Automated Software Test Generation: Some Challenges, Solutions, and Recent Advances

George Candea[1] and Patrice Godefroid[2(✉)]

[1] Ecole Polytechnique Fédérale de Lausanne (EPFL), 1015 Lausanne, Switzerland
[2] Microsoft Research, One Microsoft Way, Redmond, WA 98052, USA
pg@microsoft.com

Abstract. The automation of software testing promises to delegate to machines what is otherwise the most labor-intensive and expensive part of software development. The past decade has seen a resurgence in research interest for this problem, bringing about significant progress. In this article, we provide an overview of automated test generation for software, and then discuss recent developments that have had significant impact on real-life software.

Keywords: Software testing · Program analysis · Symbolic execution

1 Introduction

Software testing is generally used to assess the quality of a program, where "quality" can mean reliability, performance, usability, compliance, etc. depending on context. The purpose of this assessment can be to debug the program, because testing points out programming errors and ways to reproduce the program failure they induce. Another purpose can be assessing whether the program is acceptable to a client, because tests can reveal not only bugs but also gaps between what the client wanted and what the developer thought she wanted. All in all, software testing is a method for critiquing a program rather than demonstrating its correctness. In Dijkstra's words, "testing shows the presence, not the absence of bugs" [45].

Testing a program consists of executing the program with a given set of inputs and observing its behavior. *Test automation* entails running several tests in an automated fashion, such as every night or every time a major change is made to the program. Such automation requires one or more *test cases*, each consisting of specific inputs to the program, and an automated means of validating the outcome, often called a *test oracle*. Test cases can be written in a black-box manner (where the test developer chooses scenarios solely based on knowledge of what the program is supposed to do) or a white-box manner (where internal knowledge of the program source code supplements the external knowledge in choosing test scenarios). There exist many types of testing (e.g., unit, feature,

B. Steffen and G. Woeginger (Eds.): Computing and Software Science, LNCS 10000, pp. 1–27, 2018.
https://doi.org/10.1007/978-3-319-91908-9_24

functional, system, regression) that fulfill various goals. Testing is complementary to other non-dynamic methods for checking program correctness, such as visual code inspection or static program analysis.

The outcome of running tests is measured in various ways. For example, counting how many tests succeed vs. fail is often a proxy metric for the program's code quality. Another example is test coverage, which measures the rigorousness of testing, e.g., by computing what fraction of the program's instructions were executed during a test suite.

The fundamental challenge in thoroughly testing a program is that the number of possible inputs is large. For example, consider a program that takes four 64-bit integers and adds them up: there exist 2^{256} different combinations of integers that could be provided to this program. In contrast, scientists estimate that the observable Universe has on the order of 2^{240} atoms [115]. A naive approach of trying all inputs one by one would therefore not complete, so one must be clever about picking test inputs.

This brings us to *test generation*, i.e., producing "interesting" inputs to test software. There is an inherent trade-off between how long it takes to choose the inputs vs. how long it takes to run the corresponding test and measure outcomes. For example, if executing a program takes a long time, it makes sense to spend time on smartly choosing inputs, so as to minimize the number of times the program must execute during testing. However, if running the program is quick, then taking a long time to choose inputs can be detrimental compared to running the program many times with many inputs.

A good way to reduce the time spent choosing test inputs is to automate the process. This observation gave rise to the field of *automated test generation*, which is the subject of this article. We discuss the spectrum of techniques used for automated test generation (Sect. 2) along with some of the challenges they face when applied in practice (Sect. 3). We highlight two recent advances in the engineering and application of these techniques: SAGE (Sect. 4) and S2E (Sect. 5). While this article is by no means a survey of automated test generation techniques, we do mention many approaches, techniques and tools throughout, as well as in Sect. 6. We conclude with a few thoughts on future directions in this field of research (Sect. 7).

