Symbolic Execution and Recent Applications to Worst-Case Execution, Load Testing and Security Analysis

Corina S. Păsăreanu, Rody Kersten, Kasper Luckow, Quoc-Sang Phan

Abstract

Symbolic execution is a systematic program analysis technique which executes programs on symbolic inputs, representing multiple concrete inputs, and represents the program behavior using mathematical constraints over the symbolic inputs. Solving the constraints with off-the-shelf solvers yields inputs that exercise different program paths. Typical applications of the technique include test input generation and error detection. In this chapter we review symbolic execution and associated tools, and describe some of the main challenges in applying symbolic execution in practice: handling of programs with complex inputs, coping with path explosion and ameliorating the cost of constraint solving. We also survey promising applications of the technique that go beyond checking functional properties of programs. These include finding worst-case execution time in programs, load testing and security analysis, via combinations of symbolic execution with fuzzing.

1 Introduction

As computer systems become more pervasive and complex, it has become increasingly important to develop techniques and tools that effectively ensure software dependability. Symbolic execution [57] is a systematic program analysis technique which explores multiple program behaviors at once, by collecting and solving symbolic path conditions collected over program paths. Symbolic execution can be used for finding bugs in software, where it checks for runtime errors or assertion violations during execution and it generates test inputs that trigger those errors.

Nowadays there are many symbolic execution tools available [22, 78, 28, 43, 24, 89] which have found numerous vulnerabilities and other interesting bugs in software. Much of the success of symbolic execution in recent years is due to significant advances in constraint solving and decision procedures [33, 6] as well as to the availability of increasingly cheap computational power and cloud computing platforms [43, 28], allowing to scale the technique to large applications.

In this chapter we review symbolic execution and associated tools, and we describe the main challenges in applying symbolic execution in practice: handling of programs with complex inputs, coping with path explosion and ameliorating the cost of constraint solving. We also survey some applications of the technique that go beyond checking functional properties of programs. These include finding worst-case execution time in programs, load testing and security analysis, via combinations of symbolic execution with fuzzing. These applications are perhaps less studied in the literature but we believe they hold much promise for the future. We conclude with directions for future work.

2 Symbolic Execution

Symbolic execution [57] is a program analysis technique that executes a program on symbolic, instead of concrete, input values and computes the effects of the program as *functions* in terms of these symbolic inputs. The result of symbolically executing a program is a set of symbolic paths, each with a path condition PC, which is a conjunction of constraints over the symbolic inputs that characterizes all the inputs that follow that path. All the PCs are disjoint.

When executing a branching instruction with condition c, symbolic execution systematically explores both branches and updates the path condition accordingly: $PC \leftarrow PC \wedge c$ for the *then* branch and $PC \leftarrow PC \wedge \neg c$ for the *else* branch. The feasibility of the path conditions is checked using off-the-shelf constraint solvers such as Z3 [33]. If a path condition is found to be unsatisfiable, symbolic execution stops analyzing that path (since that path is not feasible). For the feasible paths, the models returned by the constraint solver can be used as test inputs that execute these paths. To deal with loops and recursion, typically a bound is put on the exploration depth.

Several tools implement "classic" symbolic execution which is essentially a static analysis technique, as it analyzes a program without running it; in Symbolic PathFinder, the program is actually "run", but this is done inside the *custom* JVM of the Java pathFinder tool. Dynamic symbolic execution techniques, on the other hand, collect symbolic constraints at *run time* during concrete executions. Examples of such dynamic techniques are implemented in DART (Directed Automated Random Testing) [42] and Klee [22].

Dynamic test generation as first proposed by Korel [60], consists of running the program starting with some random inputs, gathering the symbolic constraints on inputs at conditional statements, using a constraint solver to generate new test inputs and repeating the process until a specific program path or statement is reached. DART performs a similar dynamic test generation, where the process is repeated to attempt to cover *all* feasible program paths, and it detects crashes, assert violations, runtime errors etc. during execution.

2.1 Complex heap data structures

Invented in the 70s, traditional symbolic execution has been proposed for programs with a fixed number of numerical inputs. However, modern programming languages such as C++ and Java contain a variety of data structures, e.g. linked lists or trees, that might dynamically allocate objects at run time. A naive approach to this problem is to impose *a priori* bounds on the inputs. For example, for a program that takes a linked list as input, one needs to initialize it with k list nodes, and each one can be symbolic. However, k have to be defined beforehand. A pessimistically large k leads to path explosion problem, and small k (incorrectly) reduces the search space of symbolic execution. Moreover, it is not straightforward to describe the bounds for data structures such as tree.

To address the problem above, Khurshid et al. [55] introduced the *lazy initialization* algorithm, which has become the state-of-the-art way of handling heap data structures. This algorithm works as follows.

1. When a symbolic input is of reference type, i.e. linked list, execute the program without initializing it.

2. When an uninitialized symbolic variable is de-referenced, exhaustively enumerate all possible concrete objects that it can reference to **(i)** null; **(ii)** new object; **(iii)** previously initialized objects of the same type (i.e. it is an alias)

In the second step, symbolic execution case splits on each of possible choices, which leads to rapid path explosion. Therefore, there have been multiple efforts on improving the enumeration of this step.

Deng *et al.* proposed the *lazier* algorithm [34], which delays case splitting in lazy initialization by grouping together choices in (ii) and (iii) (non-null choices), into a symbolic variable. Case splitting on non-null variables occurs later when they are accessed. The same authors then introduced a more enhanced algorithm, called *lazy#* [35], with group together all choices in (i), (ii) and (iii) in the same manner.

Symbolic initialization [47] uses a guarded value set to capture all choices in (i), (ii) and (iii) in the same symbolic heap. This completely avoids case splitting in symbolic paths when initializing a symbolic variable of reference type. This, however, comes with the cost of solving constraints with greater complexity, since case splitting is actually delegated to the SMT solver.

Geldenhuys *et al.* [40] took a different approach, instead of delaying case splitting, the authors aim to reduce the number of choices in (iii) by considering only non-isomorphic structures. This is done via pre-computed tight field bounds. Computing those bounds is very expensive, but the authors argue that they can be re-used to test different methods in the program.

