A Survey of Symbolic Executions Techniques

Hallet Adrien Sens Loan

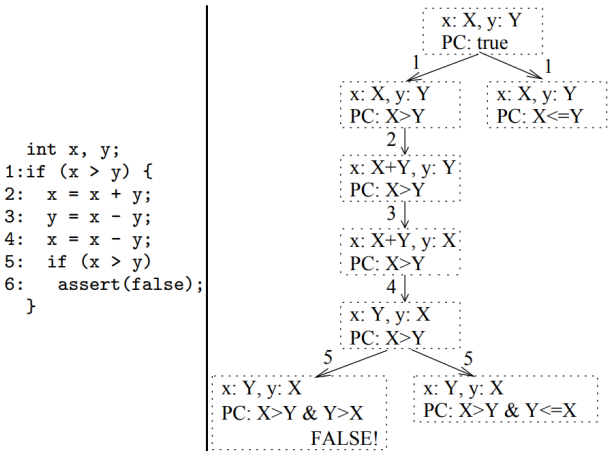

Fig. 1. Swapping two integers and its symbolic execution tree

Abstract

I. Introduction

A. A definition

The first occurences of symbolic execution described the then-new method as a middle ground [12] between the two most-used method of its time: *program testing* and *program provign*. Nowadays, symbolic execution is both described as (part of) the core of many modern techniques to software testing [16] and an effective way to create tests suites with extensive coverage. [5]

B. The concept

The idea behind symbolic execution is to test an algorithm with *symbolic values* rather than concrete values. So instead of using unit testing where a variable is set to a (usually random) value, the symbolic execution maintains a formula that contains all the possible values for the code to reach a particular point in the program. This formula is updated every time the program reaches a branching point. In figure 1, we show an example from [17] of a symbolic execution. Notice how it produces constraints over the variables to explore the algorithm's branching tree.

II. History

We find the first papers on symbolic execution around 1975 [12]. Early methods proposed simple structures to hold conditions with a SAT-solver, the support of simple data types and were focused on algorithm testing (instead of large programs). While papers continue to grow on the subject, it really is around 2005 that symbolic execution really becomes a large thing with more and more frameworks and tools for software verification. The last ten years have seen more papers on the subject[1] (around 200.000) than the three decades following its introduction (half that much). Among the supporters of the method, we can identify two clear research poles, China [6], [19], [20] and the United States of America, with an heavy participation of Microsoft [9] which was pooling heavy resources in software and OS reliability, and the NASA [15], [17], [18].

III. Method

A. Useful concepts

Algorithms can be modeled as graphs where nodes are basic blocks (*i.e.: a part, one or multiple instructions with a single entry and exit point*) and edges are the branches (issued from conditional statements). Def-use pairs use the same concept, although they base their graph on the *definition* and *usage* of a variable. With the *branches* and *def-use pairs*, we can model an algorithm's behavior to follow the values of its variables and determine the *execution path*.

B. Basis

Symbolic execution works on those concepts by updating an internal list of symbols. The execution generates a new symbol for each introduced variable in an algorithm [12]. The symbolic execution runs over the algorithm's statements and builds the symbolic values when it encounters a branching point. The symbolically executed algorithm creates a *state* [17] containing the symbolics values, a counter identifying the next line to be executed and a *path condition*. This path condition is

[1]Data from Google Scholar

a simple boolean formula over the symbols, it creates a constraint for the algorithm to reach the current state of the program (this also allows to check for unreachable paths in programs [2]). The path condition allows to recreate the execution up to its state. The states are stored in a *symbolic execution tree* with the states as nodes and the transitions as edges (see figure 1).

C. Problems

1) State-Space Explosion: Symbolic Execution cannot be that perfect and hosts its bundle of problems that reduce either the confidence in or the performances of the concept. We have seen symbolic execution tree in III-B. Small algorithms can use such methods but actual programs need to be tested in *integration*. In large environments, the tree's branching factor will produce too many nodes (*state-space explosion*) for the performances to stay relevant, sometimes creating infinite loops in the graph [14]. Reducing the state space or pruning them is not enough. To improve the performances, we can depth-first-search the graph but it does not prevent infinite loops until we add a max depth (as KLEE or EXE do). Pruning heuristics (*e.g.:def-use pairs distance*) can be used to reduce the tree's branching factor, we may randomly select a path (emphi.e: the path condition), weighting the shallowest nodes to avoid dead-loops. The random technique is exploited by a lot of *fuzzing techniques* [6] which uses the injection of random values to detect program's faults. Another solution lies in the *concolic execution* (see IV-A).

