ECE 653 Project Report (Choice 0)

Weiqiang Yu[1] and Yichu Li[1]

University of Waterloo, Waterloo N2L 3G1, Canada

Abstract. In this project, we selected choice 0 and implemented two important components for our existing symbolic execution engine. We focus on feature 3 and 5, by changing the exploration method from BFS to DFS and applying incremental solutions with Z3's scopes (push and pop techniques). This report shows the source code about how we implemented these features and how they can improve the efficiency and performance of the current Wlang symbolic execution engine. Also, we provided test cases to achieve 100% complete statement and branch coverage for the new functions that we added. By using these test cases, we can get to know that our functions run as we expected. Overall, this report shows the improvements that we made for the existed Wlang symbolic execution engine.

Keywords: Symbolic execution engine · DFS · Incremental solving.

1 Introduction

The development of programming languages and tools is important in dealing with the growing complexity of software systems in the field of software engineering. In this project, we enhanced Wlang symbolic execution engine to improve this engine's ability to analyze and validate wlang programs.

The symbolic execution is a crucial tool for software testing and validation, which can simulate the execution of a program and also traverse all the possible paths. This process can help to identify possible bugs and vulnerabilities. We have improved this symbolic engine in two ways. First, we expanded our original Breadth First Search (BFS) symbolic executiuon engine by adding Depth First Search (DFS) as an exploration strategy (feature 3) to the Wlang symbolic execution engine. Second, we implemented incremental solving using the Z3 solver's scopes (feature 5).

1.1 Depth Frist Search Exploration Method

In the area of symbolic execution[1][2][3], the selected strategy for path exploration plays an important role in determining the efficiency and effectiveness of the symbolic engine. DFS is an algorithm that probs as far as possible along each branch before backtracking. This method is particularly beneficial when space is a limited factor and we are trying to find a solution that is likely located very deep in the search space.

Compared with BFS[4], which we used before for the exploration, requires storing all nodes at the current depth before moving to the next. BFS can be infeasible in symbolic execution when the paths grow exponentially. In contrast, DFS is more memory-efficient as it requires storage for only a single path with its backtracked states.

DFS allows us to fully explore one path before moving to another. Given the nature of symbolic execution, paths can diverge significantly, DFS can help to understand the implication of certain code deeply.

In conclusion, using DFS as the exploration method for symbolic execution engine is motivated by its memory efficiency, depth exploration capabilities, and natural alignment with the symbolic execution engine. By using DFS, we are allowed to create a more efficient and effective tool for program analysis and verification for this symbolic execution engine.

1.2 Incremental Solving

In the symbolic execution, particularly for the wlang engine, the concept of incrementality plays a crucial role in enhancing the performance and efficiency of our execution engine. Incremental solving is a technique that is widely used in Z3 solvers[5]. It deploys a dynamic method to check the satisfiability of assertions. It begins with an initial set of assertions and progressively incorporates additional assertions into the solver.

The core advantage of using incremental solving is its ability to maintain and control a set of constraints. This approach can help to save computational resources and also avoid redundant recalculations.

Our implementation leverages the concept of scopes in Z3 solvers to manage the lifecycle of assertions. By using the push and pop operations, we create and revert local scopes within the solver. Assertions added within a push are automatically retracted on a corresponding pop, allowing us to efficiently explore different states of the program without losing the context of previous states.

In general, by incorporating incremental solving into our symbolic execution engine for wlang, particularly in regards to the utilization of scopes in push/pop operations within Z3, we can achieve a notable improvement. This feature not only improves performance by reducing redundant computations but also provides a more nuanced control over the solver's state, enabling more sophisticated exploration strategies in symbolic execution and providing a flexible mechanism to explore different combinations of assertions efficiently.

2 Depth First Search Strategy for Symbolic Execution Engine

In previous assignment, we designed a symbolic execution engine employing the BFS strategy. For this project, we've decided to design the symbolic execution engine using DFS.