2 Automated Test Generation: An Overview

Techniques for automatically generating test inputs lay along a spectrum that has at one end blackbox random testing (Sect. 2.1) and at the other end whitebox symbolic execution (Sect. 2.2). We now describe these two end-points.

2.1 Random Testing and Input Fuzzing

Perhaps the simplest form of automated test generation is to select program inputs at random [92]. Various input probability distributions can be used, either

uniform or biased towards some specific values believed to lead to interesting corner cases, like 0, −1 or `MAXINT` for integer input values.

A more evolved form, called *fuzz testing*, consists of starting with well-formed inputs and repeatedly modifying them, more or less at random, to produce new inputs—this preserves the benefit of blackbox testing while increasing the probability of inputs found in this way being "interesting" (e.g., capable of getting past the first layer of input parsing). This proved to be an effective way to find crashes and security vulnerabilities in software, with some of the most notorious security vulnerabilities having been found this way [27]. Fuzz testing (or "fuzzing" for short) has become a standard fixture in most commercial software testing strategies [1,75,91].

Key to the effectiveness of fuzzing is test *quantity*, i.e., the ability to generate and try out new inputs at high rate. Large-scale fuzzing efforts (such as ClusterFuzz [5] that tests the Chromium web browser) test software round the clock, using as many machines as are available. The number of bugs found is limited by the number of CPU resources given to the fuzzer—intuition suggests that the more tests the fuzzer gets to run, the more likely it is to find bugs.

The other key ingredient is test *quality*. First, testing many random inputs in a blackbox fashion can at best discover shallow bugs, whereas picking inputs smartly can penetrate deeper into the program code and reduce the number of executions needed to find a bug. To improve the quality of generated inputs, modern fuzzers use feedback from prior executions to steer input generation toward those inputs that are more likely to uncover bugs. Second, detecting anomalous behaviors automatically during the test runs increases the chances of detecting the manifestation of a bug. Therefore, fuzzers check for a wide range of "illegal behaviors," with memory safety violations being the most popular. The premise is that higher-quality tests are more likely to find bugs.

Most modern blackbox fuzzers, like AFL [119] or LibFuzzer [84], have moved away from the initial "random testing" end-point of the spectrum: they operate in a feedback loop, as shown in Fig. 1. They rely on instrumentation to detect program features triggered by tests, e.g., a basic block being executed or a buffer overflow. Whenever a feature is seen for the first time, the fuzzer reacts: it adds the test to its corpus of interesting testcases, or reports a bug.

Different types of instrumentation can detect various features of interest. For example, *coverage bits* detect when a particular edge in the control-flow graph of the program is executed. When a coverage bit fires, the fuzzer knows that it has found new code. A *coverage counter* similarly detects how often an edge has been executed, and can signal to the fuzzer that it made progress when exploring a loop. *Safety checks* detect abnormal conditions, alerting the fuzzer that a bug has been found. Such checks can be either added by developers in the form of assertions, or done automatically by tools such as Purify, AppVerifier, Valgrind, UndefinedBehaviorSanitizer, ThreadSanitizer, AddressSanitizer, `FORTIFY_SOURCE`, AFL's `libdislocator`, stack-protector, and others [34,44,52,78,108,109]. Using safety checks increases the quality of tests, and thus the number of bugs that fuzzers can detect. Without them, developers

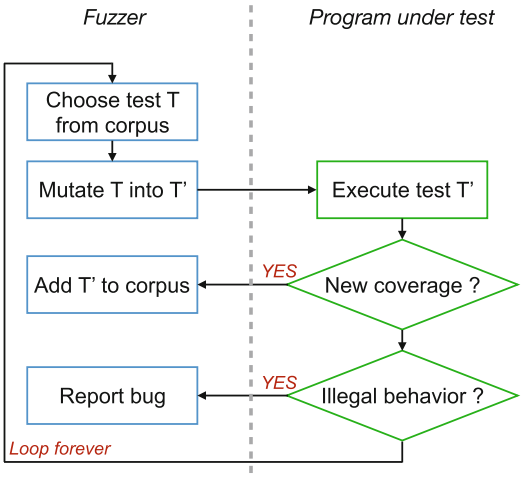

Fig. 1. Typical workflow of a fuzzer driven by coverage feedback

need to hope that illegal behavior leads to a segmentation fault or other visible exception. This does not always happen, particularly for tricky cases such as use-after-free or buffer over-read bugs.

