ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/372311330

Structural Test Input Generation for 3-Address Code Coverage Using Path-Merged Symbolic Execution

Conference Paper · May 2023

DOI: 10.1109/AST58925.2023.00012

CITATIONS
0

READS
51

5 authors, including:

Soha Hussein
University of Minnesota Twin Cities
11 PUBLICATIONS **84** CITATIONS
SEE PROFILE

Stephen McCamant
University of Minnesota Twin Cities
75 PUBLICATIONS **4,124** CITATIONS
SEE PROFILE

Vaibhav Sharma
University of Minnesota Twin Cities
21 PUBLICATIONS **208** CITATIONS
SEE PROFILE

All content following this page was uploaded by Soha Hussein on 26 August 2023.

The user has requested enhancement of the downloaded file.

Structural Test Input Generation for 3-Address Code Coverage Using Path-Merged Symbolic Execution

Soha Hussein[‡]*, Stephen McCamant*, Elena Sherman[†], Vaibhav Sharma* and Mike Whalen*
*Department of Computer Science, University of Minnesota, USA
Email: soha@umn.edu, mccamant@cs.umn.edu, vaibhav@umn.edu, mwwhalen@umn.edu
[†]Department of Computer Science, Boise State University, USA
Email: elenasherman@boisestate.edu

***Abstract*—Test input generation is one of the key applications of symbolic execution (SE). However, being a path-sensitive technique, SE often faces path explosion even when creating a branch-adequate test suite. Path-merging symbolic execution (PM-SE) alleviates the path explosion problem by summarizing regions of code into disjunctive constraints, thus traversing at once a set of paths with the same prefixes. Previous work has shown that PM-SE can reduce run-time up to 38%, though these improvements can be impaired if the summarized code results in complex constraints or introduces additional symbols that increase the number of branching points in the later execution.**

Considering these trade-offs, examining the ability of PM-SE to generate branch-adequate test inputs is an open research problem. This paper investigates it by developing a technique that extracts structural coverage-related queries from disjoint constraints. Using this approach, we extend PM-SE to generate branch-adequate test inputs.

Experiments compare the effectiveness and efficiency of test input generation by SE and PM-SE techniques. Results show that those techniques are complementary. For some programs, PM-SE yields faster coverage, with fewer generated tests, while for others, SE performs better. In addition, each technique covers branches that the other fails to discover.

I. Introduction

Traditional symbolic execution (SE) is a widely used verification technique that explores the program paths symbolically. Besides verifying program properties, SE is extensively used for generating program test inputs. However, SE's performance is hindered by the enormous number of paths it needs to explore, i.e., the *path-explosion* problem.

One of the techniques that aim at alleviating this problem is path-merging. *Path-merged symbolic execution (PM-SE)* is an extension of SE that traverses a set of paths at once. In PM-SE, execution paths are collapsed and summarized into a disjunctive logical constraint that describes the behavior of different paths within a code region. Thus, PM-SE works similarly to SE but looks for opportunities to collapse regions of code into logical constraints instead of traversing them individually. Previous work [1] shows that PM-SE can substantially speed up SE, e.g., in some cases reducing the running time by 38%. However, PM-SE has two shortcomings that could affect its performance: PM-SE increases the complexity of constraints due to expressing larger code with disjunctions, and PM-SE summarization encoding can introduce new symbolic variables, which PM-SE could use for additional future branching.

Our interest in this paper is to generate test inputs for various *structural* coverage criteria of control flows such as branch coverage, or the modified condition and decision coverage (MCDC) [2] criterion. Throughout the paper, we use the term *obligations* to refer to any structured coverage target in the code. For example, to achieve branch coverage, each side of each branch in the code represents an obligation. To generate test inputs using SE, one can easily compute the satisfying assignment (model) of the path constraint at the end of every execution path. This approach creates test inputs that are path adequate. While this coverage criterion subsumes other criteria [3], it is not practical as it can generate an exponential number of test inputs. Unfortunately, generating branch-adequate test inputs using SE could in the worst case degrade to the exploration of all paths. That is because, without a non-trivial analysis, only exploring all path SE can rule out infeasible coverage targets. Thus, while it can reduce the number of generated inputs from path-adequate to branch-adequate, it might not reduce the number of explored paths.

On the other hand, generating test inputs that are branch adequate is not as straightforward for PM-SE. PM-SE cannot reason about each branch separately, since the effectiveness of the path merging comes from summarizing branches in a single disjunctive constraint.

One way for PM-SE to generate test inputs that target uncovered branches is to disable the path merging for code regions containing them and enable it otherwise. Unfortunately, path-merging benefits are lost for the paths that cover a single branch outcome of a conditional statement. Thus, the challenge is to enable branch-adequate test input generation while preserving the run-time efficiency of the path merging.

Our proposed technique solves this problem by following this high-level idea. When PM-SE is about to collapse a region of code into a region summary, it checks whether this code region contains any uncovered obligations. If it does, it expands its complex conditions, if any, into multiple nested constraints. Then, it assigns each obligation condition to a fresh boolean symbolic variable. A value of `true` for such a variable represents coverage of its obligation.

At the end of an exploration, the technique conjoins a disjunction of obligation variables to the resulting path-constraint

[‡]Ain Shams University, Egypt
Lecturer on leave of absence (soha.hussien@cis.asu.edu.eg)

formula. It checks for the satisfiability of the augmented formula. If it is satisfiable, it obtains the model as a test case and checks off obligations that have "true" assignments for their symbolic variables. The technique then removes those symbolic variables from the disjunction.

The process is then repeated over the remaining uncovered obligations. Once the constraint becomes unsatisfiable, indicating no more coverage can be produced from this execution path, PM-SE then resumes execution by backtracking and exploring another path.

To decrease the number of calls to the solver, our implementation takes advantage of the satisfiability calls made during exploration to determine some test inputs. Also, our implementation skips covered obligations from marking during the merging process, thus reducing the overall complexity of the resultant disjunctive constraint.

We implement our technique as an extension to Java Ranger (JR) [1], a path merging symbolic execution tool for Java bytecode programs based on Symbolic Java PathFinder (SPF) [4]. We evaluate our approach using eight SPF/JR benchmarks to generate branch-adequate test inputs. Our approach handles partially summarized regions, regions with multiple `return`-statements, and complex conditions.

Our evaluations show that out of 8 programs JR achieves better coverage for 3, generates fewer test inputs for 3 and improves performance for 4 programs when compared with SPF. Overall we conclude that JR and SPF are complementary approaches, whose performance depends on program structure and constraint complexity.

This paper makes the following contributions:

- We present a new approach that generates test inputs with PM-SE by defining and collecting the control-flow structured obligation coverage.
- We use this approach to obtain branch coverage for three-address code and implement it as an extension to Java Ranger (JR): a PM-SE for Java.
- We evaluate JR's ability to generate branch-adequate test inputs for eight benchmarks and compare the results with traditional SE.

This paper is organized as follows. Sec. II discusses some background and presents an example. Sec. III presents our technique for PM-SE obligation adequate test input generation. Sec. IV shows how the technique can use it to collect branch coverage for three address code . Finally, Sec. V, Sec. VI, and Sec. VIII discuss our evaluation, related work, and conclusion.

