A review on automatic test generation through symbolic execution

Zhen Gong, Shaowei Sun, Aiyuan Liu

University of Waterloo

Abstract. Though many algorithms have been proposed in software automatic testing field, symbolic execution plays an important role in it, with contributions to debugging and automated program analysis. Different from other testing methods, symbolic execution tests programs with abstract symbols instead of concrete values. It is one of static test methods, which is also commonly understood as symbols prediction. At the same time, generated based on symbolic execution method, dynamic symbolic execution adds concrete execution to the symbolic execution to implement dynamic execution, which is also called concolic execution. In this article we are going to present the methods of both symbolic execution and dynamic symbolic execution. And in order to better understand the study area, in this article we are going to introduce two state-of-the-art test generation tools based on symbolic execution.

1 Introduction

As an essential formal method and software analysis technology, symbolic execution replaces program variables with abstract symbols. In this case, a single abstract execution offers multiple possible inputs of the program which share a common path through the code. The execution treats the input in symbolic and provides a result expressed in symbolic way representing these input values. Thus, with the problem of detecting and identifying vulnerabilities in software systems, symbolic execution offers a solution of exploring many possible execution paths systematically and drops necessarily requiring concrete inputs.

Developed based on the methods of symbolic execution, dynamic symbolic execution is a well-known technique for automatically generating tests in order to achieve high level of coverage of testing. Dynamic symbolic execution contains both concrete values and symbolic values. With regard to these two mentioned testing approaches, we are going to look more into them in the following sections of this article.

When it comes to verify the correctness and quality of a given system, test suites are generated to handle this problem. However, generating test suites still poses an obstacle whether the tests are able to sufficiently cover requirements of the system. Thus, mutant testing, as a form of white-box testing, is created in order to implement new software tests and guarantee the quality of generated software tests. Mutant testing contains process of modifying a software program

in small ways. Mutant is created by mutating the existing program and more tests are defined to detect and reject mutants by causing different behaviors from original version. The main purpose of applying mutant testing is to help testers define effective tests and locate weaknesses within the test data used for the program.

The remainder of this article is structured as follows: Section 2 presents the traditional symbolic execution. Section 3 describes the methods of dynamic symbolic execution. Section 4 involves the details of mutant testing. Section 5 presents two test generation tools via symbolic execution in detail. Finally, section 6 concludes the work we have done in this article.

2 Traditional Symbolic Execution

The core idea of classical symbolic execution is to use symbolic values instead of concrete values as program input and use symbolic expressions to represent the values of program variables related to symbolic values. When the program branch instruction is encountered like if statement, the program's execution should also searches each branch separately. The branch condition is added to π of the program state saved by the symbol execution; π represents the constraint condition of the current path. After collecting the path constraints, the program will use constraint solver to verify the solvability of the constraints to determine whether the path is reachable. If the path constraint is solvable, it means that the path is reachable; Otherwise, it means that the path is unreachable, and the path analysis is ended.

Take the example code in Figure 1 as an example to illustrate the principle of symbol execution. There is an error in line 9 of the program. Our goal is to find a suitable test case to trigger the error. If the method of randomly generating test cases is used to test the program, there are 2^{32} values of integer input variables x, y and z respectively. If the values of x, y and z are randomly generated as the input of the program test, the possibility of triggering program errors is slight. Symbolic execution can solve this problem effectively. The processing of symbol execution is to use the symbol value instead of the concrete value. In the process of symbol execution, the symbol execution engine always maintains a state of information, which is expressed as (PC, π, σ), of which:

1. PC points to the next program statement to be processed, which can be an assignment statement, conditional branch statement or jump statement.
2. π refers to the path constraint information, which is expressed as the conditional branches that need to be passed by to execute the concrete statement of the program, And the expression of the symbolic value α_i at each branch. In the analysis process, it is initially defined as $\pi = true$.
3. σ Represents the set of symbolic values related to program variables, including concrete and symbolic values of α_i.

```
1. foo(int x,int y,int z){
2.    a=0,b=0,c=0
3.    if(x>0){a=-2;}
4.    if(y<5){
5.       if(y+z>0){b=1;}
6.       c=2;
7.    }
8.    if(a+b+c==3)
9.          //some error
10.    exit()
11. }
```