2.1 Recursive Function: visit_Next

We've designed a new function called visit_Next which will help us to realize recursive features. The pesudo-code of *visit_next* looks like this:

Algorithm 1 visit_Next

```
1: function VISIT_NEXT(..., kwargs)
2:     idx ← kwargs["idx"] + 1
3:     level ← kwargs["level"]
4:     state ← kwargs["state"]
5:     statement_list ← kwargs["statement_list"]
6:     cont ← kwargs["cont"]
7:     if cont then
8:         if idx < len(statement_list) then
9:             kwargs["idx"] ← idx
10:            statement ← statement_list[idx]
11:            self.visit(statement, ..., kwargs)
12:        else
13:            if level > 0 then
14:                nkwargs ← kwargs["prev"]
15:                self.visit_Next(..., nkwargs)
16:            else
17:                _, new_state ← state.fork()
18:                self.states.append(new_state)
19:            end if
20:        end if
21:    end if
22: end function
```

There are several parameters to consider for this function:

- **idx**: The `idx` parameter indicates the current index of the statement list. It helps identify the next statement to be executed.
- **level**: The **level** refers to the nesting depth of statement lists. Whenever a statement list node ({}) is encountered, the level is incremented by 1. Upon completing the execution of the statement list, the level is then decremented by 1. For example.

```
havoc y;   level 0
x := 2;   level 0
if y > 0 then {    level 0
    z := 0;     level 1
    while x > 0 do {   level 1
        z := z + 1;     level 2
        x := x - 1      level 2
    }
```

```
} else {
    x := x - 1;     level 1
    y := y + 1     level 1
};
assert y >= 0   level 0
```

- **state**: The `state` parameter represents the current symbolic state of execution.
- **statement_list**: The `statement_list` parameter contains the statements in the statement list node. Note that whenever a new statement list node is encountered, this parameter is updated. Further details will be discussed in Section 2.2.
- **cont**: The `cont` variable, short for continuous, is useful for deciding whether to proceed to the next statement. It is particularly valuable in the context of a while statement, ensuring that the program does not exit the loop prematurely before meeting the specified conditions.

Next, let's briefly discuss the functionality of this function.

1. If the current `cont` variable is false, then do nothing.
2. Otherwise, if the current `idx` is less than the length of the statement list, update the parameters and recursively call the next statement.
3. If the current `idx` is equal to the length of the statement list, check whether the `level` parameter is greater than 0. If true, read parameters from the `prev` entry and continue exploring the remaining statements from the preceding statement list.
4. When reaching the end of the first statement list (level = 0), make a copy of the current state and append it to `self.states` for the final output.

To ensure the entire program is recursive, every other statement function should invoke the self.visit_Next method. For instance, Algorithm 2 represents the modified code for the assignment statement.

Algorithm 2 visit_AsgnStmt

function VISIT_ASGNSTMT($node, *args, **kwargs$)
    $state \leftarrow kwargs['state']$
    $state$.env[$node$.lhs.name] $\leftarrow$ self.visit($node$.rhs, $*args, **kwargs$)
    self.visit_Next($*args, **kwargs$)
end function

2.2 Statement List

We've updated the `visit_StmtList` function to achieve the following purposes.

1. If the statement list is empty, representing the beginning of the program, we update the `kwargs` parameter with predefined values (e.g., setting `idx` to -1 and `cont` to True). Introducing a new parameter called **loop** helps track executed loop counts, ensuring while statements don't exceed 10 iterations.
2. If the statement list is not empty, a dictionary called `nkwargs` is created and initialized with default parameters. The old parameters are then stored in `prev` entry to enable revisiting when the new statement list is finished. Additionally, we increment the `level` by 1, signifying that this statement list can further descend to continue the previous statement list.

Algorithm 3 visit_StmtList

```
function VISIT_STMTLIST(node, *args, **kwargs)
    statement_list ← kwargs["statement_list"]
    if len(statement_list) > 0 then
        nkwargs ← dict(kwargs)
        kwargs['prev'] ← nkwargs
        kwargs['level'] ← kwargs['level'] + 1
    else
        kwargs['level'] ← 0
    end if
    kwargs['idx'] ← −1
    kwargs["statement_list"] ← node.stmts
    kwargs['loop'] ← {}
    kwargs["cont"] ← True
    visit_Next(*args, **kwargs)
end function
```

2.3 If Statement

For an if statement, we first explore the path when the condition is satisfied (the "then" branch). Once that path is completed, we then visit the alternative path (the "else" branch or skip). Specifically,

- If the condition is satisfied, invoke `self.visit` on the "then" statement.
- If the condition is not satisfied, invoke `self.visit` on the "else" statement if one exists; otherwise, simply call `self.next_Visit`.