II. Background and a Motivating Example

A. Background

Java Ranger (JR) [1], [5] is a path merging symbolic execution tool for Java programs. It is implemented as an enhancement of Symbolic PathFinder (SPF) [4], which is a traditional symbolic execution for Java programs. JR works by intercepting symbolic branches that SPF is about to execute, and trying to collapse paths from that instruction until its immediate post-dominator. This region of code is called a *multi-path* region. JR collapses paths in the multi-path region by describing its behavior as a disjunctive constraint. First, JR

```
1.  public int getSetBits(int i) {
2.     int numOfSetBits = 0;
3.     while (i != 0) {
4.       if ((i & 15) != 0) // oblg_4_TK/NT
5.         numOfSetBits += count4Bits(i);
6.       i = (i >>> 4);    }
7.     return numOfSetBits;}

8.   public int count4Bits(int i) {
9.     int count = 0;
10.    if ((i & 1) == 1) // oblg_10_TK/NT
11.       count++;
12.    if ((i & 2) == 2) // oblg_12_TK/NT
13.      count++;
14.    if ((i & 4) == 4) // oblg_14_TK/NT
15.      count++;
16.    if ((i & 8) == 8) // oblg_16_TK/NT
17.      count++;
18.     return count;  }
```

Fig. 1: Computing the number of set bits in a 32-bit integer by iteratively counting those set in the 4 right-most bits.

```
(i != 0) ∧ (t1 == (i & 15)) ∧ (t1 != 0) ∧ ···
```

Fig. 2: Partial path condition when executing lines 2, 3, 4, 9 and before executing line 10 in Fig.1 using SE.

translates control flow graph of multi-path region to its Intermediate Representation (IR). Then, JR uses multiple transformations to translate Java features into disjunctive constraint. At that point, JR conjoins the disjunctive constraint onto the path condition, updates the state, and points SPF to execute the instruction right after the immediate post-dominator.

A single exploration in SE, a *path*, is an execution from the beginning of the program to the termination, where at each branch, a single side is executed. On the other hand, a single exploration in PM-SE, is an execution from the beginning of the program to termination, where some regions may have been summarized as a disjunctive constraint.

B. Motivating Example

Consider the code in Figure 1, where comments are used to label branch obligations. We use `taken (TK)`, and `not taken (NT)` to refer to the "then" and the "else" branches of a conditional statement.

The program counts the number of set bits in a 32-bit integer that it takes as an argument i, and returns this count in `numOfSetBits`. First, the program checks whether i is not zero (line 3), and then it checks whether the last four bits of i are not all zeros (line 4). After ensuring that at least one of the last four bits is set to one, it invokes the method `count4Bits` at line 5. `count4Bits` uses a bit-wise `and` operator to determine whether a bit at a specific index location is not zero and updates `count`. On returning from `count4Bits`, the program updates its local count and shift bits of i to the right by four places so that it can look for set bits in the next four bits.

When SE interprets this code, at each conditional statement, SE must choose at most one branch at a time to explore.

```
(t1 == (i & 15) ∧
  (((((t2 == (i & 1)) ∧
    (((t2 == 1) ∧ (count1 == 1)) ∨
    ((t2 != 1) ∧ (count1 == 0)))) ....)
```

Fig. 3: Snippet of the JR's disjunctive constraint for merging lines 4–5 in getSetBits on the first iteration of the while-loop.

```
1  t1 := i & 15                  //line 4
2  t2 := i & 1                   //line 10
3  count1 := γ(t2==1, 1, 0)
4  t3 := i & 2                   //line 12
5  count2 := count1 + 1          //line 13
6  count3 := γ(t3==2, count2, count1)
7  t4 := i & 4                   //line 14
8  count4 := count3 + 1          //line 15
9  count5 := γ(t4==4, count4, count3)
10 t5 := i & 8                   //line 16
11 count6 := count5 + 1         //line 17
12 count7 := γ(t5==8, count6, count5)
13 numOfSetBits1 := 0 + count7 //line 5
14 numOfSetBits2:=γ(t1==0, numOfSetBits1, 0)
15 ----------------------------------------
16 oblg_4_TK1 := γ(t1==0, 0, 1)
17 oblg_4_NK1 := γ(t1==0, 1, 0)
18 oblg_10_TK1 := γ(t2==1, 1, 0)
19 oblg_10_NK1 := γ(t2==1, 0, 1)
20 oblg_10_TK2 := γ(t1==0, oblg_10_TK1, 0)
21 oblg_10_NK2 := γ(t1==0, oblg_10_NK1, 0)
```

Fig. 4: Upper part: JR IR representation, just before transforming it into a disjunctive constraint. Lower part: a snippet of the amended IR for identifying obligation coverage. `oblg_10_TK2` and `oblg_10_NT2` used to propagate the condition of `t1` for the satisfiablity of `oblg_10_TK` and `oblg_10_NT`.

For example, an exploration that is traversing the program by executing **lines 2, 3, 4, 9, 10, 11, 12, 14, 16, 18, 5, 6, 3** and **7**, is a path that SE can explore. Note that at each condition SE only follows a single branch, and the constraint describing the taken side must be conjoined to the path constraint (PC). For example, Fig. 2 shows the PC when SE is about to execute **line 10** in the above path [1]. Thus, to generate a test input to cover a specific branch, e.g., the taken branch `oblg_10_TK` of the conditional statement on line 11, upon checking its feasibility SE follows that taken branch and generates the corresponding PC. After solving the PC, SE obtains test inputs that guarantee to cover `oblg_10_Tk`.

Even though generating test inputs using SE is straightforward, in our experiment, after running for one hour SPF generates 28,636 path constraints (PCs) for this example program while producing 6 test inputs covering all 12 branch obligations. PM-SE approaches PCs generation differently. Fig. 3 depicts the resulting single disjunctive PC that JR generates by merging lines 4 through 5 (including method invocation). Here, **count1** is an additional intermediate variable defined by JR'IR. With such compact representation, PM-SE loses the one-to-one relationship between a PC and branch obligations, which makes computing branch coverage more challenging. Clearly, a disjunctive constraint describing the code region retains no information about coverage targets.

To explain the proposed approach for generating branch-adequate test inputs from such constraints, let's examine Figure 4. The upper part of Figure 4 shows JR's IR, in 3-address code [6](page 466), before its translation to this constraint. In Fig. 4, `i` is the symbolic input, and `t1-t5` are fresh temporary variables capturing the side effects of the `if-statements` conditions on **lines 4, 10, 12, 14**, and **16** respectively in Figure 1. In this IR, variable numbering is used to identify variables uniquely, such as in `count1 - count7`, i.e., similar to the gated single static assignment [7] of various variables. **Lines 3, 6, 9, 12, 14** in Figure 4 capture the value of the variables `count`, and `numOfSetBits` after the `if-statements` using a γ-expression. A ($\gamma(cond, e_1, e_2)$) in JR's IR is a conditional expression that evaluates to e_1 if the condition ($cond$) holds or to e_2 otherwise. For example, the value of `count3` at line 6 in Fig. 4 either increments the last value (**count2**) by 1, or holds the value of the last count, i.e., `count1`.

Similar to SE, PM-SE checks the satisfiability of the generated disjunctive PC to decide whether to proceed with the program execution or not. As the example demonstrates, without some mapping between the coverage targets within the collapsed region to the logical expression in the constraint responsible for its coverage, PM-SE has no information about what branch obligations are covered by a given satisfiable PC assignment.

To overcome this challenge, our extension tracks branch obligations using assignments to special obligation variables. An *obligation variable* is an internal IR variable associated with a particular source obligation. For example, the lower part of Fig. 4 (**lines 16-21**) shows a snippet of the obligation mapping created by the extension (the entire obligation marking is removed for simplicity). Using this information, PM-SE looks up the values of various obligation variables in a satisfiable assignment. It uses the values of these obligation variables to determine the coverage information of their corresponding obligations. For example, if a satisfiable assignment evaluates to true the obligation variable `oblg_4_TK1` then the corresponding taken branch obligation at **line 4** is covered by that test input. Since a summary might have multiple obligations covered, our technique checks and marks off obligations within the summarization. We implemented this technique as an extension to JR. Unlike SE, JR finished executing the same program in 6 seconds while generating 9 PCs and producing 5 test inputs that cover 12 branch obligations. In the next section, we present our approach in detail, including algorithms that can be implemented within a tool supporting PM-SE.

