A Survey of
Symbolic Executions Techniques

Hallet Adrien
adrien.hallet@student.uclouvain.be

Sens Loan
loan.sens@student.uclouvain.be
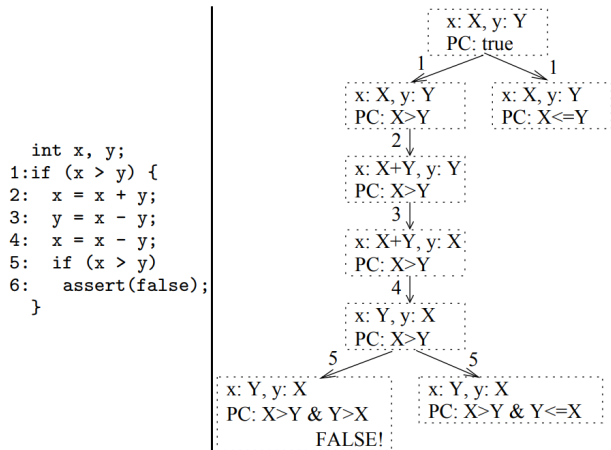
```
        int x, y;
1:if (x > y) {
2:    x = x + y;
3:    y = x - y;
4:    x = x - y;
5:    if (x > y)
6:      assert(false);
    }
```

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

ABSTRACT

This survey aims to introduce the reader to Symbolic Execution as a technique to find bugs in a problem. We will introduce the concepts, challenges, some variants to the main principle and tools that implement them.

I. DEFINITION

The first occurences of symbolic execution described the then-new method as a middle ground [14] between the two most-used method of its time: *program testing* and *program proving*. Nowadays, symbolic execution is both described as (part of) the core of many modern techniques to software testing [21] and an effective way to create tests suites with extensive coverage. [7]

II. HISTORY

We find the first papers on symbolic execution around 1975 [14]. In 2005, a regain of interest in the subject[1] revives the worldwide research with two clear poles,

---

[1]Google Scholar reports more than twice the amount of papers from 2005 to today than thre three preceding decades

China [8], [25], [26] and the United States of America, with an heavy participation of Microsoft [11] which was pooling heavy resources in software and OS reliability, and the NASA [20], [23], [24].

III. METHOD

A. Objective

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 [23] of a symbolic execution. Observe how it produces constraints over the variables to explore the algorithm's branching tree.

B. Handling branches and variables

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). With the *branches* and the constraints applied to variables (*e.g.: a math operation on an integer*), we can model an algorithm's behavior to follow the values of the variables and determine the *execution path*.

C. 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 [14]. 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* [23] containing the symbolics values, a counter identifying the next line to be executed and a *path condition*. This path condition is

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 [4]). The set of path conditions (*path constraint*) 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).

D. 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-C. 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 [17].

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, see sections V-D, V-E).

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* [8] which uses the injection of random values to detect program's faults. Another solution lies in the *concolic execution* (see IV-A).

Very recently (july 2018), a team of researchers [20] managed to successfully test programs that were previously untested due to complex memory usage leading to complex explosion. They used a new approach with neural networks to solve the constraints within the states.

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* that stores the address to an allocated memory block. The initial approach [14] 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 [16], 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 [10] proposes the use of a *symbolic* address 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* [26] 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*, is a testing method in which we mix symbolic executions alongside concrete ones. Using both concrete and symbolic values is a widely used technique that many contemporary tools use because of its ability to speed up the execution while reducing the used memory.

Concolic execution approaches: This technique concept was first introduced in 2005 [12] (more details on section V-A). Since then the idea was further extended and combined with other testing techniques.

1) Dynamic Symbolic Execution: Dynamic Symbolic Execution (DSE) also kwown as *dynamic test generation* [12] is a popular approach of concolic execution. Its main concept is to drive the symbolic execution by the concrete execution.