2.4 While Statement

Regarding the while statement, we need to address two scenarios: one where an invariant (inv) is provided, and the other where it is not. When a loop invariant is provided, we essentially follow the code structure outlined below:

Algorithm 4 visit_IfStmt

```
1: function VISIT_IFSTMT(node, *args, **kwargs)
2:     state ← kwargs["state"]
3:     cond ← self.visit(node.cond, *args, **kwargs)
4:     state.push()
5:     state.add_pc(cond)
6:     if not state.is_empty() then
7:         self.visit(node.then_stmt, *args, **kwargs)
8:     end if
9:     state.pop()
10:    state.add_pc(z3.Not(cond))
11:    if not state.is_empty() then
12:        if node.has_else() then
13:            self.visit(node.else_stmt, *args, **kwargs)
14:        else
15:            self.visit_Next(*args, **kwargs)
16:        end if
17:    end if
18: end function
```

```
assert inv;
havoc V;
assume inv;
if b then { s ; assert inv; assume false }
```

Algorithm 5 presents the code for `visit_WhileInv`. In line 22, we set `cont` to false, ensuring only the body statement is executed. Detailed considerations are as follows:

1. If the body statement is not a statement list (e.g., an assignment statement), we call `visit_Next` after executing this statement. With `cont` set to false, exploration stops.
2. If `node.body` is a statement list node, we record the current configuration in `prev` entry. We then update the statement list, `idx`, cont etc. When exploring the new statement list, `cont` is true, allowing execution of multiple statements. After finishing the statement list, we return to the previous statement list, reload the settings, and execute `visit_Next` again. As `cont` is still false, further exploration halts.

Following the execution of the body statement, we revert `cont` back to True for further exploration.

Algorithm 5 visit_WhileInv

```
1: function VISIT_WHILEINV(node, *args, **kwargs)
2:     inv1 ← self.visit(node.inv, *args, **kwargs)
3:     state.push()
4:     state.add_pc(z3.Not(inv1))
5:     if not state.is_empty() then
6:         print("inv fails initiation")
7:     end if
8:     state.pop()
9:     state.add_pc(inv1)
10:    if not state.is_empty() then
11:        self.uv.check(node.body)
12:        vars ← self.uv.get_defs()
13:        for v in vars do
14:            state.env[v.name] ← z3.FreshInt(v.name)
15:        end for
16:        inv2 ← self.visit(node.inv, *args, **kwargs)
17:        state.add_pc(inv2)
18:        cond ← self.visit(node.cond, *args, **kwargs)
19:        state.push()
20:        state.add_pc(cond)
21:        if not state.is_empty() then
22:            kwargs['cont'] ← False
23:            self.visit(node.body, *args, **kwargs)
24:            kwargs['cont'] ← True
25:            inv3 ← self.visit(node.inv, *args, **kwargs)
26:            state.push()
27:            state.add_pc(z3.Not(inv3))
28:            if not state.is_empty() then
29:                print ("inv fails initiation")
30:            end if
31:            state.pop()
32:        end if
33:        state.pop()
34:        state.add_pc(z3.Not(cond))
35:        if not state.is_empty() then
36:            self.visit_Next(*args, **kwargs)
37:        end if
38:    end if
39: end function
```

Additionally, we must address scenarios where a loop invariant is not provided. In such cases, we restrict the while statement to execute no more than 10 times. Algorithm 6 elucidates the overarching concept.

– The key value is defined as $key = \text{loop} + \text{idx}$. Including idx is crucial because one statement list may consist of multiple while loops, and idx helps distinguish them. Upon encountering a new statement list node, we store the

current loop value inside the prev entry and create a new empty dictionary to track loop values for the new statement list. This approach prevents conflicts when different loops in two distinct statement lists share the same key. Moreover, if the loop executes more than 10 times or exits prematurely, it is essential to reset the corresponding key to 0. This is crucial because the while loop might be executed again in another path.

– To ensure that the program does not proceed to the next statement before completing the while loop, a slight adjustment is made. We set idx to idx - 1(line 18). This way, when the body statement concludes and visit_Next is called, the same while statement is invoked again. With the loop parameter controlling finite execution, our while statement is both sound and complete.