III. PM-SE Obligation Adequate Test Input Generation

A. Overview of the Approach

When PM-SE executes the program in Fig. 1, it identifies **lines 4–5** as a multi-path region. Note that this multi-path region also includes the summarization of the method invocation of `count4Bits(int i)` at **line 5**.

[1]We use ∧ and ∨ for logical conjunction and disjunction.

Inside a multi-path region, there could be multiple obligations. For example, if branch coverage is the targeted coverage criterion, then we need to cover both the `Taken (TK)` and the `Not Taken (NT)` branches of conditional statements within the multi-path region. For example, covering the branches of the conditional statement on **line 4**, requires two obligations to be covered: one where the condition `(i & 15 != 0)` evaluates to `true` and another when the condition `(i & 15 != 0)` evaluates to `false`. To label the obligations of conditional statements inside a multi-path region, our technique introduces *obligation variables*: boolean auxiliary symbolic variables for each branch. For example, the two obligation variables introduced to cover the branch **line 4** of Fig. 1: `oblg_4_TK` and `oblg_4_NT`, to cover the true and the false obligations, respectively. A value of `true` for an obligation variable indicates that its corresponding side of the symbolic branch is covered. Similarly, we label obligations in the `count4Bits` method because it is invoked at **line 5**.

When PM-SE attempts to merge a region, our technique creates these auxiliary variables for uncovered obligations, in the PM-SE state, and assigns them to true in the appropriate control flow. Note that these variables are internal to JR's, i.e., they are not forked on, and thus they do not introduce additional future paths. Then, PM-SE performs its customary path merging. At the end of a path PM-SE creates a new *obligations clause*, a disjunction of all uncovered obligation variables created along that path. The algorithm conjoins this clause with the PC and queries the underlying SMT solver for the satisfiability of this conjunction. If the conjunction is satisfiable, then at least one obligation variable in the obligations clause must have been assigned to true. The algorithm uses the satisfying assignment, i.e., the model, to inquire about obligation variables with true values. It records their corresponding obligations as covered and extracts the covering test input from the model.

To obtain different assignments where other obligation variables evaluate to true, the algorithm removes previously covered obligation variables from the obligation clause and repeats the process. This process continues until either all the auxiliary variables have an assignment with true value, thereby covering all branch obligations, or the SMT solver returns unsatisfiable, indicating that no more obligation variables can be covered in the current PC. Then, PM-SE backtracks to traverse another set of paths or terminates if all of them are explored. In the latter case, uncovered branches can be marked as infeasible. For simplicity, PM-SE defaults to using SE to collect their coverage for obligations that are explored but are not part of a merged region, i.e., marking coverage once execution of the taken or not-taken instruction.

For example, when PM-SE analyzes the example in Fig. 1, and attempts to collapse the multi-path region (in **lines 4–5** including the summarization of the entire code of `count4Bits`), the test input generation extension creates obligation variables (indicated in **green**) in Fig. 1 and assigns them to true where needed. Then at the end of the path, our algorithm examines covered obligations. Our approach attempts to check for coverage of all obligations enclosed within the path-merged region. This includes, on the first iteration, checking for coverage of `(oblg_4_TK ∨ oblg_4_NT ∨ oblg_10_TK ∨ oblg_10_NT ∨ oblg_12_TK ∨ oblg_12_NT ∨ oblg_14_TK ∨ oblg_14_NT ∨ oblg_16_TK ∨ oblg_16_NT)`. We omit the unique representation of obligation variables for simplicity.

The algorithm conjoins this new clause with the PC and queries the SMT solver for a satisfying assignment. Let us assume that the returned satisfying assignment has $oblg\_4\_TK$ and $oblg\_10\_TK$ being set to `true`. This indicates that their corresponding branches are covered and that the solver's assignment of values to the method inputs constitutes a test input. To obtain additional test inputs that cover additional obligations, the algorithm removes these two obligation variables from the obligation clause resulting in `(oblg_4_NT ∨ oblg_10_NT ∨ oblg_12_TK ∨ oblg_12_NT ∨ oblg_14_TK ∨ oblg_14_NT ∨ oblg_16_TK ∨ oblg_16_NT)`. The PM-SE extension conjoins this new clause with the PC and queries the SMT solver once again for a satisfying assignment. If at some point the SMT solver returns unsatisfiable, then it means that no more obligations can be covered along this path due to conflicting clauses, and thus PM-SE resumes program exploration until either all obligations are covered, or all paths are analyzed.

B. *Algorithm Details*

Alg. 1 outlines the main steps of symbolically executing a program using PM-SE, including test input generation extension. Besides symbolically executing the program, the algorithm returns the set of test inputs T that cover obligations during execution. The algorithm inputs are:

- s_0: the initial statement to be executed.
- i: the set of symbolic input parameters
- Θ: the set of all coverage obligations. For branch coverage, there are two obligations per branch.
- $\mathbb{T}$: the set of all values
- V: a map of all valuations

The algorithm uses the following data structures:

- Δ: the set of symbols
- Π: the path condition (PC) created for the current path
- Θ_c: the set of covered obligations
- Σ: map between obligation variables and obligations. Note that an obligation variable must map to a single obligation, while each obligation can be satisfied by multiple obligation variables. For example, in the first two iterations of the `while-loop` in Fig. 1, each time when PM-SE collapses **line 4–5**, it introduces new obligation variables to represent the two sides of coverage targets of each branch, i.e., to cover when `((i & 15 ) != 0)` holds, it creates two obligation variables `oblg_4_TK1` and `oblg_4_NT1`. Then, in the second iteration it creates `oblg_4_TK2` and `oblg_4_NT2`. Note that, the satisfiablity of either of `oblg_4_TK1`, or `oblg_4_TK2` indicates the coverage of the obligation `oblg_4_TK`.
- $\mathbb{W}$: the worklist set of 4-tuple. A 4-tuple consists of the next statement to be executed (Smt), along with the path

Algorithm 1: General Alg. for defining, collecting, and generating test cases for any coverage criteria using PM-DSE. Our extension is highlighted in grey.

```
   input: Initial Stmt $s_0$: IR $Stmt$
   input: Set of symbolic input parameters $i$
   input: Set of all obligations $\Theta$
   input: Set of values $\mathbb{T}$
   input: Valuations map $V : \Delta \rightharpoonup \mathbb{T}$
   Data: Set of symbols $\Delta$
   Data: Path predicate $\Pi : V \to \mathbb{B}$
   Data: Set of covered obligations $\Theta_c \subseteq \Theta$
   Data: Map from obligation variables to obligations
         $\Sigma : \Delta \to \Theta$
   Data: Worklist $\mathbb{W} \subseteq Stmt \times \Pi \times \Delta \times \Sigma$
1  $\mathbb{W} \leftarrow \{(s_0, \top, i, \emptyset)\}$
2  $\Theta_c \leftarrow \emptyset$
3  while $\mathbb{W} \neq \emptyset$ do
4      $(w, \mathbb{W}) \leftarrow pickNext(\mathbb{W})$
5      $s \leftarrow w[0]$   $\pi \leftarrow w[1]$   $\delta \leftarrow w[2]$   $\sigma \leftarrow w[3]$
6      switch type(s) do
7          case if-stmt do
8              $s_{next}, \pi', \delta', \sigma' \leftarrow$ merge($w$)
9              $\mathbb{W} \leftarrow \mathbb{W} \cup (s_{next}, \pi', \delta', \sigma')$
10         end case
11         case halt do
12             $T, \Theta_c \leftarrow$ collectTestInputs($w, \Theta_c, i$)
13         end case
14         otherwise do
15             $s_{next}, \pi', \delta' \leftarrow$ resume SE($s, \pi, \delta$)
16             $\mathbb{W} \leftarrow \mathbb{W} \cup \{(s_{next}, \pi', \delta', \sigma)\}$
17         end case
18     end switch
19 end while
20 return $T$
```

predicate obtained so far (Π), the set of symbols (Δ), and the map of obligation variables to obligations (Σ).