The lazy initialization-based approaches have been adapted to take into account the shape of the input. For example, when the input is designed to be singly linked list, the choices in (iii) should be restricted to avoid configurations of a circular linked list and so on. To address this problem preconditions are

used in e.g. [55, 92] to constraint heap inputs. This is implemented as using an API, `verify.ignoreIf`, to tell symbolic execution to stop exploring when a method *pre*() representing preconditions returns false. This approach delegates testing preconditions to the users. For example, to impose the constraint that the input is a binary tree, such a method *pre*() needs to implement a depth-first search to detect cycles.

Braione *et al.* [16] introduced *Heap EXploration Logic* (HEX) as a specification language for lazy initialization. When symbolic execution enumerates the choices in (i), (ii) and (iii), the HEX verifier checks those choices against a specification, written in HEX, and prune off invalid states. HEX lacks numerical operators, and thus it cannot express numerical properties of the data structures such as the size of a linked list. The HEX language is not expressive enough to describe shapes of data structures either. Users have to provide methods, called *triggers*, to check properties that cannot be checked by HEX.

Pham *et al.* [71] use separation logic [48, 80] with inductive definitions to describe the symbolic heap and the shape of the input data structures. When an uninitialized symbolic variable is de-referenced and if it is defined by an inductive predicate, they unfold it to capture the footprints, i.e. the resources it accesses. This unfolding process updates the heap configuration, and a SAT solver for separation logic [62] is then used to check if the updated heap configuration is satisfiable.

3 Tools and Scalability Challenges

Because of its capability of finding subtle bugs, and its applications in a widespread of domains, symbolic execution has been developed on several platforms, for different programming languages. The following table contains a (likely incomplete) list of symbolic executors.

Language	Tool	Link
Java (bytecode)	Symbolic PathFinder [78]	https://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc
	Java StarFinder [71]	https://github.com/star-finder/jpf-star
	jCUTE	https://github.com/osl/jcute
	janala2	https://github.com/ksen007/janala2
	jDART [65]	https://github.com/psycopaths/jdart
	JBSE [17]	https://github.com/pietrobraione/jbse
	KeY	http://www.key-project.org/
X86(-64) binaries	Project Springfield	https://www.microsoft.com/en-us/springfield/
	SAGE [43]	—
	Mayhem [24]	http://forallsecure.com/mayhem.html
	Miasm (many different binaries)	https://github.com/cea-sec/miasm
	BAP (Also ARM) [20]	https://github.com/BinaryAnalysisPlatform/bap/
	S^2E (Also ARM) [27]	http://s2e.epfl.ch
	Angr	http://angr.io/
	Pathgrind	https://github.com/codelion/pathgrind
	pysymemu	https://github.com/feliam/pysymemu/
	Triton	http://triton.quarkslab.com
C / C++	CUTE [82]	—
	CREST	http://www.burn.im/crest/
	DART [42]	—
	KLOVER [63]	—
	EXE [23]	—
	Otter	https://bitbucket.org/khooyp/otter/overview
LLVM	KLEE [22]	http://klee.github.io/
	Cloud9	http://cloud9.epfl.ch/
	Kite	http://www.cs.ubc.ca/labs/isd/Projects/Kite/
.NET	Pex	http://research.microsoft.com/en-us/projects/pex/
JavaScript	SymJS	http://www.cs.utah.edu/~ligd/publications/SymJS-FSE14.pdf
	Jalangi2	https://github.com/Samsung/jalangi2
	Kudzu [81]	—
Dalvik bytecode	SymDroid [50]	—
Python	PyExZ3 [8]	https://github.com/thomasjball/PyExZ3
VineIL	BitBlaze [85]	http://bitblaze.cs.berkeley.edu
	FuzzBALL [7]	http://bitblaze.cs.berkeley.edu/fuzzball.html
Boogie	Symbooglix [64]	https://github.com/symbooglix/symbooglix
CIVL language	CIVL	http://vsl.cis.udel.edu/civl/
Ruby	Rubyx	http://www.cs.umd.edu/~avik/papers/ssarorwa.pdf

3.1 Challenges

There are two main challenges in scaling up symbolic execution: there are too many paths to explore and the path conditions are too difficult to solve. Addressing these two challenges are active areas of research.

Path explosion. Recall that symbolic execution explores symbolic paths of the program, which form a (symbolic execution) tree. Each path of the tree is independent of the others, thus there have been multiple efforts on parallelizing symbolic execution [56, 84, 87, 28, 46], and distributing the exploration process to multiple workers. This idea is very promising thanks to recent advances in cloud services, however balanced distribution of workload among workers remains a big challenge, as the depth and breadth of the symbolic execution tree are not known in advance.

Other approaches to the path explosion problem reducing the number of paths using state or path merging and also compositional techniques [41, 3, 61, 5, 83, 79]. Some of these approaches use disjunction or set to represent (symbolic) values of the resultant merged states. Thus the reduction of the number of path comes with the cost of solving constraints with greater complexity. Related techniques use different forms of abstraction to reduce the number of paths in looping programs [49, 44].

Other techniques use sampling or different search heuristics [38] to try to hit the bug faster using sampling. The idea is that most symbolic execution engine employs depth-first search, which systematically searches the symbolic

execution tree from one side to the other. When the execution tree is too big for exhaustive search, depth-first search may always get stuck in the beginning parts of the tree, and thus sampling can increase the chances of hitting the bugs.

Improving constraint solving. As constraint solving is expensive, an intuitive idea is to cache the result, and look up the cache before invoking the solver. KLEE [22] exploits the fact that when a program has independent branches, the path condition will be comprised of independent constraints. Therefore, decomposing the constraints into multiple independent subsets, and caching results for those subsets increases the possibility of cache hit.

Green [91] took one step further, normalizing the constraints and then saving them, together with their results, offline to a database. In this way, constraint solving can be reused across programs, analyses and solvers. Green was implemented for linear integer constraints, while Cashew [18], built on top of Green, extends the Green approach to string constraints.

