removed from concrete variables, while those becoming exclusively concrete should be added to concrete variables.

E.g. for assignment $v := e$, $(v, c_v) \in CV$ is removed if e is not concrete, or it is but calculation does not involve any concrete variables in CV. Otherwise v and its value is added to CV if there is no $(v, c_v) \in CV$, and e is concrete and depends on certain variables and values in CV.

When updating virtual states, ones might point to the same execution state and possess equal concrete variables and values. These states can be merged in the graph since distinguishing such sub-states makes no sense to merging and forking. If all virtual states of an execution state are merged, we can swap the production for the execution state itself, which means that all its historical merging actions do not influence any concrete variable and hence, will never invoke active forking.

Forking. For an execution state es in G, passive forking by conditional branch is the same as in traditional DSE. Otherwise, forking action is applied to every sub-state $vs \in \psi(es)$ and Δ is updated with branch decision accordingly for forked productions.

In the case of active forking aimed at eliminating historical merging actions, the work flow gets more complicated: it is required to partitioning sub-states, constructing path predicates and updating expressions.

The execution state es to be forked encodes multiple sub-states which are represented by virtual states in $\psi(es)$. Active forking is invoked because some variables are turned from concrete to symbolic by merging actions. Thus, the first step is to identify concerned variables. Every virtual state $vs \in \psi(es)$ where these variables are concrete is separated to be a forked state. For sake of efficiency, states with same concrete values for these variables can be joined. In this way, all sub-states of es are partitioned into several groups and each group corresponds to a potential forked state.

For each sub-state vs, its path predicate can be calculated by backtracking to its ancestor execution state and combining with newly added conditions denoted by Δ along the path. The results are used to construct path predicate for each potential forked state and decide ITE selections of variables expressions. The original state es is then forked into several execution states as if some historical merging actions were eliminated. Besides, it is remarkable in real case that optimization of expressions and constraints based on data structure, constant propagation, query cache and other practical techniques is adopted to cut down efforts and boost speed of calculation in state forking.

IV. Dynamic Merge-fork Analysis

Active state forking makes it possible to raise dynamic merging strategies, and next task is to identify ”good” merging actions. In this section, we will explain how to identify possibilities of concrete execution and how to make decisions on merging by analyzing code piece by piece.

A. *Identify Concrete Values and Concrete Execution*

State merging avoids repeated efforts in analysis but may burden underlying solvers as trade-off. Considering examples in Fig.2, the ideal solution would be that we can selectively merge states so that 1) variables turned symbolic are used to calculate future if-conditions as little as possible; 2) merging actions with no influence on future if-conditions are as many as possible. The former prevents state merging from being obstacle to concrete execution while the latter indicates greater benefit from merging.

A bottom-up estimation identifying variables frequently used is presented in [14]. The basic idea is tracking variables used in if-conditions along use-def chain. Our method also pays attention to use in if-condition, but works in top-down order instead because analysis will be break down in different pieces of code. A list of variables with concrete values is maintained and updated when executing. Thus, we can identify possible concrete values and in turn, find out chances of concrete execution.

B. Local Merge-fork Analysis

Active state forking makes it possible to redraw decisions on merging actions, so we can cut a program into pieces and analyze code piece by piece. The division is randomly made under these rules: difference in scale between pieces is limited within a proper bound; cut points are located at exit of if-blocks, end of inlined functions, or terminators of sequential blocks. And a piece is divided further if it contains greatly different parts, e.g. variables used in one part are totally disjoint with ones in another. Nevertheless, too detailed division achieves little increase in performance while goes time-consuming, so we limit the times of further division.

After code is divided into pieces, we can make decision for states whether they should be merged or not in a piece. There are still situations where different parts of code have opposite votes for a merging action in the same piece. To resolve the issue, we propose another two types of decision: merge until a location and merge after a location. The former performs merging temporally and invokes analysis again later, while the latter make a delay to merging. Because of these choices of merging decision, analysis of how to combine paths is invoked at junction node in control flow structure, at the beginning of each piece, and at locations specified by ”merge until” and ”merge after” decisions.

Hence, our merge-fork analysis determines locally how to handle each merging opportunities within a piece of code. For the simplified language used in our discussion, this process can be implemented by counting occurrences of if-condition and in turn estimating number of queries needed. Besides, unresolved function call and code block difficult to reason can also be counted to influence strategy of merging and forking. When decided to merge two states, path exploration is temporally taken over from searching heuristics and involving states are prioritized to visit.