The algorithm starts by initializing $\mathbb{W}$ with the initial statement (s_0), a true path predicate $\top$, a set of the symbolic parameters (i), and an empty map form obligation variables to obligations Σ (**line 1**). The set of covered obligations is initialized to the empty set (**line 2**). The `while`-loop in **lines 3–20** iterates over the worklist elements until it is empty. In **lines 4–5** the algorithm selects the next element from the worklist w, and renames the remaining elements to $\mathbb{W}$. and the elements of its tuple are assigned to the following variables: statement s, path predicate π, set of symbolic variables δ, and the mapping of symbols to obligations σ. Depending on the type of s: whether it is an `if-statement`, a program termination statement (`halt-statement`), or any other statement, the program proceeds to execute one of these three cases. If s is an `if-statement` (**lines 7–10**), then the algorithm invokes an extension of the merging process of PM-SE, which generates the disjunctive constraint that describes s. Besides updating s and π, the extension generates new obligation variables and creates a set of tuples σ' that maps those variables to obligations. It assigns updated values to a 4-tuple on line 8, which then on line 9 is added to the worklist.

When s is a `halt` statement, indicating the end of a path, then the extension collects test inputs of all obligations that could be covered along the traversed path (**line 12**). This is done by constructing the obligation clause and by using multiple solver queries. The paper describes this process later in this section. Finally, if s is any other statement, then it is symbolically executed with traditional SE, and the state is updated accordingly (**lines 14–17**).

Algorithm 2: Expanding the extended merging process for PM-SE. Our extension is highlighted in grey

```
   input: A worklist element $w \in \mathbb{W}$
1  $w' \leftarrow w$
2  repeat
3      $w \leftarrow w'$
4      $s \leftarrow w[0]$   $\pi \leftarrow w[1]$   $\delta \leftarrow w[2]$   $\sigma \leftarrow w[3]$
5      $s' \leftarrow$ expand conditions($s$)
6      $s_{oblg}, \delta_{oblg} \leftarrow$ mark oblg($s, \delta$)
7      $s_{flat}, \delta_{flat} \leftarrow$ inline invocations($s_{oblg}, \delta_{oblg}$)
8      $s_{gsa}, \delta_{gsa}, \sigma_{gsa} \leftarrow$ create oblg. GSA($s_{flat}, \delta_{flat}, \sigma$)
9      $s', \delta', \sigma' \leftarrow$ eliminate ref.($s_{gsa}, \pi, \delta_{gsa}, \sigma_{gsa}$)
10     $w' \leftarrow (s', \pi, \delta', \sigma')$
11 until $w == w'$
12 $w'$ = handle exceptions ($w$)
13 $e$ = generate constraint ($w'$)
14 populate outputs($w'$)
15 return (next($s$), $w'[1] \wedge e, w'[2], w'[3]$))
```

IV. Branch Coverage of Three Address Code

In this section, we illustrate how we use the general Alg. 1 in computing test inputs for branch coverage. In particular, we show how we achieve coverage of branches in merged regions with complex conditions, how we mark obligations , how we collect test inputs at the end of a path, and finally, how we adapt various optimizations to minimize the number of solver calls and reduce the complexity of the solver queries.

Alg. 2 shows, in black, the main steps for merging a code region into a logical constraint. To do that PM-SE runs an iterative computation, where it keeps eliminating one Java feature after another from the statement s. More precisely, it inlines method invocations, and it eliminates references. When no change is detected, the algorithm then eliminates exceptions, generates the disjunctive constraint e describing paths within the region, updates the heap and stack and finally, it returns the new state after conjuncting the disjunctive constraint e onto the path condition π. Note that the function `next(`s`)` returns the next sequentially composed statement to s. For more information, readers are referred to [1].

Statements for TCG extensions are emphasized in grey. This includes expanding complex conditions with multiple booleans into an equivalent single boolean statement, marking obligations that need coverage, and finally, uniquely identifying obligation variables by generating their corresponding GSA. In the rest of this section, we detail each of these steps.

```
1 t2 := (i & 1 );
2 if ( t2 == 1 ) {
3   oblg_6_TK := 1
4   counter1 := (0 + 1 )
5 } else
6   oblg_6_NT := 1
7 ... // remaining IR omitted
```

Fig. 5: Snippet of IR after marking of obligations

A. Expanding Conditions

Our technique works on decompiled IR statements from the program's bytecode. Since the decompilation step might create complex conditions (conditions with "and" and "or" operations), expanding these conditions allows our technique to distinguish the obligations of various simple branches. Our extension removes complex conditions from the IR statement s by expending the conditions. This facilitates isolating the obligation of each condition which helps in the marking step to be discussed next. To expand the conditions we use the following rewrite rules:

(1) `if` $(e_1$ && $e_2)$ s_1 `else` $s_2 \rightarrow$ `if` (e_1) { `if` (e_2) s_1 `else` s_2} `else` s_2

(2) `if` $(e_1 \mid\mid e_2)$ s_1 `else` $s_2 \rightarrow$ `if` (e_1) s_1 `else` { `if` (e_2) s_1 `else` s_2 }

B. Marking Obligations

Now that each if-statement has a single condition, we can start marking their corresponding obligations. We divide the marking process into three steps: (1) identification of obligations within a statement s, i.e., the statement that is about to be collapsed as a single logical constraint, (2) addition of obligation variables to s, and (3) transformation of s to create Gated Single Static Assignment (GSA) for obligation variables. We discuss each of these steps:

(1) Identifying obligations: this step identifies all obligations Θ to include in s. We do that by locating all branches in s and creating a label for them using their location in the bytecode. For example, to do the path-merging in Fig. 1 PM-SE creates obligation labels (`oblg_4, oblg_10, .. , oblg_16`).

(2) Introducing obligation variables: this step creates obligation variables for each obligation that is not covered yet, and maps them to their obligation labels.

This mapping is important, as it keeps track of the coverage of an obligation once an obligation variable has been satisfied. For example, we create two new obligation variables `oblg_6_NT`, and `oblg_6_TK`, and map them with their corresponding obligation labels (in σ) from the previous step.

Finally, we insert assignment statements within the structure of s where these obligations are true. For example, we insert the assignment statements in **line 3**, and **line 6** in Fig. 5.