In a different context, since symbolic execution often has to be run several times on the same program, e.g. first to check the error, then to verify the program after fixing the error, memoized symbolic execution [93] uses a trie to store the whole symbolic execution tree in the first run, then re-uses the summaries from the trie in the following runs.

Another active research area is to extend symbolic execution to programs with complex constraints, such as non-linear numerical constraints [86] or combination of string constraints and numeric constraints [96, 10].

4 Applications

4.1 Worst-Case Execution Time (WCET) Analysis

Symbolic execution has been used in several works related to real-time systems. Real-time systems are characterized by having timing requirements in addition to functional requirements. As an example, systems operating in the safety-critical domain often have hard temporal requirements on responding to stimuli from the environment, such as an airbag that must be deployed within a specific timeframe upon collision.

An important aspect of of real-time systems is the Worst Case Execution Time (WCET) of the (real-time) tasks constituting the system. In hard real-time systems (i.e. systems where deadline violations can not be tolerated), it is often insufficient to rely on measuring execution times of the tasks with various inputs. Symbolic execution has been extensively used in the context of WCET analysis [12, 51, 68, 88, 58, 67, 52, 14, 13].

Generally, symbolic execution is used in the field of WCET analysis as the *high-level* analysis that restricts focus to obtaining information about feasibility of program paths. This information is subsequently used in combination with a *low-level* analysis that gathers platform specific information, including behavior of processor-specific features such as caching, pipelines etc. In this combination,

only the feasible paths as determined by symbolic execution are used, allowing higher precision of the analysis result.

An early work that takes this approach is that of [25] that performs timing analysis of SPARK Ada code. The work by [68] uses a similar approach by using cycle-level symbolic execution to integrate path and timing analysis for obtaining tight WCET estimates. Using this technique, the authors were able to perform a perfect WCET estimation for six out of seven test subjects. They also showed that this combination can improve WCET estimation by a factor of twenty when comparing it with a more conservative method that does not prune infeasible paths, but only relies on the structure of the program.

The work of [88] uses symbolic execution to prune infeasible paths in straight-line code (i.e., no loops or recursion)—a commonly found approach to embedded systems development. Imposing such restrictions on the control-flow, guarantees termination for symbolic execution. As with the previous work, the motivation for symbolic execution is to check for feasibility of paths.

The WCET analysis tool, r-TuBound, uses selective symbolic execution [13]. It uses a selective approach to avoid the high computational costs of exhaustive analysis. The symbolic execution is only invoked when the information obtained is limited and when other analysis techniques incorporated in r-TuBound fail.

The work of [67] presents an approach for modeling the real-time tasks of a real-time system written in a variant of the Safety-Critical Java profile. The tool extracts the real-time (periodic and sporadic) tasks and symbolically executes them using Symbolic PathFinder [78]. The paths obtained, are translated into a Network of Timed Automata—the modeling formalism of the UPPAAL model checker—and combined with models of the scheduler. The complete NTA can be used for reasoning about temporal properties that can be expressed in the Timed Computation Tree Logic variant that UPPAAL supports. This includes the *schedulability* of the tasks, i.e. under all different task schedules (taking into account task interactions and sporadically firing tasks), will the system never violate a task deadline? In addition, the tool also supports querying WCETs as well as Best Case Execution Times (BCETs) and response times of the tasks.

4.2 Performance Testing

Symbolic execution effectively enumerates all paths through a program, up to a user-specified bound. It can therefore be used to find performance bottlenecks, e.g. paths that exhibit a large cost with respect to time, memory, power or energy consumption, and so on. By finding a solution for the corresponding path condition, an actual input that triggers this behavior can be generated.

Load testing In *load testing*, a system is analyzed with its behavior under peak loads. Typically, one increases the size of the test input to increase the load on the system. In many cases, however, it is possible to increase the load by carefully choosing input values rather than by increasing the input size. Moreover, when simply increasing the input size, certain program behaviors

may remain undetected. Larger but similarly shaped input may execute the same behavior more often, yet miss other potentially costly behaviors.

Directed incremental symbolic execution is applied by Zhang, Elbaum and Dwyer to automatically generate load tests in [95]. Their approach is *directed* by a cost model, in the sense that it favors more costly paths. It is *incremental* in that it works in phases. It is implemented in a modified version of Symbolic PathFinder.

The user specifies two parameters: the number of test cases to generate and the depth of each phase of symbolic execution. Each phase starts with exhaustive exploration up to the user-specified depth, either from program entry or from a set of locations resulting from the previous phase. Next, the most costly paths are scheduled for further exploration. The number of paths that will be explored further is exactly the number of requested test cases. To increase diversity among paths selected for further exploration, paths are first clustered and in case these do not satisfy a *diversity measure*, further exhaustive exploration of all paths is performed.

Evaluation shows that load tests generated with directed incremental symbolic execution can incite bigger loads, often at smaller input sizes, than human written or randomly generated tests. The approach is also shown to scale up to input sizes of 100MB.

Finding performance bugs In [21], Burnim, Juvekar and Sen apply symbolic execution to find what they call *performance bugs*. Such a bug is said to exist when the complexity of the implementation does not match the theoretical complexity of the implemented algorithm. Their algorithm is called WISE. It uses a clever trick to increase scalability of the analysis, based on the observation that worst-case program behaviors at small input sizes are often good indicators of the worst-case program behavior at larger input sizes.

In a first step, exhaustive exploration at small input sizes is used to construct a *worst-case generator*. Such a generator specifies which paths are likely to lead to the worst case and which are not. For a conditional b in the program, if in the paths leading to the worst case at small input, the same decision (true, false) is always made, it is conjectured that this decision will lead to the worst-case at greater input sizes as well. The generator can then be used at greater input sizes to prune paths that are not likely to lead to the worst case. The paper provides a theoretical guarantee that there is an input size N that is large enough to capture all program behaviors and that, therefore, the generator resulting from exhaustive exploration up to size N accurately predicts the worst-case behavior at any input size $M > N$.