Figure 1 example code of symbol execution

The symbol execution algorithm is shown in algorithm 1. First, x, y, z is the input of the program, while the program defines the values of x, y, z as symbolic variables $\sigma : \alpha, \beta, \gamma$, and since no conditional branch has been executed, the initial state is $\sigma : \{x \to \alpha, y \to \beta, z \to \gamma\}$; $\pi = true$. It should be noted here that in addition to x, y, and z initially defined as symbolic variables, if there is an assignment operation related to x, y, and z during the code operation, the newly generated variables must also be treated as symbolic variables. After that, the symbolic engine will modify the path condition and the symbolic values based on the different operations of the program. In principle, traditional symbolic execution can fully cover program paths, and test cases that conform to each path can be generated. The program execution tree of the example code is shown in Figure 2. There are three branch judgment points (if statement) in the program, with a total of 6 paths. That is, the symbol execution engine needs to perform six constraint solutions and get test cases for six paths. Among them, for the test case obtained by solving the constraint $x \leq 0 \wedge y < 5 \wedge y + z > 0$, the execution result is a = 0, b = 1, b = 2, which will trigger the program error.
Traditionally, in software testing, symbolic execution is used to explore different program paths as many as possible, generate corresponding concrete input data according to each path condition, and then use these data as input to execute the program to check whether there are software errors, which contains aspects of assertion violations, security vulnerabilities, uncapped exceptions and memory corruption. Being able to generate concrete input data is also an essential advantage of symbol execution: from the perspective of test case generation, it can construct a high coverage set of test cases, and from the perspective of finding errors, it can provide developers with concrete input instances that can trigger

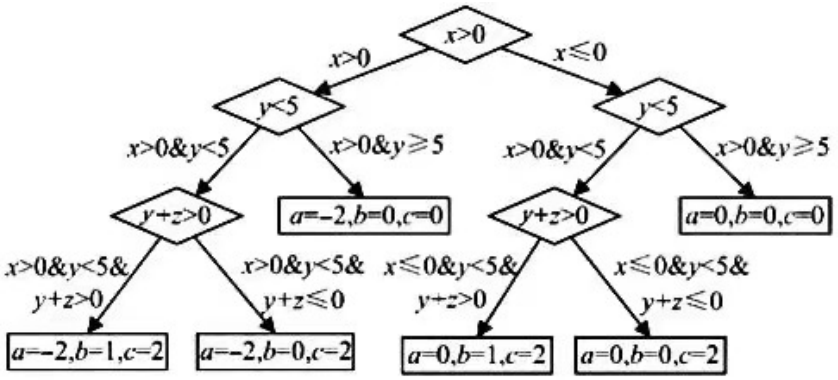

Figure 2 Program Execution Tree of Sample Code

errors. The ability of symbol execution to detect program errors is better than traditional dynamic execution technologies. Because if there is an error on a path, the symbol execution can automatically generate the input data that triggers the error to find it. The traditional dynamic execution technology must provide input data that can trigger errors in advance. Symbolic execution can find more program errors than other program analysis techniques. Symbol execution also can find general errors such as buffer overflow and memory leak and check higher-level program attributes such as complex program assertions.

The main disadvantages of symbol execution are that it is difficult to deal with complex dynamic data structures and to solve path conditions containing complex constraints. For example, C language programs usually include pointers, and the pointer's value is generally determined at runtime. The position of the pointer may not be determined during symbol execution, so symbol execution is difficult to deal with complex dynamic data structures. Because current SMT solvers cannot solve nonlinear constraints efficiently, they cannot efficiently solve path conditions with nonlinear constraints. The path conditions containing transcendental functions are even more powerless. Besides that, how to deal with path explosion and symbol execution of parallel programs also need to be considered.

3 Dynamic Symbolic Execution

3.1 Dynamic symbol execution logic

Dynamic symbolic execution extends the traditional symbolic execution. It adds concrete execution to the symbolic execution to utilize the dynamic execution information. Dynamic symbolic execution is also called concolic execution, in which the word concolic combines two words: concrete and symbolic. It overcomes the shortcomings of traditional symbolic execution that it is difficult to

deal with dynamic data structures and cannot solve path conditions containing complex constraints.

In dynamic symbolic execution, the concrete values (data and address) obtained by program execution are used to replace the symbolic variables related to the dynamic data structure in symbolic execution and simplify the difficult arithmetic operations. Each dynamic symbol execution instance maintains not only a symbol state but also a concrete state. Similar to the symbolic state, the concrete state stores the mapping of all variables to their concrete values; However, unlike symbolic execution which sets the value of variables in the initial state to null, concolic execution usually sets the value of variables in the initial state to concrete values (these values are the input data for the concrete execution of the program) to start concrete execution.