(3) Creating a Gated SSA: In this step, we generate the GSA form [7] for obligation variables. For example, Fig. 6 shows a snippet of the resultant IR after this step. Here besides creating unique names for variables, we introduce γ-expressions at the joining point of each branch. The appended unique number to the obligation variable can be seen in `oblg_6_NT1` instead of

```
1  t1 := (i & 15 )
2  if (! ( t1 == 0 )) {
3    oblg_10_TK1 := 1
4    t2 := (i & 1 )
5    if ( t2 == 1 ) {
6      oblg_6_TK1 := 1
7      count2 := 0 + 1
8    } else {
9      oblg_6_NT1 := 1
10   }
11   oblg_6_TK2 := γ(t2==1, oblg_6_TK1, 0)
12   oblg_6_NT2 := γ(t2==1, 0, oblg_6_NT1)
13 ...} // remaining IR omitted
```

Fig. 6: Snippet IR after the creation of GSA

Algorithm 3: Generating test inputs for branch coverage

```
   input: worklist at the end of the path $w \in \mathbb{W}$
   input: covered obligations $\Theta_c$
   input: input parameters $i$
1  $\pi \leftarrow w[1]$    $\sigma \leftarrow w[3]$
2  repeat
3      $\delta_{uncovered} \leftarrow \{a \mid (a : \Delta, o : \Theta) \in \sigma \wedge o \notin \Theta_c\}$
4      oblg$_{clause}$ = $\bot \bigvee \delta_{uncovered}$
5      (isSat,$M$) $\leftarrow$ check sat ($\pi \wedge$ oblg$_{clause}$)
6      if isSat then
7          $\Theta_c \leftarrow \Theta_c \cup$ extract coverage($M, \delta_{uncovered}$)
8          $T \leftarrow T \cup$ extract parameter values ($M, i$)
9      end if
10 until not isSat
11 return $T, \Theta_c$
```

`oblg_6_NT`. While the introduced γ-expressions can be seen in **lines 10,** and **11**. These expressions enable the propagation of the path constraint to obligation variables.

For example, in **line 10** indicates that `oblg_6_TK2` can be true (by transitivity) if `t2 == 1`.

Observe that we also update the set of obligation variables to obligations (Σ) to preserve the relationship between the two.

At this point, Alg. 2 has encoded the conditions that constrain the satisfiability of obligation variables. As this IR translates to a disjunctive constraint in **line 13**, the satisifiability constraints for obligation variables are encoded within it. This encoding allows us to generate test inputs by looking for satisfied obligation variables at the end of an execution path.

C. Collecting Test Inputs

At the end of a path on line 12 of Algorithm 1, the extension identifies all newly covered obligations. To do that, it constructs the obligation clause and queries an SMT solver about the satisfiablity of the resulting constraint conjuncted with the path condition π.

It does it by collecting the set of all uncovered obligation variables (**line 3** Alg. 3). Next, it constructs the obligation clause by disjoining $\bot$ and all obligation variables (**line 2**. Then it checks the satisfiability of the obligation clause conjuncted with the path condition π **line 5**. The result is a boolean variable `isSat`, and the model M.

If the query is unsatisfiable, then the algorithm updates the set of covered obligations (**line 7**) by checking which uncovered obligations are true in M. Then, it extracts the values of the input parameter i from the model and updates the set of test inputs T (**line 8**). This process continues until no more obligations can be satisfiable/covered, in which case, the algorithm returns the updated test inputs T and the updated set of covered obligations Θ_c (line 11).

To see how this works using our motivating example, let us assume that we come at the end of the first execution path, where the instantiated summary in Fig. 4 is collapsed within the path condition. At that point, we construct an obligation clause of the form: $\perp \vee$ `oblg_4_TK1` $\vee$ `oblg_4_NT1` $\vee$ `oblg_10_TK2` $\cdots \vee$ `oblg_16_NT2`. Then, we conjoin the obligation clause with the path condition, and check the satisfiability of the resulting constraint. If the query is satisfiable, then at least one of the obligation variables is true. Let us assume that `oblg_4_TK1`, and `oblg_10_TK2` were satisfied. At this point, we use the solver's model to check the values of the input parameters; let us assume that the input parameter (i) in Fig. 4 is $i = 1$, and that the set of test inputs is T. Finally, we add these covered obligations from the set of covered obligations Θ_c to become $\{$`oblg_4_TK1`, `oblg_10_TK2`$\}$.

To reduce the complexity of the obligation annotated queries, we only perform the processes of removing complex conditions, marking obligations, and creating obligation GSA for *uncovered* obligations. Also, to minimize the number of solver calls needed to collect test inputs, we use the solver models for any query that is checked along the path. If any of the obligation variables evaluate to true, then we mark off their corresponding obligation. This allows us to minimize the number of solver queries at the end of the path.

When Alg. 3 returns the updated set of test inputs and updated set of covered obligations, Alg. 1 continues processing elements in the worklist W, until it becomes empty. Then, the algorithm terminates while returning the set of test inputs.

V. Evaluation

We implemented our technique for generating branch-adequate test inputs as an extension to JR [5] (accessible through GitHub [8]). To compare it with SE, we also updated the existing SPF feature for generating path-adequate test inputs to generate branch-adequate ones; we call it branch test input generation, BTIG. For that, we updated SPF to maintain a set of all covered obligations. Then, we collect covered obligations when SPF executes their corresponding taken or not-taken instructions. Since our evaluation is comparing SPF's and JR's BTIG extension, we will refer to these extensions as SPF and JR, respectively, in the remainder of this section.

BTIG also attempts to reduce the number of tests that SPF finally outputs. Instead of obtaining a solution at each satisfiabilty check during a path exploration, BTIG collects only one test input at the last staisfiability check. This is sound because the last generated test input on a pat implies the coverage of all encountered obligations along the same path.

The JR extension also uses our SPF's BTIG for obligations appearing on non-merged paths. Both SPF and JR are configured to stop running after all branches are covered. However this does not improve performance in practice, as there are usually obligations that cannot be covered, and both techniques need to complete full path exploration to confirm this.

TABLE I: Benchmarks borrowed from JR [1]

Benchmark	**SLOC**	**# classes**	**# methods**
ApacheCLI	3612	18	183
NanoXML	4610	17	129
TCAS	300	1	12
WBS	265	1	3
Schedule	306	4	27
Siena	1256	10	94
PrintTokens	570	4	30
replace	795	1	19

Table I describes the set of 8 benchmarks used in evaluating JR [1], omitting for MerArbiter, as our current implementation does not support collecting obligations for inner classes. Both SPF and JR extensions use the default search strategy in SPF, which is depth-first search. Note that each benchmark can have its arguments processed by SPF/JR as either concrete or symbolic. We use this configuration in our experiments to control complexity and thus the run-time of the experiments; more symbolic inputs mean a harder program analysis problem. Also, we use the same configuration of symbolic/concrete inputs for both SPF and JR to ensure that the target coverage problem is the same for both tools. To reduce the cache effect, in our experiments, we ran each configuration three times and report the results of the last run.

We used Z3 [9] with the bit-vector theory as the underlying solver for representing and solving constraints during the execution of both JR and SPF. The experiment machine ran Ubuntu 16.04.6 on a 3.6 GHz Intel Core i7-7700 CPU processor with 32 GB RAM. We used a 12GB Java heap size.

To evaluate PM-SE's ability to generate test inputs, we answer the following research questions:

- RQ1: Compared to SPF, what is the JR's run-time overhead for generating branch adequate test inputs?
- RQ2: What is the effectiveness of the coverage over time for each technique?
- RQ3: Are SE and PM-SE complementary to one another?