Ideally, a fuzzer should simultaneously have high test throughput and high test quality, but unfortunately these two requirements conflict: obtaining good feedback comes at the cost of throughput. Both detecting program misbehavior and collecting code coverage information is done using program instrumentation, which competes for CPU cycles with the actual instructions of the program being tested. It is in fact not unusual for fuzzers to invest less than half of their resources into executing code of the target program, and spend the rest on improving test quality.

2.2 Test Generation with Symbolic Execution

At the other end of the spectrum, the most precise form of automatic code-driven test generation known today is dynamic test generation with symbolic execution.

Symbolic execution is a program analysis technique that was introduced in the 70s (e.g., see [14,37,76,79,103]). Symbolic execution means executing a program with symbolic rather than concrete values. Assignment statements are represented as functions of their (symbolic) arguments, while conditional statements are expressed as constraints on symbolic values. Symbolic execution can be used for many purposes, such as bug detection, program verification, debugging, maintenance, and fault localization [38].

Symbolic execution can be used to symbolically explore the tree of all computations the program exhibits when all possible value assignments to input parameters are considered [79]. As an example, consider the simple program on the left of

Fig. 2 and its computation tree on the right. This program takes an integer value *rpm* as input. The set of possible values for program variable *rpm* is represented by a *symbolic* value λ that can initially take on any integer value: this is represented by the constraint $\lambda \in Z$. During symbolic execution of that program, whenever a branch depending on λ is encountered, a new constaint is generated to capture how to make that input-dependent branch condition evaluate to true (e.g., $\lambda > 1000$) or false (e.g., $\lambda \leq 1000$) respectively. By repeating this process going down the tree, we obtain an execution tree annotated with conjunctions of input constraints which characterize what input values are required in order to reach what parts of the program. Those conjunctions of constraints are called *path constraints*, or *path conditions*, and are shown in grey on the right of Fig. 2.

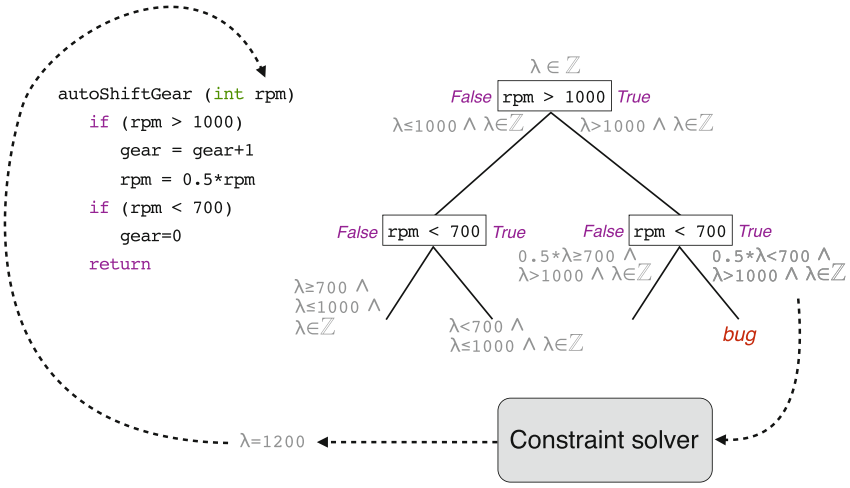

Fig. 2. Test generation for a simple program using symbolic execution.