V. Experimental Evaluation

In this section we illustrate some experimental evaluation on our proposition. We build a prototype to implement our merge-fork techniques (Section V-A), and run it with Coreutils programs to analyze performance in two scenarios: exhaustively exploring all feasible paths (Section V-B) and incomplete exploration within certain time budget (Section V-C). Besides, we also make a comparison with previous work of Kuznetsov et al. [14] (Section V-D).

A. Prototype

To evaluate the effect of our merge-fork framework, we implement active state forking and dynamic merge-fork analysis in a prototype on the foundation of the famous symbolic execution engine KLEE. It takes LLVM bitcode compiled from source code as input. The original executor of KLEE performs path exploration with no approximation, and path feasibility is checked at every conditional branch. State forking is invoked for each non-determined branch, and states at the same control location are never merged. KLEE guides exploration with a number of optional search strategies including random search, traditional breadth/depth first search, strategy prioritizing visiting previously unexplored code, and combination of various strategies.

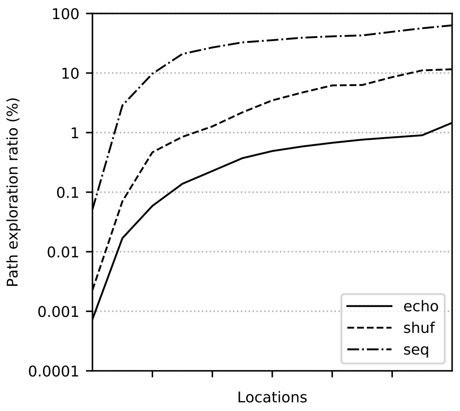

Fig. 3: Ratio of explored paths for 3 representative programs.

KLEE has built in models of execution state and symbolic execution tree. We extend them to implement our active state forking method. Thanks to implementation of expressions and optimization of constraints in KLEE, it is easy to track updates of path predicates and we could perform state forking by directly imposing conditions of dismissed merging action on path predicates and symbolic store. To support dynamic merge-fork analysis, we make a call in state selection ($selectState$) to an LLVM Pass constructing concrete dependence graph and estimating merging effects locally. If merging is suggested, search strategy is temporally fixed to continuously extending paths to target merging location. Otherwise, states are explored according to original search strategies of KLEE.

The original work of KLEE presents an experiment on GNU Coreutils programs which are widely used UNIX command-line utilities. There are 101 files in total adding up to over 70,000 lines of code. We test these programs using our prototype and collect information from output files.

B. Speedup Path Exploration

Our merge-fork framework is aimed at identifying valuable merging opportunities and conducting state merging, so we are first concerned with the influence of our method on number of explored paths.

A direct comparison is measuring number of explored paths with different time spent. However, the number is hard to figure out since there are paths explored partially if exhaustive exploration is not completed. We therefore choose to calculate the number of paths at a specific location instead of with some time spent during the execution. The size of symbolic inputs

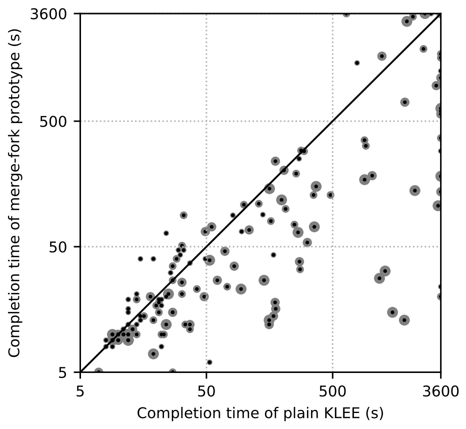

Fig. 4: Compare completion time with plain KLEE for exhaustive exploration tasks.

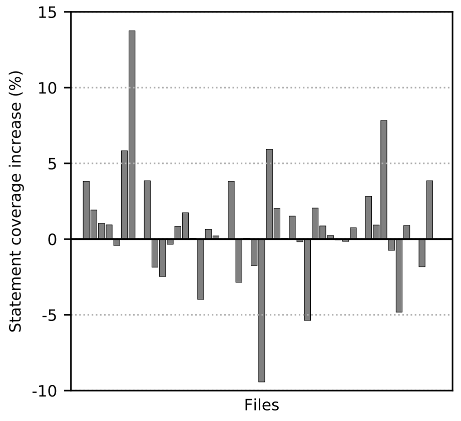

Fig. 5: Compare statement coverage with plain KLEE for uncompleted exploration tasks.