The algorithm process of dynamic symbol execution is as follows:

1. First, insert the functions that can perform symbolic execution in the tested program.
2. Judge whether the conditions for dynamic symbol execution are met. If yes, proceed to step 3; otherwise, end the whole dynamic symbol execution process.
3. If there is no new concrete input data, the input data will be randomly generated; otherwise, the new concrete input data will be provided to the program for concrete execution directly.
4. During the concrete execution of the program, the inserted function simulates the execution process of the program and records the current execution path conditions. If the symbolic variable involves nonlinear operations, dynamic data structures, etc., the symbolic variable is replaced with a concrete value.
5. When reaching the end position of the program under test, reverse a branch condition in the path condition to get a new path condition. If the unique path condition constraint can be solved, generate a new test case and turn to step 2.

The termination condition of the above algorithm is that the generated test cases meet some code coverage criteria or the number of iterations reaches a preset value. Take the example code given in Figure 1 to explain the testing process of concolic testing. Figure 3 shows the path constraint tree of the program. It can be seen that the program has six different paths, and each path has its corresponding constraint set.
The path constraint tree shows that the code has five paths to the end of normal execution and one path that may lead to error. The trajectory of program execution is determined through the constraint set, and the program is guided to execute along the set path. For the sample program, the set of constraints that can trigger program errors is $(x \leq 0) \wedge (y < 5) \wedge (y + z > 0)$. During the dynamic

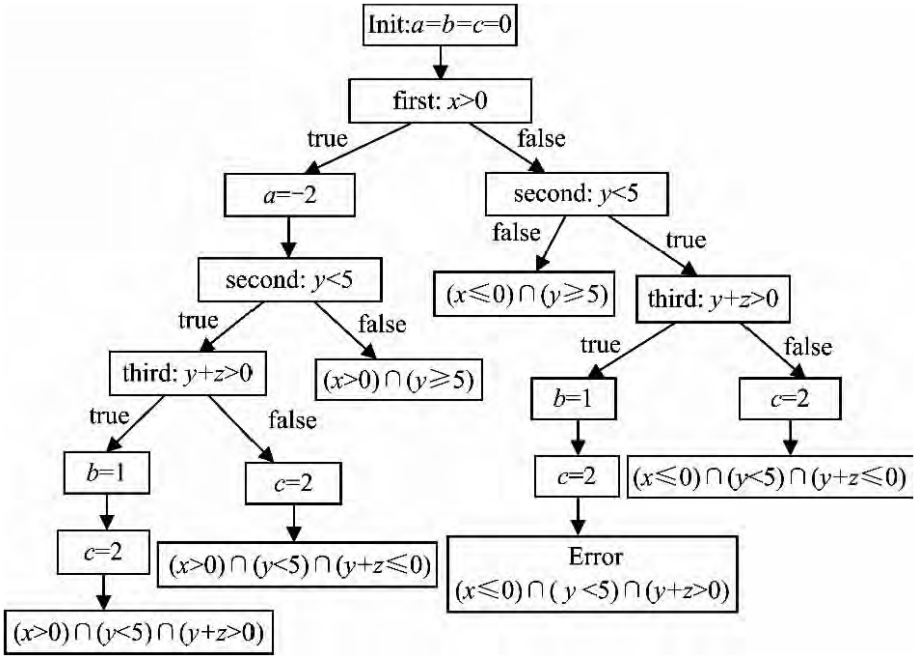

Figure 3 Path constraint tree of the sample code

symbolic execution process, constraints on the execution path are collected and saved. To fully cover the code path, the program will be executed six times, and each time, a constraint condition is selected to be negated, and then the new test case is solved to test another path.

Dynamic symbolic execution replaces some symbolic variables with concrete values (for example, variable values cannot be solved by constraints, its variable values are related to external function libraries, etc.), which improves the ability of symbolic execution to deal with various complex situations. Replacing symbolic variables is usually effective because, in the worst case, this replacement is equivalent to the random assignment of symbolic variables.