In other words, for each *control path* p, that is, a sequence of control locations of the program, a *path constraint* ϕ_p is constructed that characterizes the input assignments for which the program executes along p. All the paths can be enumerated by a search algorithm that explores all possible branches at conditional statements. The paths p for which ϕ_p is satisfiable are *feasible* and are the only ones that can be executed by the actual program. The solutions to ϕ_p characterize the inputs that drive the program through p. This characterization is *exact* provided symbolic execution has *perfect precision*. Assuming that the theorem prover used to check the satisfiability of all formulas ϕ_p is sound and complete, this analysis amounts to a kind of exhaustive symbolic testing of all feasible control paths of a program.

Work on automatic code-driven test generation using symbolic execution can roughly be partitioned into two groups: *static* versus *dynamic* test generation. Static test generation (e.g., [79]) consists of analyzing a program P statically, by using symbolic execution techniques to attempt to compute inputs to drive P

along specific execution paths or branches, *without ever executing the program.* In contrast, dynamic test generation (e.g., [22,23,31,62,68,80,98,107]) consists of executing the program P starting with some concrete inputs, while performing symbolic execution *dynamically*, collecting symbolic constraints on inputs gathered from predicates in branch statements along the execution, and then using a constraint solver to infer variants of the previous inputs in order to steer the next execution of the program towards an alternative program branch.

The key practical advantage of dynamic test generation compared to static test generation is that the entire program does not need to be executed symbolically for test generation. Imprecision in dynamic symbolic execution can easily be alleviated using concrete values: whenever dynamic symbolic execution does not know how to generate a constraint for a program statement depending on some inputs, one can always simplify this constraint using the current concrete values of those inputs. One can prove [60] that dynamic test generation is more precise than static test generation mainly because of its ability to observe concrete values and record those in path constraints.

In practice, the key strength of symbolic execution is that it can generate quality test inputs that exercise program paths with much better precision than random testing or other blackbox heuristic-based test-generation techniques. However, symbolic execution does not have perfect precision, constraint solvers are typically not sound and complete, and programs can have (infinitely) many control paths due to loops or recursion. Moreover, symbolic execution is complex to engineer properly. We discuss these challenges in the next section.

3 Symbolic Execution Meets Practice: Challenges and Solutions

In practice, automatic test generation using symbolic execution suffers from several important limitations. This section discusses these challenges and various solutions.

Fortunately, approximate solutions are sufficient in practice. To be useful, symbolic execution does not need to be perfect, it must simply be "good enough" to drive the program under test through program branches, statements and paths that would be difficult to exercise with simpler techniques like random testing. Even if a systematic search cannot typically explore all the feasible paths of large programs in a reasonable amount of time, it usually does achieve better coverage than pure random testing and, hence, can find new program bugs.

3.1 Exploring New Program Paths

Dynamic symbolic execution is used to systematically explore the execution tree of a program. These paths are discovered incrementally and can be explored independently "in parallel": each inner node is a branching decision, and each leaf is a program state that contains its own address space, program counter, and set of constraints on program variables. In order to exercise a new program

path during such a systematic search, the path constraint for this new path is solved with a constraint solver. If the constraint is satisfiable, the solver returns a satisfying assignment for every symbolic variable in the constraint, which is then mapped to new program inputs. There are two main ways to explore new program paths of a program.

One approach consists of running the program with some fixed concrete inputs and performing dynamic symbolic execution along that execution (using runtime instrumentation) until the program terminates or a specific limit is reached. In this approach, exploring a different execution path requires re-running the target program from the beginning with new concrete inputs. This is the approach used in, e.g., [62,64,107].

A second approach consists in literally "forking" (using the `fork` system call) the program state before branch decisions. This way, a new address space is created for a copy of the program that now explores an alternate path through the tree. Copy-on-write techniques can be used to efficiently deduplicate these address spaces. This is the approach used in, e.g., [23,31].