A. *RQ1: JR's Performance Overhead*

To answer the first question, we ran SPF with BTIG, and the JR with BTIG, as well as all path-merging features, enabled. Here we configured each benchmark with the maximum number of symbolic inputs such that all runs terminate in less than an hour, covering the same obligations, then we compared performance. We used 4 symbolic inputs for ApachiCLI, Replace, and Siena, 5 symbolic inputs for NanoXML and Schedule, and 2, 11, and 9 symbolic inputs for PrintTokens, TCAS, and WBS, respectively. Also, an experiment consists of three consecutive runs of an approach on a given benchmark to eliminate any cache effect. We repeated each experiment three times, and computed the average time for the last run in each experiment Tbl. II together with other information. For each benchmark and analysis, the table shows the total number of paths explored (**paths**), the number of successful code merges by JR (**merges**), the total number of solver queries (**# queries**),

TABLE II: Performance of generating test inputs using SPF versus JR. The table shows the number of paths, successful path merging, solver queries, tests, as well as query solving time, average query time, total execution time, and time overhead.

benchmark	analysis	paths	merges	# queries	tests	solving time	query avg	total time
ApachiCLI	SPF	7278	-	139262	8	1619.9	0.012	1737.9
	JR	1950	7296	4286	8	389	0.091	412.2
NanoXML	SPF	38923	-	91554	46	1242.2	0.014	1391.6
	JR	6148	1602	19060	44	391.6	0.021	428
WBS	SPF	13824	-	27646	12	329.9	0.012	349.9
	JR	1	35	7	6	8.9	1.276	11.7
TCAS	SPF	200	-	1256	21	15.2	0.012	17.4
	JR	1	4	17	16	12.9	0.761	15.6
Schedule	SPF	16807	-	33612	7	386.8	0.012	415.9
	JR	16807	0	33612	7	395.1	0.012	427.8
Siena	SPF	20736	-	52780	6	645.3	0.012	698.9
	JR	20736	0	52780	6	645.9	0.012	703.3
PrintTokens	SPF	489	-	10616	43	134.3	0.013	141.7
	JR	393	451	9179	43	176	0.019	186.26
Replace	SPF	1314	-	11304	27	123.8	0.011	131.8
	JR	506	252	35136	36	2328.4	0.066	2358.7

the number of generated test inputs (**tests**), the average time in seconds spent on generating the equivalent SMT query, solving and examining the returned solver model (**solving time**), the average time in seconds per query (**query avg**), and finally, the average total analysis time in seconds (**total time**).

In Tbl. II we see that JR has significantly improved performance while generating fewer test inputs on four out of six benchmarks. This is observable on ApachiCLI, NanoXML, TCAS, and WBS. For ApachiCLI and NanoXML, reducing the number of explored paths (from 7278 to 1950) in ApachiCLI, and from (38923 to 6148) in NanoXML. We also observe that despite the average query time is up, it generally did not affect the overall performance of JR. Thus, the relatively expensive average solving time per query when comparing these two benchmarks (1.276s in WBS and 0.761s in TCAS) with the remaining benchmarks (less than 0.1s). This is expected due to the complexity of the constructed queries describing all paths. Interestingly, despite the large overhead, JR is still similar to or much faster than SPF.

For Siena and Schedule, JR has no successful merges; thus, the number of execution paths remains the same as SPF's. These two benchmarks suggest that the overhead needed for BTIG setup and generation of test inputs is less than 3%.

Replace's results fall into the other spectrum, where path merging is not beneficial as opposed to SPF. In Tbl. II, we see that SPF is 18 times faster than JR. That is, despite the reduction in the number of paths (1314 to 506 from SPF to JR), the overall running time increased (from 131.8s to 1810.7s). On analyzing this benchmark further we found that JR with the BTIG support yields an overhead of 30.26% over plain JR. This result is attributed to both a significant increase in the number of queries (from 11304 to 35163) as well as an increase in the overhead per query (from 0.051s to 0.066s).

In general, results from PrintTokens and Replace show that the performance of path merging can fluctuate widely. JR is faster than SPF by factors of 4x, 3x, 30x, 1.11x on ApachiCLI, NanoXML, WBS, and TCAS, while SPF is faster than JR by factors of 1.03x, 1.01x, 1.3x, 18x on Schedule, Siena, PrintTokens and Replace. The results suggest that the fluctuations are more related to the general performance of JR than it is related to BTIG extension [2]. This points to the importance of future research that defines and measures heuristics to distinguish useful path-merging opportunities.

TABLE III: Performance of SPF and JR test input generation. All times are in seconds.

benchmark	mode	paths	merges	queries	tests
ApachiCLI	SPF	12962	0	277540	12
	JR	10144	38718	25664	14
NanoXML	SPF	94190	0	223436	52
	JR	40733	10681	126496	53
WBS	SPF	138864	0	277744	12
	JR	1	65	6	5
TCAS	SPF	35532	0	254222	1
	JR	1	8	31	1
Schedule	SPF	135395	0	270818	8
	JR	134539	0	269106	8
Siena	SPF	104473	0	265938	10
	JR	104307	0	265516	10
PrintTokens	SPF	11921	0	249446	48
	JR	5866	5504	102361	53
Replace	SPF	51002	0	285516	16
	JR	5514	39	62770	21

B. RQ2: JR's Coverage Effectiveness Over Time

To answer the second research question, we changed the number of symbolic variables so that neither SPF nor JR can finish exploring all paths or achieving 100% branch coverage in 1 hour. We used a larger number of symbolic inputs to increase the chances of exploring new paths and thus cover more code. Specifically, we used the maximum existing configured symbolic inputs for ApachCLI, NanoXML, PrintTokens, Replace, Seina, and Schedule, 9, 9, 8, 11, 9, and 15. As WBS and TCAS describe reactive components with any number of iterations, we used the same number of symbolic inputs previously used in [1], 15 and 24. Results of this experiment are shown in Tbl. III, while accumulation of coverage over time is shown in Fig. 15, and finally, complementary coverage information is shown in Tbl. IV.

In terms of the achievement of coverage over time, we observe that for benchmarks where path-merging is not possible

[2]Same results were found between JR to JR+BTIG but omitted for space.

(Siena and Schedule), the difference in obligation coverage is insignificant (Fig. 7, and Fig. 8). The minor performance difference, is due to JR's static analysis. For benchmarks where path merging is beneficial, in terms of reducing the overall running time, such as in ApachiCLI and NanoXML in Tbl. II, we observe faster obligation coverage (in ApacheCLI, Fig. 10), with more obligation coverage collected (in NanoXML Fig. 9).

On the other hand, for benchmarks where path merging is not useful, i.e., it did not reduce the overall running time (such as in Replace and PrintToken in Tbl. II), we observe that in PrintTokens (Fig. 12), and in Replace (Fig. 11), JR covers new branches sometimes much slower, particularly in Replace.

Finally, we see mixed but also positive results on benchmarks where the entire code is merged (in the case of WBS and TCAS), making path merging similar to model checking. More precisely, we observe that while WBS shows no better obligation coverage (Fig. 13), TCAS shows a single additional coverage found by JR (Fig. 14). This suggests that while collapsing the entire code into a single constraint is expensive (Fig. III), there is always the benefit of comprehensively expressing the entire behavior of the code, allowing JR to cover hard obligations. The above results suggest that achieving additional coverage is more dependent on the performance of the baseline JR rather than on BTIG extension. In general, we conclude that our extension is able to collect more coverage when baseline JR is performing better than baseline SPF, and that, in general, the performance of path merging is dependent on the code structure of the benchmark it runs on.

C. RQ3: Uniqueness of SE & PM-SE Test Inputs

To evaluate whether SPF and JR are complementary in BTIG, consider Table. IV, which shows the common coverage (**common**) that both SPF and JR were able to reach, and the number of extra coverage elements achieved by SPF (**SPF extras**) and JR (**JR extras**). We can see that SPF and JR both can reach complementary obligations suggesting that both techniques, to some extent, have their own advantages in generating branch-adequate test inputs. It is important to observe that while the number of achieved coverage is important, reaching a few hard-to-find obligations is non-trivial. The latter is what we believe JR was able to achieve.

Finally, to investigate the number of test inputs generated by both techniques, we can see in Tbl. II that JR was able to produce fewer test inputs for the same amount of branch coverage for three programs (NanoXML, WBS, and TCAS), and particularly them 50% for WBS. On the other hand, SPF is able to produce a smaller number of test inputs for a single program (Replace). This result indicates that PM-SE is likely to generate fewer test inputs when used for branch coverage.

VI. Related Work

Symbolic execution has a wide range of applications including test input generation [10], [11], program and specification repair [12], [13], equivalence checking [14], [15], vulnerability finding [16], [17], invariant discovery [18], and protocol correctness checking [19]. In general, automatic testcase generation is a broad area of research most recently surveyed by Anand et al. [20].