3.2 Dynamic Symbolic Execution Advantages

Before people realized the advantages of DSE, many security tools mainly used static and black box fuzzing technology. However, DSE is much more accurate than typical static analysis and has advantage of better code coverage over black box technology. Thus, it has become one of the most popular research topics. And here are the main reasons:

Static test generation is often ineffective
Static techniques can be ineffective when it comes to external or complex arith-

metic functions contained in another function. In this scenario, constraint solvers might fail to analyze the program symbolically. The following example shown in Figure 4 describes this problem: assume that external function is a hash function or a complex arithmetic function. In this case, symbolic execution is unable to be performed on the function. Therefore, it is unlikely to generate an input value which guarantees to hit the error in statement 3. Unfortunately, most programs contain complex program statements (such as pointer operation, arithmetic calculation and etc.), calls to the operating system and encapsulated library functions. Such problems are common, so static test generation cannot detect program errors in most cases.

```
1: void example2(int x, int y)
2: {
3:   if (x == hash(y)) abort(); // error()!
4: }
```

Figure 4 An example to show the limitation of static test generation

In this case, DSE is more effective than static test generation. With the help of the runtime information of the program, the problems caused by external/complex arithmetic functions can be solved. Consider the above function again. In the first round of execution, we usually cannot hit the program branch, which may lead the program to fail. Still, we can dynamically collect the calculation of hash (y) in this round of execution. Therefore, in the next round of program execution, we can fix the previous y to give the program a new input x, equal to hash (y), so that the program can execute to the then branch.

Black-box fuzzing always has low code coverage
Black box fuzzing can automatically generate random input values and find code errors in many applications. In addition, an important reason for its attraction is that it requires little human intervention. However, black box fuzzing has a serious disadvantage: it is blind to the internal execution of the program, and it will miss the errors triggered by the input in concrete cells. Therefore, the coding coverage of black box fuzzing is bound to be low.

However, DSE only needs to generate up to eight purposeful test cases (eight are the upper limit, and the actual number of test cases required to reveal errors depends on the search algorithm) [4] to ensure that program errors can be hit.

4 Mutant Testing

Generally, software testers use control flow or data flow analysis to define test adequacy criteria and guide the design of corresponding test cases. Mutation

testing measures test adequacy from another perspective. This testing technology can be used to access and improve the adequacy of the test suite.

Mutant testing simulates the actual defects in the software by artificially injecting defects into the original program. The copy of the program injected with defects is called a mutant. Using the same test input, execute the original program and the mutant. If a test case can distinguish a mutant from the original program from the execution result, the mutant is said to be killed; If no test case can kill a variant, the mutant is called an equivalent mutant.

Testers initially design mutation operators according to the features of the tested program. Mutation operators generally make minor changes to the given program in order to conform to the syntax. Then the mutation operator is applied to the tested program to generate many mutants. After identifying the equivalent mutants, if the existing test cases cannot kill all non-equivalent variants, it is necessary to design new test cases and add them to the test case set to improve the adequacy of testing. In addition to the adequate evaluation of the test case set, variation testing is able to simulate the real defects of the tested software by using variation defects to assist in evaluating the effectiveness of the test methods proposed by researchers. The above variation test analysis is sometimes referred to as variation analysis.

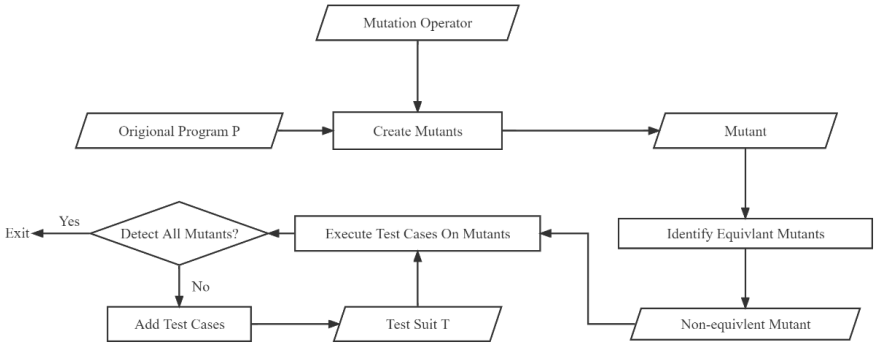

Figure 5 Traditional process of mutation testing analysis

The traditional process of mutation testing analysis is shown in Figure.5. Given the tested program P and the test case suit T, a series of mutation operators are set according to the characteristics of the tested program. Then a large number of mutants are generated by executing the mutation operator on the original program P. After that, the equivalent mutants will be identified from a large number of variants. Finally, execute the test cases in the test case set T on the remaining non-equivalent mutants. The mutation test analysis is completed if

all non-equivalent mutants can be detected. Otherwise, new test cases should be designed and added to the test case set T for the undetected mutants.