The WISE algorithm is extended in [66] in a tool called SPF-WCA. SPF-WCA generates *guidance policies* which, similarly to worst-case generators, dictate which paths to follow during symbolic execution to discover likely worst-case behaviors. However, the policies are made more expressive by taking into account the history of decisions for each conditional. This means that even though both $true$ and $false$ are seen on the worst-case paths, the generator

can still make a suggestion by looking at patterns in the history of decisions. Precision is also improved by making the policies context-aware, in the sense that only decisions within the same calling context can affect the generator for a conditional. Furthermore, the algorithm in [66] is extended to infer the complexity at any input size, by fitting a function to the results for increasing input sizes. Costs are obtained for input sizes $1 \dots N$, then functions corresponding to common complexity classes are fitted against the results. The application of this work is finding performance related security bugs: if the actual complexity of an algorithm does not match the theoretical complexity, then an adversary can potentially deny service to benign users by sending input that triggers the worst-case complexity. Such inputs can be found by solving the path condition of worst-case paths.

4.3 Security Analysis

Automated Exploit Generation. Automated exploit generation as proposed in [4] uses symbolic execution to find vulnerabilities and to generate working exploits for them. The exploits can redirect control flow to execute injected shell-code, perform a return-to-libc attack, and so on. With the goal being discovering some particular types of exploitable bugs, symbolic execution is used with heuristics based on domain knowledge about different types of bugs. For example, buffer overflow can only occur when an input is copied to a buffer with smaller size, thus the approach uses a light-weight analysis to determine the minimum length k to overwrite any buffer in the program. Performing symbolic execution with the precondition that the input should be at least k significantly prunes off uninteresting input space. Moreover, buffer overflow often occurs at the end of loops, so symbolic execution is customized to give higher priority to the paths that fully exhaust the loop.

Non-interference testing. Undesired flows of information between different sensitivity levels can seriously compromise the security of a system. In a security context, a program can be viewed as a communication channel where information is transmitted from a source `H` to a sink `O`. When `H` contains confidential information and `O` can be observed by public users, information flow from `H` to `O` is not desirable. Traditional information flow analysis considers source and sink as variables of the program: `H` is an input with sensitive data (e.g. a user password), and `O` is the program output. Absence of information flow means the variable `O` is not interfered by the variable `H`, which can be formalized as a *non-interference* policy [30, 45].

A prominent approach to checking non-interference involves self-composition [32, 11], which checks the following Hoare triple on the composition of program `P`:

$$\{\mathtt{L} = \mathtt{L}_1\}\mathtt{P};\mathtt{P}_1\{\mathtt{O} = \mathtt{O}_1\}$$

Here `L` is the public input of the program `P`, and $\mathtt{P}_1$ is a copy of `P` where `L` and `O` are renamed to $\mathtt{L}_1$ and $\mathtt{O}_1$, respectively. The program `P` satisfies non-interference

if when executing the sequential composition of $\mathtt{P}$ and $\mathtt{P}_1$ with the precondition $\mathtt{L} = \mathtt{L}_1$, after the execution, the postcondition $\mathtt{O} = \mathtt{O}_1$ holds.

Symbolic execution was used for checking self-composition as described in [72]. This approach assumes that the program P can be fully explored to obtain the set of all symbolic paths, and uses path manipulation to avoid the cost of executing the self-composed program. This work is later generalized in [36], which releases the assumption, and handles recursions and unbounded loops using user-defined loop invariants and method contracts.

Balliu *et al.*[9] took a different approach and formalized non-interference using an epistemic logic. Formulae in this logic are then checked using an algorithm based on symbolic execution, implemented on top of Symbolic PathFinder.

Quantitative information flow analysis. Non-interference is often overly pessimistic and in practice unachievable. To illustrate consider a password checking program whose public output, which rejects or accepts a user-provided input, depends on the value of the password. Such a program does not satisfy non-interference and it leaks a small amount of information, i.e. if the input matches the secret password or not. The program will eventually leak the whole password if the adversary is given enough attempts. However, with a strong password the amount of leaked information is too small, and the program is considered to be secure.

To address the limitation above, quantitative methods for information flow [29, 69] have been developed, which, instead of enforcing zero interference, measure interference. We use the two terms "*interference*" and "*information flow*" interchangeably, since O is interfered by H if there is information flow from H to O. Thus programs with "*small*" interference can be accepted as secure. Information leakage is measured using information theory metrics [31] such as Shannon entropy, Rényi's min-entropy and channel capacity.

To compute channel capacity, i.e. maximum amount of information leakage, symbolic quantitative information flow (SQIF) [76, 74] adds conditions to test every bit of the output O, and uses symbolic execution to explore all possible values of O. Using bitvector solvers, SQIF can perform quantitative information flow analysis over programs with non-linear constraints.

Instead of using symbolic execution to enumerate values of the output O, and thus compute channel capacity, QILURA [75] uses symbolic execution to partition the input space, and counts (using an off-the shelf model counting tool) the blocks in the partition that lead to leakage of information. By delegating the counting process to Latte [2], a model counter for systems of linear integer inequalities, QILURA achieves significant improvement in performance compared to SQIF. However, by using Latte, it can only analyze programs with linear constraints, and by counting the input, it only returns an upper bound on channel capacity.

Side-channel analysis. Side channels allow an attacker to infer information about a secret by observing non-functional characteristics of a program, such

as execution time or memory consumed. Recall that a program can be viewed as a communication channel where information is transmitted from a source `H` to a sink `O`. For side-channel analysis, the sink `O` is not necessary an output variable but rather a non-functional characteristic of program execution, such as running time, power consumption, number of memory accessed or packets transmitted over a network. Side-channel attacks [59, 53, 19, 26] have been used successfully to uncover secret information in a variety of applications, including web applications and cryptographic systems.

In previous work [77, 10, 73], we have studied the use of symbolic execution for side channel analysis. Different from SQIF and QILURA, in this line of work we compute Shannon entropy of the leakage and we tackle the problem of multi-run attacks, that is we consider scenarios when an adversary can execute the program multiple times with different and *gradually* uncover a secret. Solving this problem is difficult, since quantifying leakage for a weak or random single-runattack could not provide a guarantee for all possible attacks, and thus we need to synthesize optimal attacks.