2) Modeling the memory: Another technical difficulty lies in the *memory model*. As said before, a symbol α represents a variable `a` with a value. In the programming world, it means there is a pointer `a`$\star$ that stores the address to an allocated memory block. The initial approach [12] proposed a *fully symbolic memory*. The symbols are stored in plain states holding their path condition with either a duplication of the states depending on their memory status called *state forking* where every possible execution is forked from the main branch (*e.g.: accessing an array of size 10 with a variable i that depends on the context will fork 10 states from the main memory state, each one with i from 0 to 9*). Fully symbolic memory is therfore slowed by an increased *state-space explosion*. To reduce memory usage, some tools may represent an artificial memory space [13], which reduces the size of the general space by allowing more memory to each path condition, although doing so will void some execution paths, leaving potential bugs out of the symbolic execution. An italian approach [8] proposes the use of a *symbolic* adress which holds the condition to which address the variable points to, which drastically reduces the amount of memory states.

Instead of fully modeling the memory, there also is the *abstract symbol table* [20] which records tuples [variable, address, symbolic value]. This method has the advantage of supporting complex data types (some fully memory models cannot express structures) and memory aliasing (instead of creating a new variable copied from another in older methods).

IV. Variants

A. Concolic execution

Concolic, portmanteau from *concrete* and *symbolic*, the idea of this testing method is to mix symbolic execution alongside concrete ones.

Concolic execution approaches: This technique concept was first introduced on 2005 [10] (more details on section V-A). Since then the idea was further extended and combined with other testing techniques.

However, the general principle has been explored with different angles.

1) Dynamic Symbolic Execution: Dynamic Symbolic Execution (DSE) also kwown as *dynamic test generation* [10] is a popular approach of concolic execution. Its main feature is to have the symbolic execution driven by the concrete execution.

For this method, we will first need to add a new store in order to save the concrete execution information.
The first step involves choosing an arbitrary value as input for our parameters. Then it executes the program concretely and symbolically at the same time updating both stores and the path constraints. When the concrete execution is directed on a certain branch, the symbolic execution follows it and the constraint is appended the the set of path constraints.
In order to explore different paths, we generate new control flows by negating one or more constraints. We can repeat this process as many time as we want to achieve the intended coverage.

Notice that it exists different strategies on the choice of the branch to negate, this crucial heuristic choice depends of the tool.

Downside : Imperfect symbolic execution:

- *False Negative* : Missed path. For example, when another function from the one tested is not symbolically tracked but its result is needed to explore a particular path.

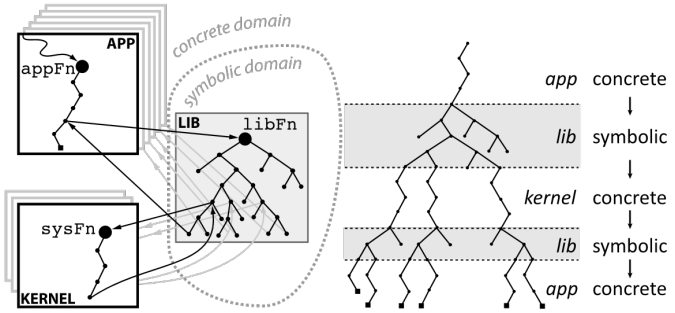

Fig. 2. Multipath/single-path execution: three different modules (left) and the resulting execution tree (right). Shaded areas represent the multipath (symbolic) execution domain, while the white areas are single-path. More details in section V-B2

- *Path Divergence* : In some situations the engine can't guess that no input can provoke an error. In other word, whenever an actual execution path does not match the program path predicted by symbolic execution for a given input vector. For example, assert on a negative value of an absolute value due to the untracked side effect of the `abs()` function. According to [11] they calculated a divergence rates of over 60 %

2) Selective Symbolic Execution: This approach is based on the observation that often only some families of paths are meaningful [7]. We focus the exploration on some designated interesting component of the software while not caring about the others. It offers an illusion of symbolically executing a full software stack, while actually executing symbolically only select components.

The selective symbolic execution switch back and forth between symbolic and concrete execution, depending if the current component is relevant or not. When it is symbolic the tree may expand in width and depth, on the other hand in a concrete execution it only grows in depth as only one branch is created (see Figure IV-A2).

V. Tools and languages

Many tools exist for symbolic execution, Wikipedia mention 22 of them. Another source claiming to "curate a list of awesome symbolic execution resources including tools" mention 35 different tools spread over 10 different languages.

A. DART : *Directed Automated Random Testing*

DART is presented as a tool for automatically testing software using concolic testing method. It was introduced in 2005 making it the first the first tool to be created using concolic techniques and more specifically dynamic symbolic execution techniques (see section IV-A1).