In mutant testing, there is an essential notion: *mutant killing*. Mutant testing is based on a series of test inputs. If the mutant program has a different execution result from the original program with the same input, then it is denoted as killed mutants. If the mutant doesn't change the program's overall semantics or cannot be killed by any input, then we will call these mutants *equivalent mutants*.

The quality of the test input is reflected by the number of mutants it kills: the more mutants it kills, the more effective it becomes. The test should meet three criteria to kill the mutant: *Reachability*, *Necessity* and *Sufficiency*. In reference [1], the author detailed discussed the three different dimensions. In the following part, P is defined as the program, T is defined as the test suit and a mutant of P on statement S is defined as M.

Reachability: Mutant M is the same with program P except the mutated statement S.

Necessity: For a test input t ($t \in T$) to kill mutant M, t must cause different internal states on P and M immediately after executing S.

Sufficiency: For a test input t ($t \in T$) to kill mutant M, t must lead to different final states for M and P.

In reference [5], the author came out with the idea of *weak mutation testing*. If the test inputs satisfy the reachability and necessity criteria, they are denoted as satisfying the weak-mutation-testing criterion. The *strong mutation killing* and *weak mutation killing* are defined as:

Definition 4.1: A test input t *weakly kills* a mutant M, iff t satisfies the reachability and necessity criteria in mutation testing.[1]

Definition 4.2: A test input t *strongly kills* a mutant M, iff t satisfies the reachability, necessity, and sufficiency criteria in mutation testing.[1]

As it is difficult to generate strong mutation killing test cases, most of the technology nowadays are using the weak mutation testing concept and the reachability and necessity criteria to generate test inputs [1].

5 Test Generation Tools via Symbolic Execution

In this part, we will present two tools used to generate mutant test cases via dynamic symbolic execution.

5.1 PexMutator

PexMutator is an approach for test-generation proposed by L. Zhang et al., which

is designed to generate mutant tests via dynamic symbolic execution. It is an extension of a testing tool called Pex, developed by Microsoft Research.

Initially, the testing program will be converted to a meta-program with mutant-killing constraints based on a series of transformation principles. Then PexMutator will perform dynamic symbolic execution on the meta-program to generate useful test inputs. During the symbolic execution process, mutant-killing constraints are collected and test inputs are generated automatically to kill the mutants.

The test generation process of PexMutator contains the following four steps:

(a) Generate mutants by converting the original program program to transformed mutated program.

(b) For each mutant, generate weak-mutant-killing constraints correspondingly.

(c) Generate a meta-program by adding the constraints to appropriate position of the original program.

(d) Produce test inputs for the meta-program using dynamic symbolic execution engine.

Mutant Generation
PexMutator contains a set of defined mutation operators where each of them represent a transformation rule that convert the program passed into mutated programs. However, it does not mean that the more mutation operators used, the better the tool performs. Generating too many mutants could cause an issue of wasting time and space resources, therefore, to prevent from this problem and generate tests effectively, PexMutator uses the following five sufficient mutation operators:

1. Absolute Value Insertion
2. Arithmetic Operator Replace
3. Logical Connector Replace
4. Relational Operator Replace
5. Unary Operator Insertion

Mutant-Killing-Constraint Generation
After mutant programs are generated, PexMutator will construct corresponding mutant-killing constraints. However, in practice, solving strong-mutant-killing constraints is expensive in terms of computation cost, thus PexMutator uses weak-mutant-killing constraints during the execution. To satisfy Weak-mutant-killing constraints, the test only requires reachability and necessity criteria to kill the mutants.

Mutant-Killing-Constraint Insertion
One more step is required before adding the constraints into the program, that is reformat the constraints to executable statements so that it is compatible with the original program. The purpose of formatting is that the raw constraints may cause syntactic faults or result in different states from the original code since the external injected code may produce different syntax or unexpected behavior without standardization. In addition, the way of insertion varies for conditional and non-conditional statements.

For conditional statements, the constant is wrapped by if(const) log.write("Mutant Killed"); The if statement creates a new branch that specifically provided for DSE to generate test cases to cover the branch and kill the mutant by solving the mutant constraints in the branch.