4.4 Symbolic Execution and Fuzzing

An idea that has been shown to be particularly promising in recent years is the combination of symbolic execution with other testing techniques that are less expensive, but also are limited in their ability of achieving a high coverage of program paths. Symbolic execution is then invoked on demand, to increase coverage.

Particularly promising is the combination of symbolic execution and fuzzing. Fuzzing is an automated testing technique that has been used successfully to discover security vulnerabilities and other bugs in software [70, 90]. In its simplest, black-box, form, a program is run on randomly generated or mutated inputs, in search of cases where the program crashes or hangs. More advanced techniques may take input formatting into account, e.g. in the form of grammars, or leverage program instrumentation or program analysis to gather information about the program paths exercised by the inputs, in order to increase coverage.

Fuzzing has shown to be very effective at finding security vulnerabilities in practice. For instance, the popular fuzzing tool AFL [94] was instrumental in finding several of the Stagefright vulnerabilities in Android, as well as numerous bugs in (security-critical) applications and libraries such as Bash, BIND, OpenSSL, OpenSSH, GnuTLS, GnuPG, PHP, Apache, IJG jpeg, libjpeg-turbo and many more. The work in [54] presents Kelinci, an AFL-based fuzzer for Java that found similar vulnerabilities in Apache Commons Imaging and OpenJDK 9 .

Fuzzing has its limitations. As inputs are tested randomly, every input value has the same probability of getting tested and code coverage is generally low. Consider, for example, the code in Listing 1. This function has a bug when the value of its input is exactly 1234. The chance of randomly testing this input is only 1 in 2^{32}.

```
void ex(int x) {
    if (x == 1234)
        abort();
}
```

Listing 1: Function that is problematic for fuzzing

This is exactly the type of problem that symbolic execution is good at. It will easily find and solve the constraint $x = 1234$ leading to new behavior. The techniques described in this sections leverage these complementary strengths of fuzz testing and symbolic execution.

EvoSuite In [39] Galeotti *et al.* observed that if there is a change in fitness after a mutation on a primitive value, then the variable this value is assigned to is important. Thus they use dynamic symbolic execution with this variable being symbolic, to derive new values for it. On the other hand, if there is no change in fitness after a mutation or the changes in fitness are not related to a primitive value, then their adaptive algorithm does not apply dynamic symbolic execution. This approach is embodied in the EvoSuite tool.

SAGE and Project Springfield A promising approach that combines symbolic execution with fuzzing is implemented in the SAGE tool which has been continued with Project Springfield. SAGE (Scalable Automated Guided Execution) [43] extends DART with a directed search algorithm. Instead of negating only the final condition of a complete symbolic execution, this *generational search* negates all conditions on the path (in conjunction with the path condition for the path leading up to them). This results in a large number of new test inputs, instead of just one. SAGE is used extensively at Microsoft and has been very successful at finding security-related bugs. Out of all bugs discovered in Windows 7, approximately one in three was found using SAGE. This is notable, as it was the last tool applied, so none of these bugs were found by other tools [43]. Microsoft is currently in the process of making SAGE available to the public as a cloud service under the name Project Springfield[1].

Driller Driller [89] is another promising tool that combines the AFL fuzzer with the angr symbolic execution engine. AFL is a security-oriented grey-box fuzzer that employs compile-time instrumentation and genetic algorithms to automatically discover test cases that trigger new internal states in C programs, improving the functional coverage for the fuzzed code. Driller is based on the idea that software consists of different *compartments*. Within a compartment, decisions are fairly uniformly distributed and, as such, fuzzing works very well. Jumps between compartments, however, may be less trivial for a fuzzer to detect. For instance, an application may expect a certain file header that is essentially a magic number as in Listing 1. To ensure progress, Driller invokes the symbolic execution engine whenever the fuzzer appears to be "stuck". It symbolically traces the program for all inputs that AFL found "interesting",

[1] https://www.microsoft.com/en-us/security-risk-detection/

then finds decisions that have unexplored branches and invokes a solver to generate inputs that drive execution down that branch. As this is expected to help crossover to new program compartments, fuzzing continues from these generated inputs. DRILLER was evaluated on 126 applications released in the qualifying event of the DARPA Cyber Grand Challenge. It identified the same number of vulnerabilities, in a similar time-frame, as the tool that performed best at the event.

Mayhem MAYHEM [24] is a symbolic execution engine that aims to find security vulnerabilities in binaries. It has a strong focus on the ubiquitous buffer overflow, and other memory-related vulnerabilities. MAYHEM augments path constraints with additional, security-related information such as if a user can load their own code into memory. If such an augmented path condition is satisfiable, then the program is vulnerable. To be able to capture such security-related properties, MAYHEM uses an index-based memory model to allow using symbolic values to point to memory locations.

It also uses a combination of dynamic symbolic execution and traditional symbolic exection, which is referred to as *hybrid* symbolic execution. A *Concrete Executor Client* (CEC) explores paths concretely. However, it does keep track of which inputs are considered symbolic and performs a dynamic taint analysis. When a basic block is reached that contains tainted instructions, it is passed to the *Symbolic Executor Server* (SES) that is running in parallel. After symbolic execution, the SES instructs the CEC on a particular path to execute. When memory is strained, MAYHEM threads can store their state to be efficiently restarted later.

The tool has been combined with a fuzzer (MURPHY) and in 2016, it won the DARPA Cyber Grand Challenge, in which 7 autonomous computer systems competed live in a search for security vulnerabilities[2].

5 Conclusion

In this chapter we reviewed symbolic execution techniques and tools and we described recent applications, including finding worst-case execution time in programs, load testing and security analysis, via combinations of symbolic execution with fuzzing. There are other promising directions for symbolic execution, among them the extension of symbolic execution to probabilistic reasoning [15, 37], with applications to reliability analysis and quantitative information flow (which we described briefly in this chapter). An in-depth review of those techniques is left for the future.

Symbolic execution is increasingly used not only in academic settings but also in industry, e.g. in Microsoft, NASA, IBM and Fujitsu, and even at the Pentagon [1]. Many symbolic execution engines have been built targeting different programming languages and architectures. This trend is expected to intensify

[2] `https://www.darpa.mil/news-events/2016-08-04`

in the future. Symbolic execution in a distributed setting, leveraging cloud technology, such as Cloud9 [28], SAGE [43], and MergePoint [5], is expected to further extend the applicability of the technique in practice.