Algorithm 6 visit_While

```
1: function VISIT_WHILE(node, *args, **kwargs)
2:     key ← "loop-" + kwargs["idx"]
3:     if key ∉ kwargs["loop''] then
4:         kwargs["loop''][key] ← 0
5:     end if
6:     loop ← kwargs["loop''][key]
7:     cond ← self.visit(node.cond, *args, **kwargs)
8:     state.push()
9:     state.add_pc(z3.Not(cond))
10:    if not state.is_empty() then
11:        self.visit_Next(*args, **kwargs)
12:    end if
13:    state.pop()
14:    state.add_pc(cond)
15:    if loop < 10 then
16:        kwargs["loop''][key] ← kwargs["loop''][key] + 1
17:        if not state.is_empty() then
18:            kwargs["idx"] ← kwargs["idx"] − 1
19:            self.visit(node.body, *args, **kwargs)
20:        else
21:            kwargs["loop''][key] ← 0
22:        end if
23:    else
24:        kwargs["loop''][key] ← 0
25:    end if
26: end function
```

3 Incrementality

3.1 Scopes

For our project, we employ the scope strategy across multiple statements. A representative use of scope is seen in the if statement. In line 4 of Algorithm 4, we

push the current state and append *cond* to the path condition. Subsequently, we explore the path under this modified condition. Once this path is completed, we pop the state, add $z3.Not(cond)$ to the path condition, and continue exploration. Another instance is observed in the assert statement.

Algorithm 7 Visit AssertStmt

```
1: function VISIT_ASSERTSTMT(node, *args, **kwargs)
2:     state ← kwargs["state"]
3:     cond ← self.visit(node.cond, *args, **kwargs)
4:     state.push()
5:     state.add_pc(z3.Not(cond))
6:     if not state.is_empty() then
7:         state.mk_error()
8:         state.is_error()
9:         print("Assertionmightbeviolated")
10:    end if
11:    state.pop()
12:    state.add_pc(cond)
13:    if not state.is_empty() then
14:        self.visit_Next(*args, **kwargs)
15:    end if
16: end function
```

In line 4 of Algorithm 7, we examine the path where $\neg cond$ holds. Subsequently, in line 11, we pop the state, incorporate *cond* into the path condition, and continue exploring the remaining path.
Similar application of the scope feature is evident in the while statement.

4 Obtained Results

4.1 Feature 3: Depth First Search Exploration Strategy

We've tested the exploration paths of the following while program under both BFS and DFS settings.

```
havoc x, y;
if (x > 0 and y < 0) then x := x / 2 else y := y * 2;
assume y > x
```

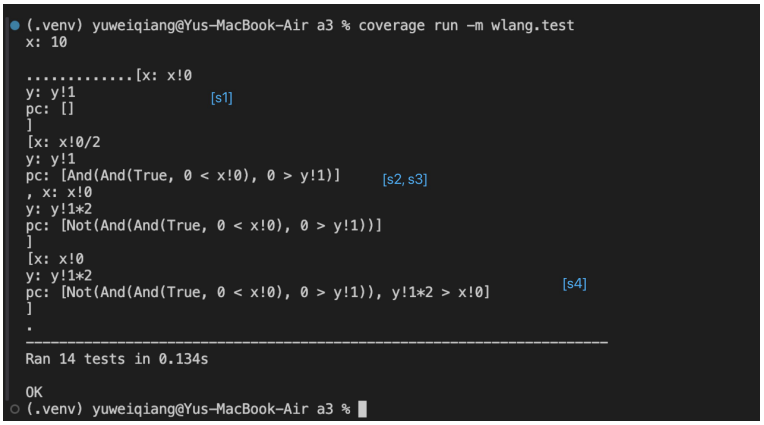

Fig. 1. BFS Sample Result

In BFS, we initially executed the havoc statement, obtaining the state [s1]. Subsequently, upon executing the if statement and considering both the then and else branches, we obtained states [s2, s3]. For each state, we then executed the assume statement. As only s4 satisfied the condition, our algorithm returned [s4] as the answer.