Constraint insertion for non-conditional state is more complicated. These states may contains arithmetic operations which are possible to throw exceptions such as division by zero error caused by division operation. Consequently, a simple wrapping like if(op1/op2) wrapping would fail to handle the exception. Fortunately, the problem can be solved by introducing a trigger condition along with the constant, for example, $if((op2 == 0)||((op1 + op2)! = (op1/op2)))$. In this case, if op2 is equal to zero, the expression has the format like if (A or B). When A is true, B will not be executed and the if statement is evaluate to true, thus it kills the mutant while preventing the occurrence of the exception.

Test Generation for the Meta-Program Using DSE
Now the meta-program is transformed and all the mutant constraints are available, the last step is to generate test inputs using the DSE engine. It attempt to generate input tests and discover all the possible paths in the meta-program while satisfying the mutant-killing constraints and weakly kill the mutants.

Since the states of the original program are not affected by the injected mutant-killing constraints, the program is guarantee to response identical result if we use the same input data in the meta-program. In other words, the generated test inputs are reliable and can be safely used for the original program.

PexMutator Implementation
The PexMutator consists of two key components: Meta-program generation and test generation via DSE.

1. Meta-program Generation

Traditionally, a meta-program can be generated either at the source level or at the compiled-file level. PexMutator works in the later level because generating meta-program at the source level is tedious and the test generation is not available until the DSE runs in the compiled file.

The developer used Common Compiler Infrastructure (CCI) to coop with the Microsoft Common Language Runtime (CLR) assemblies. The meta-program is generated in terms of the assemblies and no recompilation is required when use it for test generation.

2.Test Generation via DSE

PexMutator is an extension of Pex, which had been internally used by Microsoft to find critical faults in .NET architecture. The tool is working on the meta-program and use dynamic symbolic execution engine to generate test inputs.

5.2 KLEE

Klee[3], an open-source automated software testing tool developed by Cristian Cadar, Daniel Dunbar, Dawson Engler in Standford University. Klee runs in Linux operation system and it is built on the top of EXE with remarkable improvement in constraint solving optimization. It is widely used in automated testing and finding bugs meanwhile obtaining outstanding performance in achieving a high coverage rate.

KLEE is designed in a way that the testing paths are always corresponding to paths in the unmodified program so that there will be no false positive results. More specifically, the program will firstly be converted into bytecode using LLVM compiler to form instruction sets before passing to KLEE. Then KLEE would interpret the instructions and map them symbolically and stored the process as a state. The entire testing process can be divided into four components:

1. Symbolic representation
2. Collect and solve Path constraint
3. Path selection and execution
4. Error detection

Symbolic representation
The symbolic execution tool established a one-to-one corresponding relationship with all the variables and input data in the program including files, data generated by user interaction, system responding results and network information etc. The entire program is constructed and stored in a tree structure using registers where each node of the tree is a state. The leaves contains symbolic variables, constant and expressions and inter-medium nodes store the representation of symbolic operations.

Practically, state explosion is a critical issue that affect the time-efficiency of testing. Unlike EXE which forks two states whenever it encounters a branch

condition, Klee applies the copy-on-write technique by taking advantage of its ability of tracking all memory objects. That is, whenever a state is intended to be copied, it is marked instead. The actual operation of copy is happening when the marked memory object is modified. In other words, it allows the case that different branches use the shared memory if the memory object never been modified throughout the program. The operation of copy is delayed to a time when it is necessary to do rather than at the point of branching, which effectively improve the speed performance of the testing.

Collect and solve Path constraint
During symbolic execution, all the operations are implemented in terms of symbolic representation rather than concrete values. Therefore, the execution tool can track operations on data and variables, then forming and collecting path constraints at each branch condition. Depending on the results return from constraint solver, executable paths can be predicted.

Path selection and execution
The geometric view of path selection and execution is illustrated in figure 6. Assume root A is the starting of the program, internal node like node B represents a path condition where execution is required to be forked, and the leaves symbolized the current states such as state 1, 2, 3. States marked with dot border means the execution of the corresponding path is finished while the solid border states are to be finished.