References

[1] `https://www.cyberscoop.com/mayhem-darpa-cyber-grand-challenge-dod-voltron/`.

[2] LattE. `http://www.math.ucdavis.edu/~latte/`.

[3] S. Anand, P. Godefroid, and N. Tillmann. Demand-driven Compositional Symbolic Execution. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 367–381, Berlin, Heidelberg, 2008. Springer-Verlag.

[4] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley. Automatic Exploit Generation. *Commun. ACM*, 57(2):74–84, Feb. 2014.

[5] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. Enhancing Symbolic Execution with Veritesting. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1083–1094, New York, NY, USA, 2014. ACM.

[6] A. Aydin, L. Bang, and T. Bultan. *Automata-Based Model Counting for String Constraints*, pages 255–272. Springer International Publishing, Cham, 2015.

[7] D. Babić, L. Martignoni, S. McCamant, and D. Song. Statically-directed Dynamic Automated Test Generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 12–22, New York, NY, USA, 2011. ACM.

[8] T. Ball and J. Daniel. Deconstructing Dynamic Symbolic Execution. Technical report, Jan. 2015.

[9] M. Balliu, M. Dam, and G. L. Guernic. ENCoVer: Symbolic Exploration for Information Flow Security. In *Proceedings of the 2012 IEEE 25th Computer Security Foundations Symposium*, CSF '12, pages 30–44, Washington, DC, USA, 2012. IEEE Computer Society.

[10] L. Bang, A. Aydin, Q.-S. Phan, C. S. Păsăreanu, and T. Bultan. String Analysis for Side Channels with Segmented Oracles. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 193–204, New York, NY, USA, Nov. 2016. ACM.

[11] G. Barthe, P. R. D'Argenio, and T. Rezk. Secure Information Flow by Self-Composition. In *Proceedings of the 17th IEEE workshop on Computer Security Foundations*, CSFW '04, Washington, DC, USA, 2004. IEEE Computer Society.

[12] B. Benhamamouch, B. Monsuez, and F. Védrine. Computing WCET Using Symbolic Execution. In *Proceedings of the Second International Conference on Verification and Evaluation of Computer and Communication Systems*, VECoS'08, pages 128–139, Swinton, UK, UK, 2008. British Computer Society.

[13] A. Biere, J. Knoop, L. Kovács, and J. Zwirchmayr. The Auspicious Couple: Symbolic Execution and WCET Analysis. In C. Maiza, editor, *13th International Workshop on Worst-Case Execution Time Analysis*, volume 30 of *OpenAccess Series in Informatics (OASIcs)*, pages 53–63, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[14] R. Bodík, R. Gupta, and M. L. Soffa. Refining Data Flow Information Using Infeasible Paths. *SIGSOFT Softw. Eng. Notes*, 22(6):361–377, Nov. 1997.

[15] M. Borges, A. Filieri, M. d'Amorim, C. S. Pasareanu, and W. Visser. Compositional solution space quantification for probabilistic software analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 123–132, 2014.

[16] P. Braione, G. Denaro, and M. Pezzè. Symbolic Execution of Programs with Heap Inputs. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 602–613, New York, NY, USA, 2015. ACM.

[17] P. Braione, G. Denaro, and M. Pezzè. JBSE: A Symbolic Executor for Java Programs with Complex Heap Inputs. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 1018–1022, New York, NY, USA, 2016. ACM.

[18] T. Brennan, N. Tsiskaridze, N. Rosner, A. Aydin, and T. Bultan. Constraint Normalization and Parameterized Caching for Quantitative Program Analysis. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 535–546, New York, NY, USA, 2017. ACM.

[19] D. Brumley and D. Boneh. Remote Timing Attacks Are Practical. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 1–1, Berkeley, CA, USA, 2003. USENIX Association.

[20] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. *BAP: A Binary Analysis Platform*, pages 463–469. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[21] J. Burnim, S. Juvekar, and K. Sen. WISE: Automated test generation for worst-case complexity. In *2009 IEEE 31st International Conference on Software Engineering*, pages 463–473, May 2009.

[22] C. Cadar, D. Dunbar, and D. 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.

[23] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically Generating Inputs of Death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 322–335, New York, NY, USA, 2006. ACM.

[24] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing Mayhem on Binary Code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 380–394, Washington, DC, USA, 2012. IEEE Computer Society.

[25] R. Chapman, A. Burns, and A. Wellings. Integrated program proof and worst-case timing analysis of SPARK Ada. In *In Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'94*. ACM Press, 1994.

[26] S. Chen, R. Wang, X. Wang, and K. Zhang. Side-Channel Leaks in Web Applications: A Reality Today, a Challenge Tomorrow. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 191–206, Washington, DC, USA, 2010. IEEE Computer Society.

[27] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A Platform for In-vivo Multi-path Analysis of Software Systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 265–278, New York, NY, USA, 2011. ACM.

[28] L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, and G. Candea. Cloud9: A Software Testing Service. *SIGOPS Oper. Syst. Rev.*, 43(4):5–10, Jan. 2010.

[29] D. Clark, S. Hunt, and P. Malacaria. A static analysis for quantifying information flow in a simple imperative language. *J. Comput. Secur.*, 15(3):321–371, Aug. 2007.

[30] E. S. Cohen. Information Transmission in Sequential Programs. In R. A. DeMillo, D. P. Dobkin, A. K. Jones, and R. J. Lipton, editors, *Foundations of Secure Computation*, pages 297–335. Academic Press, 1978.

[31] T. M. Cover and J. A. Thomas. *Elements of information theory*. Wiley-Interscience, New York, NY, USA, 1991.

[32] A. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In *Proceedings of the Second international conference on Security in Pervasive Computing*, SPC'05, pages 193–209, Berlin, Heidelberg, 2005. Springer-Verlag.