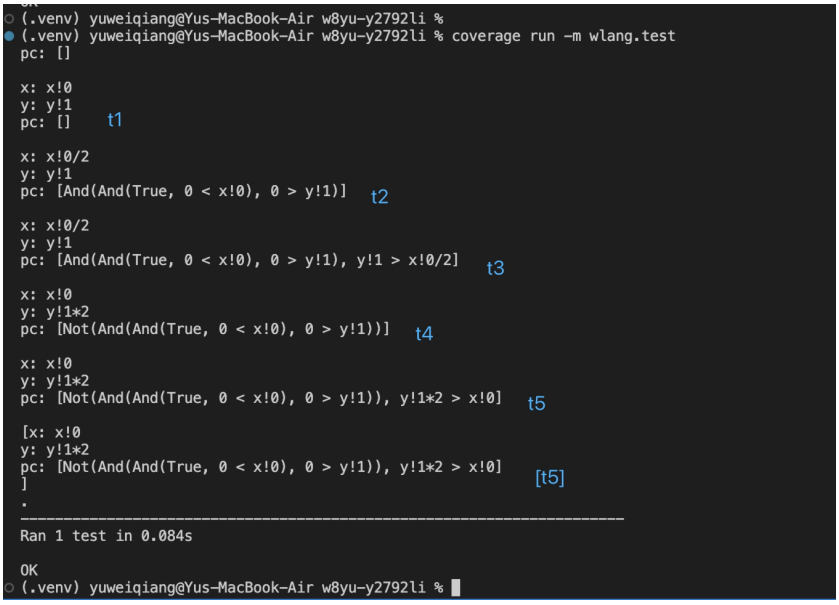

Fig. 2. DFS Sample Result

For DFS, we initiated the exploration by executing the havoc statement, yielding state t1. Subsequently, we traversed the path in the then branch of the if statement, resulting in state t2. Attempting to execute the assume statement, we found that t3 did not yield a solution. We then backtracked to the if statement, explored the path in the else branch, and obtained state t4. Executing the assume statement in this context, we acquired a valid solution, t5. We added t5 to the final states, and since there were no more paths to visit, we returned [t5] as the final result.
Figures 1 & 2 presents example outputs of the same program using the BFS and DFS approaches.

4.2 Feature 5: Incrementality (Scope)

We wrote two versions of the symbolic execution engine: one without incrementality, and the other employing scope techniques (push and pop). Then we conducted some experiments to compare the execution time of the symbolic execution engine with and without using scopes by running on ten different test cases that we created. The experimental results are presented in the table below. From the data, it's clear that using scopes generally provides a performance benefit across a variety of test cases. The most significant improvements are seen in tests involving loops and conditional branches, which are common in symbolic execution and can produce a large number of paths to explore.

Scopes enable the symbolic execution engine to run the incremental constraints more efficiently. By using scopes techniques, symbolic execution engine can focus on the current context without worrying about irrelevant computations and constraints. This approach significantly aids in reducing recalculations and improving the overall efficiency of the execution.

Table 1. Analyzing Execution Time: Impact of Enabling Scope Incrementality Feature

Test case	No Incrementality(s)	Scope(s)
test_one	0.202s	0.199s
test_aexp	0.151s	0.145s
test_while_ten_loops	0.369s	0.188s
test_while_zero_loop	0.167s	0.128s
test_rel_exp	0.259s	0.191s
test_bool_exp	0.226s	0.160s
test_if_not_into_else	0.218s	0.172s
test_while_must_into_loop	0.271s	0.201s
test_nested_if	0.263s	0.200s
test_while_more_than_10	0.308s	0.298s