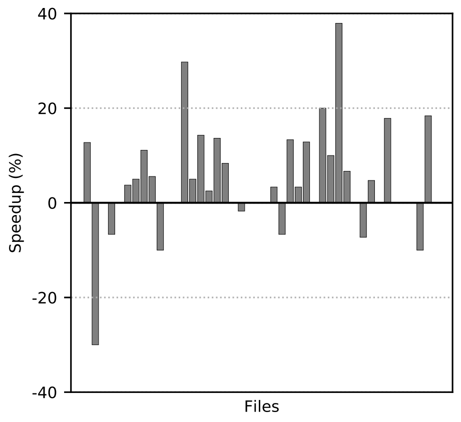

Fig. 6: Speedup over efficient state merging of [14].

is properly set to make it possible for exhaustive exploration, so the accurate number of feasible paths of the program is known. Then we run our prototype and count paths at given control locations. In this way we can reveal how state merging

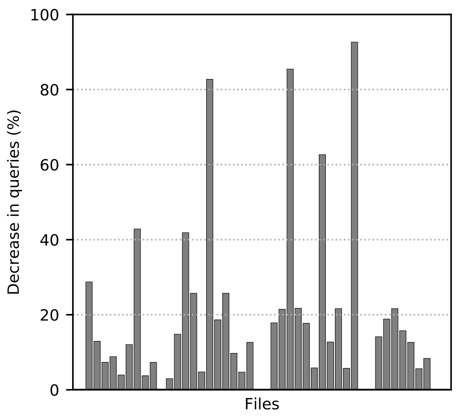

Fig. 7: Decrease in queries comparing with [14].

efficiently joins branches and decrease paths. It is remarkable that paths getting through a certain location might enclose a large gap in time.

Fig.3 shows the growth of path exploration ratio corresponding to program locations for 3 representative files. The final ratio is obtained at the exit of code. It can be seen that the effect in decreasing paths varies among the programs. While a large part of paths are merged in *echo*, state merging makes little influence on *seq*. The trend of growth indicates structure of control flow and involvement of active state forking. Since merging and forking work simultaneously, change in path numbers appear smooth.

The target of decreasing paths is to free underlying solvers' burden and achieve faster execution by omitting duplicated analysis efforts. Hence, we run Coreutils under symbolic inputs with growing size and compare completion time of plain KLEE and our prototype. In order to compare with [14] easily, we draw a similar graph Fig.4 showing the result within 1 hour time limit. The black dots correspond to experiment instances and the gray disks indicate the relative size of the symbolic inputs used in each instance. The result presents that in the majority of cases, our prototype achieves positive speedup over KLEE. The rightmost dots represent those exceed time limit in KLEE but can be solved completely in our framework.

C. Exploration Guided by Statement Coverage

We now look into the situation where tasks can not be finished completely. We choose a proper size of symbolic inputs to keep the engine busy and measure statement coverage in limited time. Plain KLEE uses coverage-oriented search strategy to guide path exploration towards uncovered locations. Our merge-fork framework would interrupt the process and temporally drive exploration towards merging states. Fig.5 shows its influence on statement coverage. While state merging saves analysis efforts, there are more paths can be visited in some cases, which leads to increase in coverage. However, other cases suffer from change in search order and in turn coverage is decreased. Data in Fig.5 confirms that estimation of merging opportunities and fast-forwarding some states do not cause any obvious troubles to the original exploration goal of searching heuristics.

D. Compare to Existed Work

While our proposition is on the foundation of the previous work of Kuznetsov et al. [14], we make some comparison with their experiment results. All the data is obtained from their official website *http://cloud9.epfl.ch*. Since experimental environment and KLEE version may vary, all the data used in our comparison is increase/decrease over our respective KLEE baseline.

First we compare the completion time under the same size of symbolic inputs. Fig.6 illustrates the result. We use completion time of plain KLEE to normalize data of both tools and then calculate the effect of speedup. Over 80% of the tasks get faster with our merge-fork method and average speedup is around 10%. Most instances with negative speedup are finished in a rather short time (10^1 magnitude in seconds).

We also collect queries generated during the execution. Fig.7 shows the comparison. The number of queries is also normalized against data from plain KLEE. It confirms the effect of active state forking since some queries are omitted on separate paths by concrete execution. The improvement in performance is achieved through decreasing paths via state merging and invoking concrete execution by restoring merged paths when needed.