For this method, we will first need to add a new store, a memory structure, 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 concurrently, updating both the 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.
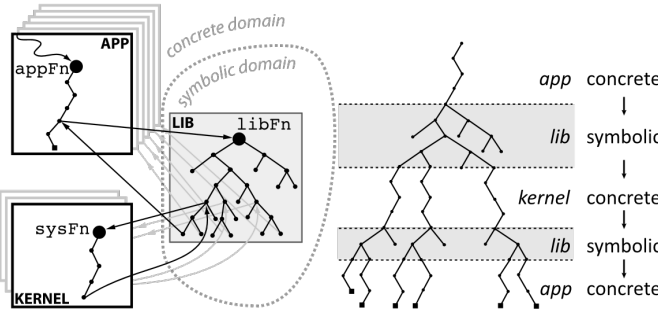In order to explore different paths, we generate new

Fig. 2: Multipath/single-path execution: three different modules (left) and the resulting execution tree (right). Gray shaded areas highlight symbolic execution, while the white are concrete executions. More details in section V-F2

control flows by negating one or more constraints. We can repeat this process as many time as we want to achieve the desired coverage.

Note that different strategies exist on the choice of the branch to negate, this choice depends on the used tool.

Downside : Imperfect symbolic execution:

- *False Negative* : Missed path. For example, when another function from the one currently tested has no symbolic tracking but needs the result to further explore the branch.
- *Path Divergence* : In some situations the symbolic engine can produce false positive because of the *isolation problem*(see section V-D). The tool then returns an input that can provoke an error in the program, only that said input cannot exist in the context. For example, the absolute value function could be called in a program. The same program would later use the value and the symbolic engine will signal an error for a negative value. Only a negative value cannot exist because of the absolute value, but the path condition has not been updated because the solver did not know how the external call changed the constraints.
  According to [13] 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 [9]. We focus the exploration on some designated interesting components of the software while not caring about the others. It offers an illusion of symbolically executing a full software, when

in reality it only executes a partial symbolic execution on the tested components.

The selective symbolic execution switches back and forth between symbolic and concrete execution, depending on the current component's relevance. When it is symbolic, the tree may expand in width (higher branching factor) and depth, on the other hand, in a concrete execution it only grows in depth as only one branch is created (see figure 2).

V. TOOLS AND LANGUAGES

Many tools exist for symbolic execution, Wikipedia mentions 22 of them. Another source claiming to *curate a list of awesome symbolic execution resources including tools* mentions 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, placing it as 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 [12] in order to automate the process of testing for a particular software:

1) *Automatic extraction of program interface*: Extract the interface of a program, including its external environment by using static source-code parsing.
2) *Automatic test generation*: Generate a test driver for this interface. It will perform random testing simulation using the most common environment the program may execute in.
3) *Dynamic analysis*: Perform a dynamic analysis on how the program behaves under random testing and automatic generation of new test inputs to drive systemically the execution along different program paths.

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

2) Example: Let consider the following program :

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 these values the function return successfully, during the execution two

```
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;
```

predicates were formed created by the `if` statements, in our case the final path constraint is: $\langle x_0 \neq y_0, 2 \times x_0 \neq x_0 + 10 \rangle$ with x_0 and y_0 being *symbolic variables*.

While we maintain these predicates, all paths will lead to the same termination. 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 $y_0 = 889801541$ are solutions. Using these values as inputs, the program end up in the `ERROR` state, as desired.

3) Key strength/originality: One of the main strengths 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 crash, assertion violation, and non-termination. Because it is based on precised dynamic analysis, DART provides an interesting alternative to static analyzers. It is expected to be very good at the detection of some kind of bugs[2] compared to a dynamic analyzers.

Any errors found 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. JDART

JDART describes itself as a dynamic symbolic analysis framework for *Java* [15].

The main goal of the tool is to build a dynamic symbolic analysis tool that can be used at industrial scale complex softwares. Another important aspect was the main design guideline; they wished it to be modular and extensible.

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

Looking at the repository of the project, it started around 2015. But it seems that it stagnates since 2017 based on the code activity.

C. Cute & jCUTE

CUTE and *jCUTE* are automated concolic testing tools, CUTE stands for Concolic Unit Testing Engine [18]. The first one is for *C* programs while the second targets *Java* ones.

Both tools consist of two main modules :
- *Instrumentation*[3]: to add some code in the program under test so that the program calls the library.
- *Library*: to perform symbolic execution, solve constraints and control thread schedule.