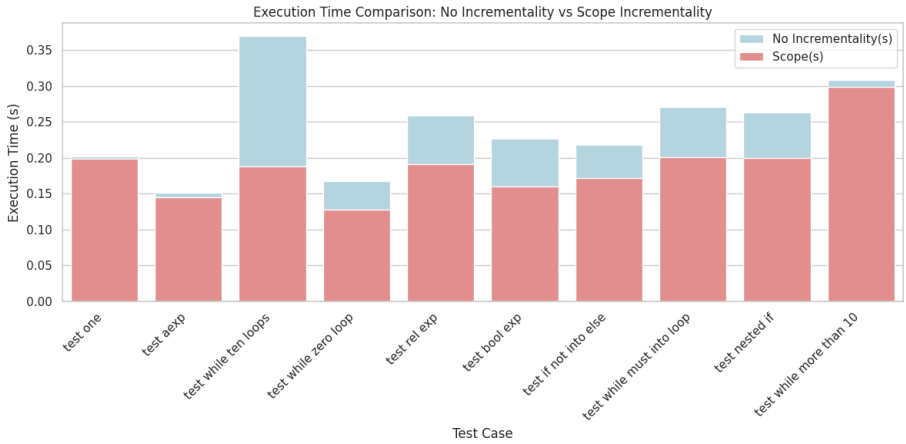

Fig. 3. Execution Time Comparison: No Incrementality vs Scope Incrementality

4.3 Testing: Code Coverage

The integrity and reliability of new code are crucial, especially in the context of symbolic execution engines where accuracy is critical. Because of this, our testing approach is comprehensive, aimed at ensuring that every new line of code and each branch of logic introduced for the wlang project was thoroughly tested[6]. In the testing part, we created over twenty test cases to achieve 100% statement and branch coverage[7] for all the new code that we wrote for this project. The process of testing was not only about verification but also about validation. It was important to confirm that the newly implemented features functioned as expected and can enhance the symbolic execution engine. Therefore, our tests were designed to reflect realistic test cases, ensuring that the features would stand up to the demands of real-world application. All the test cases can be found in test_sym.py. We also computed the coverage of our test cases and generated a HTML report to show that we achieved complete statement and branch coverage for the newly added features. We will provide some parts of the generated report to illustrate the completion of statement and branch coverage.

We revised visit_StmtList function to achieve the DFS exploration for the symbolic execution engine. The figure below shows the coverage report for our visit_StmtList function. It shows that the updated function has achieved complete statement and branch coverage.

```
360     def visit_StmtList(self, node, *args, **kwargs):
361         statement_list = kwargs["statement_list"]
362         if len(statement_list) > 0:
363             nkwargs = dict(kwargs)
364             kwargs['prev'] = nkwargs
365             kwargs['level'] += 1
366         else:
367             kwargs['level'] = 0
368 
369         kwargs['idx'] = -1
370         kwargs["statement_list"] = node.stmts
371         kwargs['loop'] = {}
372         kwargs["cont"] = True
373         self.visit_Next(*args, **kwargs)
```

Fig. 4. Test Coverage Result of visit_StmtList

The following image showcases the coverage report for our visit_Next function. visit_Next is a newly-developed function that we wrote for leveraging DFS exploration method for Wlang symbolic engine. This part of report also expresses that this visit_Next function can achieve 100% coverage for both statement and branch, underlining its comprehensive effectiveness in the context of our testing strategy.

```
125     def visit_Next(self, *args, **kwargs):
126         idx = kwargs["idx"] + 1
127         level = kwargs["level"]
128         state = kwargs["state"]
129         # print(state, idx, level)
130         # print(state._solver.assertions(), idx, level)
131         statement_list = kwargs["statement_list"]
132         cont = kwargs["cont"]
133         if cont:
134             if idx < len(statement_list):
135                 kwargs["idx"] = idx
136                 statement = statement_list[idx]
137                 self.visit(statement, *args, **kwargs)
138             else:
139                 if level > 0:
140                     nkwargs = kwargs["prev"]
141                     self.visit_Next(*args, **nkwargs)
142                 else: # if level == 0 and not state.is_empty():
143                     _, new_state = state.fork()
144                     self.states.append(new_state)
```

Fig. 5. Test Coverage Result of visit_Next

References

1. James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, jul 1976.

2. Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques, 2018.
3. Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Pasareanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic execution for software testing in practice: preliminary assessment. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 1066–1071, 2011.
4. Konrad Zuse. *Der Plankalkül*. 1972. Konrad Zuse Internet Archive. See pp. 96–105 of the linked pdf file (internal numbering 2.47–2.56).
5. D. R. Cok. The smt-libv2 language and tools: A tutorial. *Language C*, pages 2010–2011, 2011.
6. Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2016.
7. Matt Staats, Gregory Gay, Michael Whalen, and et al. On the danger of coverage directed test case generation. pages 409–424. Springer Berlin Heidelberg, 2012.