[33] L. De Moura and N. Bjørner. Z3: an efficient SMT solver. In *Proceedings of the 14^{th} international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[34] X. Deng, J. Lee, and Robby. Bogor/Kiasan: A K-bounded Symbolic Execution for Checking Strong Heap Properties of Open Systems. In *Proceedings of the 21^{st} IEEE/ACM International Conference on Automated Software Engineering*, ASE '06, pages 157–166, Washington, DC, USA, 2006. IEEE Computer Society.

[35] X. Deng, Robby, and J. Hatcliff. Towards A Case-Optimal Symbolic Execution Algorithm for Analyzing Strong Properties of Object-Oriented Programs. In *Proceedings of the Fifth IEEE International Conference on Software Engineering and Formal Methods*, SEFM '07, pages 273–282, Washington, DC, USA, 2007. IEEE Computer Society.

[36] Q. H. Do, R. Bubel, and R. Hähnle. Exploit Generation for Information Flow Leaks in Object-Oriented Programs. In *ICT Systems Security and Privacy Protection: 30^{th} IFIP TC 11 International Conference, SEC 2015, Hamburg, Germany, May 26-28, 2015, Proceedings*, pages 401–415, Cham, 2015. Springer International Publishing.

[37] A. Filieri, C. S. Pasareanu, and W. Visser. Reliability analysis in symbolic pathfinder. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 622–631, 2013.

[38] A. Filieri, C. S. Păsăreanu, W. Visser, and J. Geldenhuys. Statistical Symbolic Execution with Informed Sampling. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 437–448, New York, NY, USA, 2014. ACM.

[39] J. P. Galeotti, G. Fraser, and A. Arcuri. Improving search-based test suite generation with dynamic symbolic execution. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, pages 360–369, Nov 2013.

[40] J. Geldenhuys, N. Aguirre, M. F. Frias, and W. Visser. Bounded Lazy Initialization. In G. Brat, N. Rungta, and A. Venet, editors, *NASA Formal Methods, 5^{th} International Symposium, NFM 2013, Moffett Field, CA, USA, May 14-16, 2013. Proceedings*, pages 229–243, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[41] P. Godefroid. Compositional Dynamic Test Generation. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pages 47–54, New York, NY, USA, 2007. ACM.

[42] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. PLDI '05, pages 213–223. ACM, 2005.

[43] P. Godefroid, M. Y. Levin, and D. Molnar. SAGE: Whitebox Fuzzing for Security Testing. *Queue*, 10(1):20:20–20:27, Jan. 2012.

[44] P. Godefroid and D. Luchaup. Automatic partial loop summarization in dynamic test generation. In *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*, pages 23–33, 2011.

[45] J. A. Goguen and J. Meseguer. Security Policies and Security Models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[46] E. L. Gunter and D. Peled. Unit Checking: Symbolic Model Checking for a Unit of Code. In N. Dershowitz, editor, *Verification: Theory and Practice, Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday*, volume 2772 of *Lecture Notes in Computer Science*, pages 548–567. Springer, 2003.

[47] B. Hillery, E. Mercer, N. Rungta, and S. Person. Exact Heap Summaries for Symbolic Execution. In *Proceedings of the 17th International Conference on Verification, Model Checking, and Abstract Interpretation - Volume 9583*, VMCAI 2016, pages 206–225, New York, NY, USA, 2016. Springer-Verlag New York, Inc.

[48] S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '01, pages 14–26, New York, NY, USA, 2001. ACM.

[49] J. Jaffar, V. Murali, and J. A. Navas. Boosting concolic testing via interpolation. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 48–58, 2013.

[50] J. Jeon, K. K. Micinski, and J. S. Foster. SymDroid: Symbolic execution for Dalvik bytecode. Technical report, 2012.

[51] D. Kebbal and P. Sainrat. Combining Symbolic Execution and Path Enumeration in Worst-Case Execution Time Analysis. In F. Mueller, editor, *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)*, volume 4 of *OpenAccess Series in Informatics (OASIcs)*,

Dagstuhl, Germany, 2006. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[52] D. Kebbal and P. Sainrat. Combining Symbolic Execution and Path Enumeration in Worst-Case Execution Time Analysis. In F. Mueller, editor, *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)*, volume 4 of *OpenAccess Series in Informatics (OASIcs)*, Dagstuhl, Germany, 2006. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[53] J. Kelsey. Compression and Information Leakage of Plaintext. In *Revised Papers from the 9^{th} International Workshop on Fast Software Encryption*, FSE '02, pages 263–276, London, UK, UK, 2002. Springer-Verlag.

[54] R. Kersten, K. Luckow, and C. S. Păsăreanu. Poster: Afl-based fuzzing for java with kelinci. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 2511–2513, New York, NY, USA, 2017. ACM.

[55] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proceedings of the 9^{th} international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'03, pages 553–568, Berlin, Heidelberg, 2003. Springer-Verlag.

[56] A. King. Distributed parallel symbolic execution. In *Master Thesis, Kansas State University*, 2009.

[57] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[58] J. Knoop, L. Kovács, and J. Zwirchmayr. WCET Squeezing: On-demand Feasibility Refinement for Proven Precise WCET-bounds. In *Proceedings of the 21^{st} International Conference on Real-Time Networks and Systems*, RTNS '13, pages 161–170, New York, NY, USA, 2013. ACM.

[59] P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Proceedings of the 16^{th} Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '96, pages 104–113, London, UK, UK, 1996. Springer-Verlag.

[60] B. Korel. A dynamic approach of test data generation. In *Proceedings. Conference on Software Maintenance 1990*, pages 311–317, Nov 1990.

[61] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. Efficient State Merging in Symbolic Execution. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 193–204, New York, NY, USA, 2012. ACM.

[62] Q. L. Le, J. Sun, and W.-N. Chin. *Satisfiability Modulo Heap-Based Programs*, pages 382–404. Springer International Publishing, Cham, 2016.

[63] G. Li, I. Ghosh, and S. P. Rajan. KLOVER: a symbolic execution and automatic test generation tool for C++ programs. In *Proceedings of the 23rd international conference on Computer aided verification*, CAV'11, pages 609–615, Berlin, Heidelberg, 2011. Springer-Verlag.