VI. Conclusions

In symbolic execution, state merging decreases the number of paths but eliminates the benefit of concrete execution in contrast. To resolve the conflict and improve the performance, we proposed a merge-fork framework to enable a bi-directional switch for states to be merged and forked. State merging and active state forking can make state and path combinations fit different code dynamically. This in turn makes it possible to divide code into pieces and dynamically draw decision on state merging piece by piece. Experiment on a prototype trivially displays the potential of our approach. While explosive development in dynamic symbolic execution and search-based techniques has a huge impact on classical state merging methodology, it shows promise combining our approach with these techniques and making state merging an efficient method to solve practical issues in symbolic execution.

References

[1] L. A. Clarke, "A system to generate test data and symbolically execute programs," *IEEE Transactions on software engineering*, no. 3, pp. 215–222, 1976.

[2] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[3] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *ACM Sigplan Notices*, vol. 40, no. 6. ACM, 2005, pp. 213–223.

[4] 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.

[5] M. Wang, J. Liang, Y. Chen, Y. Jiang, X. Jiao, H. Liu, X. Zhao, and J. Sun, "Safl: increasing and accelerating testing coverage with symbolic execution and guided fuzzing," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, 2018, pp. 61–64.

[6] J. Liang, M. Wang, Y. Chen, Y. Jiang, and R. Zhang, "Fuzz testing in practice: Obstacles and solutions," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 562–566.

[7] C. Wang, Y. Jiang, X. Zhao, X. Song, M. Gu, and J. Sun, "Weak-assert: a weakness-oriented assertion recommendation toolkit for program analysis," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, 2018, pp. 69–72.

[8] J. GLiang, M. Wang, Y. Chen, and Y. Jiang, "Pafl: Extend fuzzing optimizations of single mode to industrial parallel mode," 2018.

[9] J. Guo, Y. Jiang, Y. Zhao, Q. Chen, and J. Sun, "Dlfuzz: Differential fuzzing testing of deep learning systems," 2018.

[10] J. Gao, X. Yang, Y. Fu, Y. Jiang, and J. Sun, "Vulseeker: a semantic learning based vulnerability seeker for cross-platform binary," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 896–899.

[11] Y. Jiang, H. Song, Y. Yang, H. Liu, M. Gu, Y. Guan, J. Sun, and L. Sha, "Dependable model-driven development of cps: From stateflow simulation to verified implementation," *ACM Transactions on Cyber-Physical Systems*, vol. 3, no. 1, p. 12, 2018.

[12] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[13] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser, "Symbolic execution for software testing in practice: preliminary assessment," in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 1066–1071.

[14] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," *Acm Sigplan Notices*, vol. 47, no. 6, pp. 193–204, 2012.

[15] Q. Yi, Z. Yang, S. Guo, C. Wang, J. Liu, and C. Zhao, "Eliminating path redundancy via postconditioned symbolic execution," *IEEE Transactions on Software Engineering*, 2017.

[16] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, "Fitness-guided path exploration in dynamic symbolic execution," in *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*. IEEE, 2009, pp. 359–368.

[17] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*. IEEE, 2008, pp. 443–446.

[18] 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.

[19] T. Chen, X.-s. Zhang, S.-z. Guo, H.-y. Li, and Y. Wu, "State of the art: Dynamic symbolic execution for automated test generation," *Future Generation Computer Systems*, vol. 29, no. 7, pp. 1758–1773, 2013.

[20] T. Hansen, P. Schachte, and H. Søndergaard, "State joining and splitting for the symbolic execution of binaries," in *International Workshop on Runtime Verification*. Springer, 2009, pp. 76–92.

[21] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *Proceedings of the 19th International Conference on Computer Aided Verification*, ser. CAV'07, 2007, pp. 519–531.

[22] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[23] 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, p. 10, 2008.

[24] P. Godefroid, M. Y. Levin, D. A. Molnar *et al.*, "Automated whitebox fuzz testing." in *NDSS*, vol. 8, 2008, pp. 151–166.

[25] S. Khurshid, C. S. Păsăreanu, and W. Visser, "Generalized symbolic execution for model checking and testing," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2003, pp. 553–568.

[26] E. M. Clarke, D. Kroening, and F. Lerda, "A tool for checking ansi-c programs," *tools and algorithms for construction and analysis of systems*, vol. 2988, pp. 168–176, 2004.

[27] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, "Enhancing symbolic execution with veritesting," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 1083–1094.