TABLE IV: Complementary Coverage of SPF and JR

bench	common	SPF extras	JR extras
ApachiCLI	89	0	0
NanoXML	133	0	7
WBS	67	0	0
TCAS	83	0	1
Schedule	61	0	0
Siena	39	0	0
PrintTokens	185	4	0
Replace	99	36	1

At one end of a spectrum are approaches that completely explore an execution space; this includes symbolic execution [21], [22] and model checking [23]. In fact, automatic testcase generation has been a prime motivating application of symbolic execution since the early works of the 1970s [24]–[26]. Symbolic execution engines commonly explore one execution path at a time, so for instance, creating a test case from one satisfying assignment for each path builds a suite with path coverage. These techniques are ideal for relatively small programs whose execution tree can be explored entirely; however, for more computationally challenging programs, the number of execution paths can grow impractically large, so it is necessary to be more selective somehow.

Concolic execution [10], [27] and fuzzing [28], [29] tools typically give up on completeness, as it is in many cases impractical to enumerate all execution paths to generate relevant coverage goals. Concolic execution [10], [11] is a form of symbolic execution that generates a new concrete input from a previous one by finding a solution to a path condition prefix and a negated branch condition. One advantage of concolic execution is that it is easily mixed with fast random input mutations, for instance, to search for inputs that trigger crashes (fuzzing) [16], [30]. This can be seen as automatic test input generation with a simple automatic test oracle.

Path-merging symbolic execution [1], [31]–[33] is a complete approach that tries to alleviate the path explosion problem of symbolic execution by merging paths during exploration. In path merging, instead of symbolically executing all program execution paths, path merging collapses code regions by expressing their behavior using logical constraints. This allows path merging to explore fewer execution paths, often resulting in a performance improvement.

VII. Acknowledgment

This work was partially funded by an internship at JPF's Google Summer of Code 2020, as well as, the National Science Foundation under grant 1563920.

VIII. Conclusion and Future Work

This paper showed how coverage and test inputs can be generated for path-merged symbolic execution. We applied this technique to create test inputs for bytecode branch coverage. We implemented the technique as an extension to Java Ranger. Experiments showed mixed complementary results, indicating the importance of both techniques

In the future, we plan to investigate and create heuristics that use path merging when it can yield overall better performance.

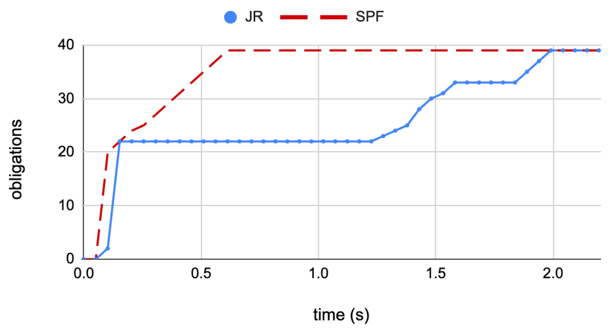

Fig. 7: Siena Obligation Coverage

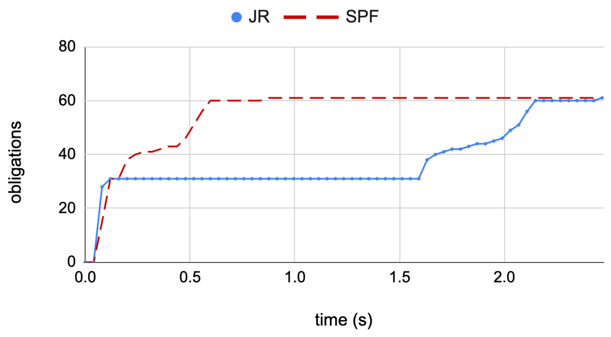

Fig. 8: Schedule Obligation Coverage

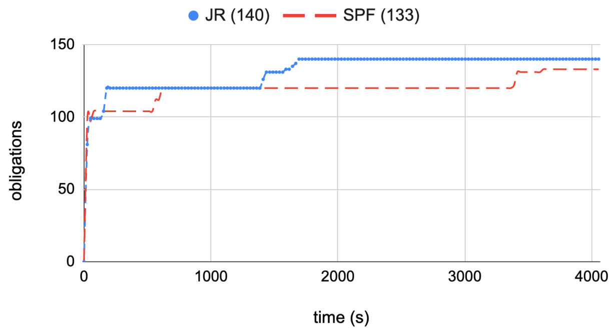

Fig. 9: NanoXML Obligation Coverage

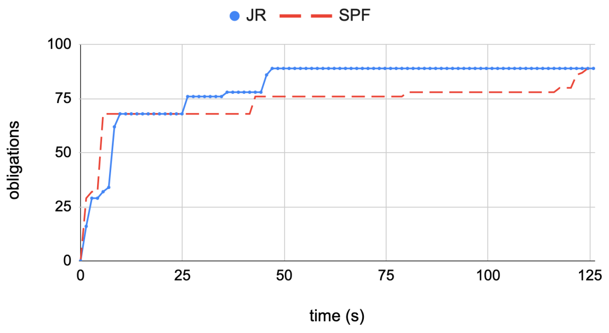

Fig. 10: ApacheCLI Obligation Coverage

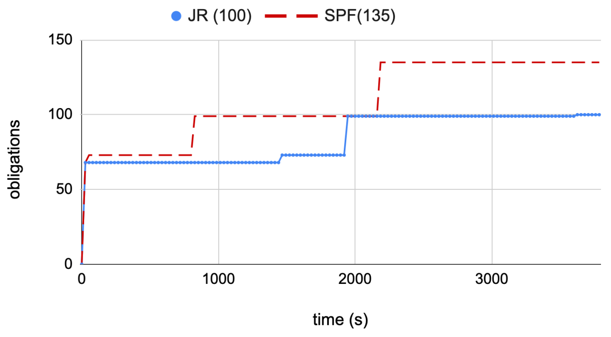

Fig. 11: Replace Obligation Coverage

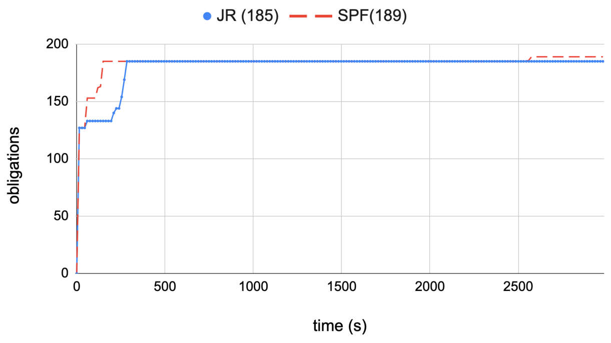

Fig. 12: PrintTokens Obligation Coverage

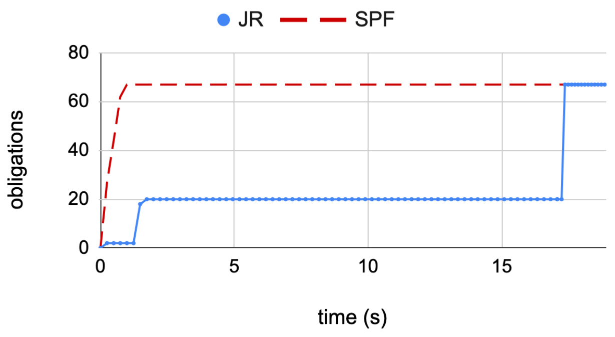

Fig. 13: WBS Obligation Coverage

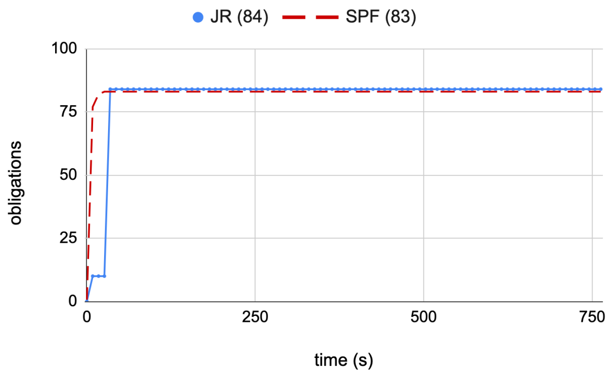

Fig. 14: TCAS Obligation Coverage

Fig. 15: Obligation coverage on all benchmarks

References

[1] V. Sharma, S. Hussein, M. W. Whalen, S. McCamant, and W. Visser, "Java ranger: Statically summarizing regions for efficient symbolic execution of java," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 123–134. [Online]. Available: https://doi.org/10.1145/3368089.3409734