1) Methodology: DART combines three main techniques [10] in order to automate the process of testing for a particular software :

1) An automated extraction of the interface of a program with its external environment using static source-code parsing
2) An automatic generation of a test driver for this interface that performs random testing to simulate the most general environment the program can operate in
3) A dynamic analysis of how program behaves under random testing and automatic generation of new test inputs to direct systemically the execution along alternatives program paths

DART chooses the *depth-first* strategy whenever it has to negate a branch.

2) Example: Let consider the following program :

```
1 Function foo(int x, int y):
2     if x != y then
3         if 2 * x == x + 10 then
4             ERROR;
5         end
6     end
7     return SUCCESS;
```

This function is defective as it may lead to an error statement for some value of x and y.
DART start by guessing values for both x and y for instance 269167349 and 889801541. With this values the function return successfully, during the execution two predicates were formed created by the `if` statements, in our case the path constraint at the end is : $\langle x_0 \neq y_0, 2 \times x_0 \neq x_0 + 10 \rangle$ with x_0 and y_0 both beings *symbolic variables*.
While we maintain this predicates, all path will lead to the same end. So in order to force the program through a potential different outcome we change one of the predicate and look at the result. If we negate the last predicate we have the following path constraint : $\langle x_0 \neq y_0, 2 \times x_0 = x_0 + 10 \rangle$ in which $x_0 = 10$ and $y_0 = 889801541$ is a solution. Using this values as inputs the program end up into the `ERROR` as wanted.

3) Key strength/originality: One of the main strength of DART is that, on any compilable program, testing can be performed completely automatically. There is no

need to write any additional harness code or test driver. During testing, DART detects standard errors such as program crashes, assertion violations, and non-termination.
Due to its based on precised dynamic analysis, DART provides an interesting alternative to static analyzer. It is expected to be very good at the detection of some kind of bugs[2] compare to a dynamic analyzers.
Any errors founded by DART during the execution is sure to be sound.

But DART has its own limitations, particularly the limited effectiveness of dynamic test generation to improve over random testing and the computational cost of running tests.

B. S^2E

S^2E is a platform for writing tools that analyze the properties and behavior of software system. It comes as a modular library that gives virtual machines symbolic execution and program analysis capabilities. [1] It uses selective symbolic execution concolic execution (see section IV-A2).

1) Methodology: S^2E switch dynamically between symbolic and concrete execution. This transition is not trivial and must be treated separately.
Let `A` and `B` be function such as `A` calls `B`.

From concrete to symbolic and back: `A` calls `B` with concrete arguments as we are still in a concrete execution. However `B` is symbolic, therefore it modify the arguments sends by `A` to make them symbolic. For instance `B(10)` become `B(`λ`)`. In can additionally be set to some constraints, for example `B(`$\lambda < 15$`)`.
After the transition occurs, S^2E executes `B` symbolically using the symbolic argument(s) and at the same time with the concrete argument(s).
Once the exploration of `B` is over, S^2E returns to `A` the concrete returned value of the concrete execution.
In this way, the execution of `A` is always consistent, meanwhile S^2E use the extra symbolic path in its analyzer plugin to check for possible issues.

From symbolic to concrete and back: `A` calls `B` with symbolic arguments as we are still in a symbolic execution. However `B` is concrete, therefore it modify the arguments sends by `A` to make them concrete while still maintaining path constraints. For instance if we have the path constraint $\langle \lambda < 5 \rangle$, we can choose -20 or 3 as a concrete value but not 5 or 42.

[2]Especially interprocedural bugs and bugs that happen through library functions use

After this choice, `B` execute concretely then return symbolically back to `A` which resume symbolically.

Note that S^2E actually employs "lazy-concretization": It converts the value of λ from symbolic to concrete on-demand, only when concretely running code is about to branch on a condition that depends on the value of λ.

It's important to mention that this transition may lead to the "overconstraining" problem, which impact both the soundness and completeness of the tool. The problem that may rises can be :

- Exclusion of paths due to the concretisation of the arguments
- Exclusion of paths indirectly due to the the constrained return value and side effects

2) Example: Let's use the figure IV-A2 as a reference example, it describes an application *app* using a library *lib* on top of an OS *kernel*. We are interested in testing the *lib*, but not the *app* nor the *kernel*. Therefore only the *lib* will be executed symbolically, the 2 others will be concrete.

app call the function *appFn* which itself calls a *lib* function *libFn*, which eventually invokes a system call *sysFn*. After that *sysFn* returns, *libFn* does some extra execution and then eventually returns to *appFn*. Once the execution crosses into the symbolic domain (gray shaded part) from the concrete domain (white part), the execution tree expands. After the executions returns again to the concrete domain, the execution tree no longer expand, it may still grows, but it does not add any new paths until the execution become symbolic domain again. A path may terminate prematurely, for example in case of hitting a crash bug.