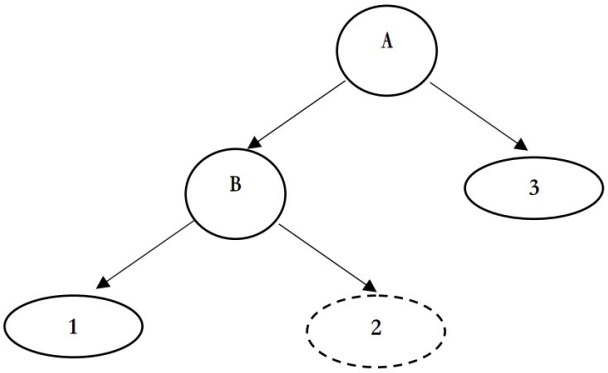

Figure 6 Geometric representation for path selection and execution

The constraint solvers are working concurrently in different paths, then which path should have the execution priority? Klee concatenate and alternatively applies the following two state scheduling methods.

The first strategy is Random path selection. While states are selected by traversing the tree from root to leave, at each branch point, every state in each sub-trees

have the same opportunity to be chosen. Instead of digging into one sub-tree, it is friendly to states on the top of the branch tree, and these states are often with less accumulated constraints has greater chance to discover new paths. Moreover, it prevents starvation, for example, the number of states could be generated dramatically within a loop and causes a problem called "fork bombing". However, random path selection allows the choice of the next executable state not to be limited inside the loop, but providing equal probabilities to any possible next executable state, thus avoid the problem.

The second approach is Coverage-Optimized search which assigns weights for each state and then randomly choose the next state based on the weight. The weight assignment rule is based on the minimum distance to uncovered code, the depth of the state and the most recent coverage history of the state. Coverage-Optimized search utilize the chance of discovering uncovered new code in state scheduling.

Sometimes combing the two methods may be less powerful if one strategy is particularly effective in the situation, but it also decreases the risk of getting stuck when things could go wrong with just one method. Therefore, it improves the overall effectiveness while keeping a high coverage rate.

Error detection
The tool is able to check dangerous operations that may raise errors during symbolic execution. For example, common dangerous operations like division by zero, load or store an address out of boundary, point reference and deference.

Division by zero error is handled by generating a checking branch whenever encounters a division instruction. When the divisor is zero, instead of killing the execution, the negation of the checking branch condition is added so that the path could continues until it hits other types of errors.

Load and store instructions are more complicated because they require to check whether the address within a valid memory object. The idea to solve the issue is to map the memory to a flat byte array so that loads and stores is simply array read and write; However, how to solve the constraints is still a problem that has not yet been resolved.

A point could be referenced by multiple objects thus it could be difficult to do bound checks. In this case, Klee uses a naive method that is deference the pointer to N objects and clone the current state N times for all the objects when read and write operations are performed. This is a time-consuming method, but it is good enough to work with most general cases because one pointer is referred to one object in most of the time.

Optimization work
Constraint solving is the key point of dynamic symbolic execution. Apparently, reduce the cost and improve the efficiency of constraint solving is critical in the

design of an automated testing tool using symbolic execution. The primary optimization methods that Klee involves are expression rewriting, constraint set simplification, implied value concretization, constraint independence and counter-example cache. These approaches are run and proved to be effective in speed-up the testing process.

6 Conclusion

This article reports the methods of both symbolic execution and dynamic symbolic execution. We also discussed the mutant testing method and how it is evaluated. Then we proposed an analysis study on two state-of-the-art test generation tools based on symbolic execution, which are PexMutator and KLEE. Finally, we conclude the current issues and development trend of this study area.

References

1. L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. de Halleux and H. Mei, "Test generation via Dynamic Symbolic Execution for mutation testing," 2010 IEEE International Conference on Software Maintenance, 2010, pp. 1-10, https://doi.org/10.1109/ICSM.2010.5609672
2. M. Papadakis and N. Malevris, "Automatic Mutation Test Case Generation via Dynamic Symbolic Execution," 2010 IEEE 21st International Symposium on Software Reliability Engineering, 2010, pp. 121-130, https://doi.org/10.1109/ISSRE.2010.38
3. Cadar C, Dunbar D, Engler D R. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs[C].OSDI. 2008, 8: 209-224.
4. Chen, T., Zhang, X. S., Guo, S. Z., Li, H. Y., Wu, Y. (2013). State of the art: Dynamic symbolic execution for automated test generation. Future Generation Computer Systems, 29(7), 1758-1773.
5. W. E. Howden, "Weak Mutation Testing and Completeness of Test Sets," in IEEE Transactions on Software Engineering, vol. SE-8, no. 4, pp. 371-379, July 1982, doi: 10.1109/TSE.1982.235571.