[2] H. Kelly J., V. Dan S., C. John J., and R. Leanna K., "A practical tutorial on modified condition/decision coverage," Tech. Rep., 2001.

[3] P. Frankl and E. Weyuker, "An applicable family of data flow testing criteria," *IEEE Transactions on Software Engineering*, vol. 14, no. 10, pp. 1483–1498, 1988.

[4] C. S. Păsăreanu, W. Visser, D. Bushnell, J. Geldenhuys, P. Mehlitz, and N. Rungta, "Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis," *Automated Software Engineering*, vol. 20, no. 3, pp. 391–425, Sep 2013. [Online]. Available: https://doi.org/10.1007/s10515-013-0122-2

[5] S. Hussein, Q. Yan, S. McCamant, V. Sharma, and M. Whalen, "JAVA RANGER: Supporting string and array operations (competition contribution)," in *Proc. TACAS (2)*, ser. LNCS 13994. Springer, 2023.

[6] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. USA: Addison-Wesley Longman Publishing Co., Inc., 1986.

[7] P. Tu and D. Padua, "Efficient building and placing of gating functions," in *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, ser. PLDI '95. New York, NY, USA: Association for Computing Machinery, 1995, p. 47–55. [Online]. Available: https://doi.org/10.1145/207110.207115

[8] "Java Ranger GitHub Repository," https://github.com/vaibhavbsharma/java-ranger.git, 2023.

[9] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.

[10] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 213–223. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065036

[11] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for c," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-13. New York, NY, USA: ACM, 2005, pp. 263–272. [Online]. Available: http://doi.acm.org/10.1145/1081706.1081750

[12] T. Nguyen, M. B. Dwyer, and W. Visser, "Symlnfer: Inferring program invariants using symbolic states," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Oct 2017, pp. 804–814.

[13] S. Hussein, S. Rayadurgam, S. McCamant, V. Sharma, and M. Heimdahl, "Counterexample-guided inductive repair of reactive contracts," in *Proceedings of the IEEE/ACM 10th International Conference on Formal Methods in Software Engineering*, ser. FormaliSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 46–57. [Online]. Available: https://doi.org/10.1145/3524482.3527650

[14] D. A. Ramos and D. R. Engler, "Practical, low-effort equivalence verification of real code," in *Proceedings of the 23rd International Conference on Computer Aided Verification*, ser. CAV'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 669–685. [Online]. Available: http://dl.acm.org/citation.cfm?id=2032305.2032360

[15] V. Sharma, K. Hietala, and S. McCamant, "Finding substitutable binary code for reverse engineering by synthesizing adapters," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. Los Alamitos, CA, USA: IEEE Computer Society, apr 2018, pp. 150–160. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICST.2018.00024

[16] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting Fuzzing Through Selective Symbolic Execution," in *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. San Diego, CA: The Internet Society, February 2016, pp. 1–16.

[17] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Krügel, and G. Vigna, "SOK: (state of) the art of war: Offensive techniques in binary analysis," in *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*. IEEE Computer Society, 2016, pp. 138–157. [Online]. Available: https://doi.org/10.1109/SP.2016.17

[18] L. Zhang, G. Yang, N. Rungta, S. Person, and S. Khurshid, "Feedback-driven dynamic invariant discovery," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 362–372. [Online]. Available: https://doi.org/10.1145/2610384.2610389

[19] W. Sun, L. Xu, and S. Elbaum, "Improving the cost-effectiveness of symbolic testing techniques for transport protocol implementations under packet dynamics," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2017. New York, NY, USA: ACM, 2017, pp. 79–89. [Online]. Available: http://doi.acm.org/10.1145/3092703.3092706

[20] S. Anand, E. K. B. andTsong Yueh Chen, J. A. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn, "An orchestrated survey of methodologies for automated software test case generation," *J. Syst. Softw.*, vol. 86, no. 8, pp. 1978–2001, 2013. [Online]. Available: https://doi.org/10.1016/j.jss.2013.02.061

[21] M. Staats and C. Păsăreanu, "Parallel symbolic execution for structural test generation," in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ser. ISSTA '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 183–194. [Online]. Available: https://doi.org/10.1145/1831708.1831732

[22] D. Trabish, A. Mattavelli, N. Rinetzky, and C. Cadar, "Chopped symbolic execution," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 350–360. [Online]. Available: https://doi.org/10.1145/3180155.3180251

[23] S. Rayadurgam and M. Heimdahl, "Coverage based test-case generation using model checkers," in *Proceedings. 8th Annual IEEE International Conference On the Engineering of Computer-Based Systems-ECBS 2001*. CA, USA: IEEE Computer Society, 2001, pp. 83–91.

[24] R. S. Boyer, B. Elspas, and K. N. Levitt, "SELECT - a formal system for testing and debugging programs by symbolic execution," in *Proceedings of the International Conference on Reliable Software 1975, Los Angeles, California, USA, April 21-23, 1975*, M. L. Shooman and R. T. Yeh, Eds. New York, NY, USA: ACM, 1975, pp. 234–245. [Online]. Available: https://doi.org/10.1145/800027.808445

[25] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976. [Online]. Available: http://doi.acm.org/10.1145/360248.360252

[26] L. A. Clarke, "A system to generate test data and symbolically execute programs," *IEEE Trans. Software Eng.*, vol. 2, no. 3, pp. 215–222, 1976. [Online]. Available: https://doi.org/10.1109/TSE.1976.233817

[27] K. Luckow, M. Dimjašević, D. Giannakopoulou, F. Howar, M. Isberner, T. Kahsai, Z. Rakamarić, and V. Raman, "Jdart: A dynamic symbolic analysis framework," in *Tools and Algorithms for the Construction and Analysis of Systems*, M. Chechik and J.-F. Raskin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 442–459.

[28] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Transactions on Software Engineering*, vol. 45, no. 5, pp. 489–506, 2019.

[29] R. Padhye, C. Lemieux, and K. Sen, *JQF: Coverage-Guided Property-Based Testing in Java*. New York, NY, USA: Association for Computing Machinery, 2019, p. 398–401. [Online]. Available: https://doi.org/10.1145/3293882.3339002

[30] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: A practical concolic execution engine tailored for hybrid fuzzing," in *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, W. Enck and A. P. Felt, Eds. Berkeley, CA, USA: USENIX Association, 2018, pp. 745–761. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/yun

[31] T. Hansen, P. Schachte, and H. Søndergaard, "State joining and splitting for the symbolic execution of binaries," in *Runtime Verification*, S. Bensalem and D. A. Peled, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 76–92.

[32] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: ACM, 2012, pp. 193–204.

[33] 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*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 1083–1094. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568293

View publication stats