C. EXE

Section III-C1 stated the memory problem. **EXE** is a tool (focused on C) that aims to reduce that with the use of concolic execution within the states, allowing to partially store the states and using the path condition to recreate the output if needed. The tool bases itself over the principle of *EGT - Execution Generated Testing* [4], which is basically the classic approach to symbolic execution. The tool uses a constraint solver working on the path condition to prune the tested algorithm's branches (simply deleting unsatisfiable constraints, branches that cannot logically exist can be generated due to the symbolic value, this pruning removes them), then to solve the constraint to generate a set of test value. The tool

also uses models to check code coverage, which is only optional.

Although good for its time, EXE still suffers from the same problems and a new one; *isolation*. The tool is isolated from the environment and any kind of interaction ruined the double execution of symbolic and concrete data. Interacting with a network, input, ... would result in the loss of the symbolic execution or would require a full model of the world to interact with (which ranges from difficult to impossible). Furthermore, the need for a model for sufficient coverage requires a realistic and complete model. Functions which return the same value for different outputs (*e.g.: in Java, if you compare an object to another*) or worse; different values for the same input, would result in models that would not quite work with the tool. On top of that, the tool works assuming it has a good SAT-solver (and as we know, 3-SAT is np-complete) and available memory.

D. KLEE

This tool [3] is a complete rework of EXE, also focused on C, by the same authors three years later (2008). While the basis were the same, KLEE managed to largely improve the execution time and space by keeping its advantages (*exact memory representation*, constraint solving, high code coverage). To reduce the space bottleneck, KLEE stores the states concurrently, sharing them, allowing larger sets of states to be used by the same execution. It is important to note that KLEE does not mindlessly check each instruction. Sets of dangerous operations which can lead to failures rather than wrong outputs are determined. In the C-language context, it means memory accesses and allocation.

VI. Conclusions

References

[1] S2e: A platform for in-vivo analysis of software systems.

[2] Romain Aissat, Frédéric Voisin, and Burkhart Wolff. Infeasible paths elimination by symbolic execution techniques. In Jasmin Christian Blanchette and Stephan Merz, editors, *Interactive Theorem Proving*, pages 36–51, Cham, 2016. Springer International Publishing.

[3] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[4] Cristian Cadar and Dawson Engler. Execution generated test cases: How to make systems code crash itself. In Patrice Godefroid, editor, *Model Checking Software*, pages 2–23, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[5] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. 56:82–90, 02 2013.

[6] Chen Chen, Baojiang Cui, Jinxin Ma, Runpu Wu, Jianchao Guo, and Wenqian Liu. A systematic review of fuzzing techniques. *Computers and Security*, 75:118 – 137, 2018.

[7] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. The s2e platform: Design, implementation, and applications. *ACM Transactions on Computer Systems (TOCS)*, 30(1):2, 2012.

[8] Emilio Coppa, Daniele Cono D'Elia, and Camil Demetrescu. Rethinking pointer reasoning in symbolic execution. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 613–618, Piscataway, NJ, USA, 2017. IEEE Press.

[9] Manuel Costa, Miguel Castro, Lidong Zhou, Lintao Zhang, and Marcus Peinado. Bouncer: Securing software by blocking bad input. pages 117–130, Stevenson, Washington, USA, October 2007. Association for Computing Machinery, Inc.

[10] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM, 2005.

[11] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *NDSS*, 2008.

[12] J.C. King. A new approach to program testing. 10:228–233, 06 1975.

[13] M. Papadakis and N. Malevris. Automatic mutation test case generation via dynamic symbolic execution. In *2010 IEEE 21st International Symposium on Software Reliability Engineering*, pages 121–130, Nov 2010.

[14] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *2010 IEEE Symposium on Security and Privacy*, pages 317–331, May 2010.

[15] S. Shen, S. Ramesh, S. Shinde, A. Roychoudhury, and P. Saxena. Neuro-Symbolic Execution: The Feasibility of an Inductive Approach to Symbolic Execution. *ArXiv e-prints*, July 2018.

[16] David Trabish, Andrea Mattavelli, Noam Rinetzky, and Cristian Cadar. Chopped symbolic execution. In *International Conference on Software Engineering (ICSE 2018)*, 5 2018.

[17] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. Test input generation with java pathfinder. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '04, pages 97–107, New York, NY, USA, 2004. ACM.

[18] Guowei Yang, Suzette Person, Neha Rungta, and Sarfraz Khurshid. Directed incremental symbolic execution. 2014.

[19] Rui Zhang and Cynthia Sturton. A recursive strategy for symbolic execution to find exploits in hardware designs. pages 1–9, 06 2018.

[20] Dai Ziying, Xiaoguang Mao, Ma Xiaodong, and Wang Rui. A memory model for symbolic execution, 01 2010.