[64] D. Liew, C. Cadar, and A. F. Donaldson. Symbooglix: A Symbolic Execution Engine for Boogie Programs. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 45–56, Apr. 2016.

[65] K. Luckow, M. Dimjašević, D. Giannakopoulou, F. Howar, M. Isberner, T. Kahsai, Z. Rakamarić, and V. Raman. JDart: A Dynamic Symbolic Analysis Framework. In M. Chechik and J.-F. Raskin, editors, *Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 9636 of *Lecture Notes in Computer Science*, pages 442–459. Springer, 2016.

[66] K. Luckow, R. Kersten, and C. Pasareanu. Symbolic Complexity Analysis using Context-preserving Histories. In *Proceedings of the 10th IEEE International Conference on Software Testing, Verification and Validation (ICST 2017)*, 2017. To appear.

[67] K. S. Luckow, C. S. Păsăreanu, and B. Thomsen. Symbolic execution and timed automata model checking for timing analysis of Java real-time systems. *EURASIP Journal on Embedded Systems*, 2015(1):2, 2015.

[68] T. Lundqvist and P. Stenström. An Integrated Path and Timing Analysis Method Based on Cycle-Level Symbolic Execution. *Real-Time Systems*, 17(2-3):183–207, Dec. 1999.

[69] P. Malacaria. Assessing security threats of looping constructs. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '07, pages 225–235, New York, NY, USA, 2007. ACM.

[70] B. P. Miller, L. Fredriksen, and B. So. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM*, 33(12):32–44, Dec. 1990.

[71] L. H. Pham, Q. L. Le, Q.-S. Phan, J. Sun, and S. Qin. Enhancing Symbolic Execution of Heap-based Programs with Separation Logic for Test Input Generation. *CoRR*, abs/1712.06025, 2017.

[72] Q.-S. Phan. Self-composition by Symbolic Execution. In *2013 Imperial College Computing Student Workshop*, volume 35 of *OpenAccess Series in Informatics (OASIcs)*, pages 95–102, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[73] Q.-S. Phan, L. Bang, C. S. Păsăreanu, P. Malacaria, and T. Bultan. Synthesis of Adaptive Side-Channel Attacks. In *2017 IEEE 30th Computer*

Security Foundations Symposium (CSF), CSF '17, Washington, DC, USA, Aug. 2017. IEEE Computer Society.

[74] Q.-S. Phan and P. Malacaria. Abstract Model Counting: A Novel Approach for Quantification of Information Leaks. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '14, pages 283–292, New York, NY, USA, 2014. ACM.

[75] Q.-S. Phan, P. Malacaria, C. S. Păsăreanu, and M. d'Amorim. Quantifying Information Leaks Using Reliability Analysis. In *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software*, SPIN 2014, pages 105–108, New York, NY, USA, 2014. ACM.

[76] Q.-S. Phan, P. Malacaria, O. Tkachuk, and C. S. Păsăreanu. Symbolic Quantitative Information Flow. *SIGSOFT Softw. Eng. Notes*, 37(6):1–5, Nov. 2012.

[77] C. S. Păsăreanu, Q.-S. Phan, and P. Malacaria. Multi-run Side-Channel Analysis Using Symbolic Execution and Max-SMT. In *Proceedings of the 2016 IEEE 29th Computer Security Foundations Symposium*, CSF '16, pages 387–400, Washington, DC, USA, June 2016. IEEE Computer Society.

[78] C. S. Păsăreanu, W. Visser, D. Bushnell, J. Geldenhuys, P. Mehlitz, and N. Rungta. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. *Automated Software Engineering*, pages 1–35, 2013.

[79] R. Qiu, G. Yang, C. S. Pasareanu, and S. Khurshid. Compositional Symbolic Execution with Memoized Replay. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 632–642, 2015.

[80] J. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*, pages 55–74, 2002.

[81] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A Symbolic Execution Framework for JavaScript. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 513–528, Washington, DC, USA, 2010. IEEE Computer Society.

[82] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *Proceedings of the 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[83] K. Sen, G. Necula, L. Gong, and W. Choi. MultiSE: Multi-path Symbolic Execution Using Value Summaries. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 842–853, New York, NY, USA, 2015. ACM.

[84] J. Siddiqui and S. Khurshid. ParSym: Parallel symbolic execution. In *ICSTE*, volume 1, pages V1–405–V1–409, 2010.

[85] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. *BitBlaze: A New Approach to Computer Security via Binary Analysis*, pages 1–25. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[86] M. Souza, M. Borges, M. d'Amorim, and C. S. Păsăreanu. CORAL: Solving Complex Constraints for Symbolic Pathfinder. In *Proceedings of the Third International Conference on NASA Formal Methods*, NFM'11, pages 359–374, Berlin, Heidelberg, 2011. Springer-Verlag.

[87] M. Staats and C. Păsăreanu. Parallel symbolic execution for structural test generation. ISSTA '10, pages 183–194, New York, NY, USA, 2010. ACM.

[88] F. Stappert and P. Altenbernd. Complete worst-case execution time analysis of straight-line hard real-time programs. *Journal of Systems Architecture*, 46(4):339 – 355, 2000.

[89] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *23nd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*, 2016.

[90] M. Sutton, A. Greene, and P. Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[91] W. Visser, J. Geldenhuys, and M. B. Dwyer. Green: reducing, reusing and recycling constraints in program analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 58:1–58:11, New York, NY, USA, 2012. ACM.

[92] W. Visser, C. S. Păsăreanu, and S. 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.

[93] G. Yang, C. S. Păsăreanu, and S. Khurshid. Memoized Symbolic Execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 144–154, New York, NY, USA, 2012. ACM.

[94] M. Zalewski. American Fuzzy Lop (AFL). `http://lcamtuf.coredump.cx/afl/`, 2017. Accessed August 11, 2017.

[95] P. Zhang, S. Elbaum, and M. B. Dwyer. Automatic Generation of Load Tests. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 43–52, Washington, DC, USA, 2011. IEEE Computer Society.

[96] Y. Zheng, X. Zhang, and V. Ganesh. Z3-str: A Z3-based String Solver for Web Application Analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 114–124, New York, NY, USA, 2013. ACM.