1) Key strength/originality: CUTE's popularity, apart from the *instrumentation module* ease of use, comes from its scalability. Instead of using the full path condition, CUTE builds the path satisfaction over **simple symbolic constraints on the variables** via their pointers. They are only approximation of the full path condition, but the speedup makes it worth.

Along with the arguably [19] better performances and user experience for its time (2005), it is important to note that CUTE not only has good internal program modelisation [18], but also a in-depth memory representation with data structures and data structures invariants that use a linked list technique for solving [19], producing a $\mathcal{O}(n)$ pointer trasversal (from $\mathcal{O}(n^2)$), allowing the use of complex data structures.

2) Abandon: As of today, *CUTE* is obsolete and has been replaced by *CREST* [1]. The last commit on the official repository of the project dating back to 2017.

jCUTE is the first concolic execution engine for *java* [15] designed and implemented around 2006. It is no longer maintained, the last commit on the official repository of the project dating back to 2013.

D. EXE

Section III-D1 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* [6], 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

[3]more precisely it uses *CIL* and *SOOT* compiler frameworks

(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.

E. KLEE

This tool [5] 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. Each time the solver reaches one of those dangerous operations in an algorithm, KLEE tries all the values allowed by the current path condition and returns all the inputs leading to faulty results.

Key strength/originality: KLEE is arguably one of the most advanced symbolic execution frameworks up to this day. Its new architecture [5], inherited from the EXE's experience, reduces usage complexity, time and space. To reduce **memory** usage, KLEE does not use a OS-generated program state as EXE did, but instead maps pages tables to a virtual heap to share memory, similar to a forking process accessing shared memory. It is called *compact internal state representation*, increasing the concurrent state amount without decreasing the memory representation quality. To reduce **time** (in relation to other symbolic execution tools), KLEE introduces many *sat-solving optimisations*. As satisfiability is a NP-complete problem and KLEE generates many of constraints that have to be solved (either to reduce a path condition form or to find a faulty input from the path condition), we can expect high computation times. The tool is able to reduce this execution time up to 15 faster than without optimisations. We can observe *klee* :

- *Expression simplification*: maintain the most simple path condition everytime a constraint is added to reduce further computation time. For example, if a first branch requires a positive number and a second condition along this path requires an integer between 10 and 20, we can drop the first condition.
- *Implied value concretization*: we have seen that both EXE and KLEE sometimes use concrete value because they are cheaper. If during an execution the condition of a variable is an equation, KLEE solves the equation to store the concrete(s) value(s) of the variable instead of a path condition.
- *Constraint independance*: inherited from EXE, this optimisatin stores the constraints in groups sorted by the accessed variables. Constraints that do not access the same variable have no need to be checked simultaneously because they do not overlap.
- *Counter-exemple cache*: is a cache of queries to eliminate both redundant queries (slightly similar to expression simplification) and unsolvable queries via a map of the constraints. We could therefore eliminate $x < 0, x > 0$ as soon as the constraint is introduced.

The tool also speeds execution up with the use of both complex heuristics on the paths and path conditions (they also use random path exploration). As many of the symbolic execution tools, KLEE has been open source since 2009 [2] and is still active today. On the negative side, we can note that KLEE does not support multithreaded algorithms, and has trouble with complex memory objects (especially those varying in size and symbolic floating points).

F. S^2E

S^2E is a platform for writing tools that analyze the behavior and properties of software system. It comes as a modular library that gives virtual machines symbolic execution and program analysis capabilities. [3] It uses selective symbolic execution concolic execution (see

section IV-A2).

The platform is related to the previously mentioned KLEE tool (see section V-E) as S²E is built upon its engine and the QEMU virtual machine emulator. The platform was launched in 2011 and is still update until now according to last commits on the project repository.

1) Methodology: S²E switches dynamically between symbolic and concrete execution. This transition is not trivial and must be treated separately.
Let A and B be functions such as A calls B.

From concrete to symbolic and back: A calls B with concrete arguments (because A is a component that is not symbolically relevant) as we are still in a concrete execution. However B is symbolic, therefore it modifies the arguments sent by A to make them symbolic. For instance B(10) becomes B(λ). In can additionally be set to some constraints, for example B($\lambda < 15$).
After the transition occurs, S²E symbolically executes B, concurrently using the symbolic argument(s) and the concrete argument(s).
Once the exploration of B is over, S²E returns the concrete execution's final value to A.
This way, the execution of A is always consistent, meanwhile S²E uses 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 (because B is a component that is not symbolically relevant) as we are still in a symbolic execution. However B is concrete, therefore it modifies the arguments sent 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.
After this choice, B execute concretely then return symbolically back to A which resume symbolically.
Note that S²E actually employs *lazy-concretization*: It converts the value of λ from symbolic to concrete on-demand, only when the concrete execution is about to branch on a condition that depends on the value of λ.

It is important to mention that this transition may lead to the *overconstraining* problem, which impact both the soundness and completeness of the tool. The problems that may arise can be :
- Exclusion of paths due to the concretization of the arguments.
- Exclusion of paths indirectly due to the the constrained return value and side effects.

2) Example: Let us use the figure 2 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 execution returns again to the concrete domain, the execution tree no longer expands, it may still grow, but it does not add any new paths until the execution goes in the symbolic domain again. A path may terminate prematurely, for example in case of hitting a crash bug.

```
1  Function libFn(int x):
2      if x < 5 then
3          buf = sysFn(x);
4          if x < 0 then
5              ...
6          end
7      end
```
Algorithm 1: libFn function (partial) code

Let us go in further detail on the *libFn* to the *sysFn* transition (symbolic to concrete transition). Let the code of the *libFn* function be as described in the algorithm 1 and their is no path constraints when the function is called.
On the first if branch, the path split on the constraint $x < 5$ which is added to the *path constraints*. The *sysFn* function call only happen on the *then* part of the branch, as it is a concrete function we must concretize the value of x according to our current *path constraints* ($\langle x < 5 \rangle$). Thus the tool make the arbitrary choice of 4 for x which is consistent with our *path constraints* to then perform *sysFn(4)*. This concretization added a $x = 4$ constraint into the *path constraints*.
After the concrete execution of *sysFn* it returns to *libFn* in the symbolic domain. But notice that *sysFn*'s returned value is correct only on the constraint added by the concretisation as all computation was done assuming that x value was 4.

VI. Conclusions

We hope that this survey has been a good introduction to the world of symbolic execution, both in the general context and the current tools that are still being developed since what we could call a rebirth of the field in 2005, thanks to the big companies previously cited (see History). It is also important to note that softwares are not the only ones being symbolically tested. While not discussed here, we can cite usage of symbolic execution in hardware [25] and networks [22]. Due to the limited size of this paper, we were not able to extensively look into techniques and complex mechanisms but we hope that we planted a seed of curiosity or interest in the subject. To wrap this survey up, we could insist on the emergence of more and more complex tools that, while still being slowed down by the almighty state-space explosion, gain in power each year. Symbolic Execution, for now, is still a program testing solution that is mainly used in academia or large companies that can afford it. Small and medium companies still prefer unit and integration testing that is easier and cheaper to setup and maintain. But this could change in the future.

References

[1] Open Systems Laboratory, October 2018, http://osl.cs.illinois.edu/software/index.html.

[2] Klee Official Git Repository, https://github.com/klee/klee.

[3] S2E Systems website, October 2018, https://s2e.systems.

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

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

[10] 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.

[11] 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.

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

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

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

[15] Kasper Luckow, Marko Dimjašević, Dimitra Giannakopoulou, Falk Howar, Malte Isberner, Temesghen Kahsai, Zvonimir Rakamarić, and Vishwanath Raman. Jdart: A dynamic symbolic analysis framework. In Marsha Chechik and Jean-François Raskin, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 442–459, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[16] 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.

[17] 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.

[18] Koushik Sen and Gul Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *International Conference on Computer Aided Verification*, pages 419–423. Springer, 2006.

[19] Koushik Sen, Darko Marinov, and Gul 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*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[20] 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.

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

[22] Mathy Vanhoef and Frank Piessens. Symbolic execution of security protocol implementations: Handling cryptographic primitives. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, Baltimore, MD, 2018. USENIX Association.

[23] 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.

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

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

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