The two approaches present different trade-offs. The first approach (DART-style concolic execution) presents, on the one hand, the benefit of not having to solve constraints at runtime to determine feasibility of execution paths and requires less memory. On the other hand, it requires re-running the program from scratch for every explored path. The second approach (KLEE-style symbolic execution), on the one hand is able to efficiently explore paths in parallel, without re-running the program, and enables flexible search strategies for deciding which program paths to explore next (e.g., in the presence of loops) based on a broad number of factors. On the other hand, it requires more CPU (in particular for solving constraints to determine path feasibility) and memory, which limits scalability.

3.2 Interacting with the Environment

In theory, symbolic execution does not use abstraction and is therefore fully precise with respect to predicate transformer semantics [46]: it generates "per-path verification conditions" whose satisfiability implies the reachability of a particular statement, so it generally does not produce false positives.

In practice, to test real-world programs, a symbolic execution engine must mediate between the program and its runtime environment, i.e., external libraries, the operating system, the thread and process scheduler, I/O interrupt events, etc. Thus symbolic execution engines need to minimize the time they spend executing the environment, while of course ensuring correct behavior of the program.

Existing solutions roughly fall into two categories: they either concretize calls to the environment and thus avoid symbolic execution of the environment altogether, or they abstract the environment as much as possible using models with varying degrees of completeness. We now discuss those two options.

Concrete Environment. The first modern symbolic execution engines [24,62,107] executed the program concretely, and maintained the symbolic execution

state only during the execution of the program code itself. Whenever symbolic execution is not possible, such as inside external library calls (possibly executed in kernel-mode or on some other machine or process) or when facing program instructions with unknown symbolic semantics, the program execution can still proceed. This approach turns the conventional stance on the role of symbolic execution upside-down: symbolic execution is now an adjunct to concrete execution. As a result, a specific concrete execution can be leveraged as an automatic fall back for symbolic execution [62]. This avoids altogether symbolic execution of the environment.

A benefit of this approach is that it can be implemented incrementally: only some program statements can be instrumented and interpreted symbolically, while others can simply be executed concretely natively. A tool developer can improve the precision of symbolic execution over time, by adding new instruction handlers in a modular manner.

The main drawback is that program behaviors that correspond to environment behaviors other than the ones seen in the concrete executions are not explored.

Modeled Environment. This drawback can be addressed with another approach: model the environment during symbolic execution. For example, KLEE [23] redirects calls to the environment to small functions that understand the semantics of the desired action well enough to generate reasonable responses. With about 2,500 lines of code, they modeled roughly 40 Linux system calls, such as `open`, `read`, and `stat`. These models are hand-written abstractions of actual implementations of the system calls. Subsequently, [18] expanded the models for KLEE to a full POSIX environment.

The benefit of this approach is that target programs can now be exposed to more varied behaviors of the environment, and their reaction can be evaluated. For instance, what does the program do whenever a `write` operation to a file fails due to a full disk? A suitably written model for `write` can have a symbolic return value, which will vary depending on success or failure of that operation. In fact, one can write models that return different error codes for different failures, and a symbolic execution engine can automatically test the program under all these scenarios.

This approach has two main drawbacks. First, by definition, the model is an abstraction of the real code, and it may or may not model all possible behaviors of that code. (If the model was fully precise, it would be equivalent to the actual implementation.) Second, writing models by hand is labor-intensive and prone to error.

To mitigate these drawbacks, selective symbolic execution [31] does not employ models but instead automatically abstracts the environment. In doing so, it is guided by consistency models that govern when to over-approximate and when to under-approximate. More details on this system appear in Sect. 5.

3.3 Path Explosion

Symbolically executing all feasible program paths does not scale to large programs, because the number of feasible paths in a program can be exponential in the program size, or even infinite if the program, such as a network server, has a single loop whose number of iterations may depend on some unbounded input, such as a stream of network packets. A program like the Firefox web browser has more than 500,000 `if` statements; if just one thousandth of them were to have both a `then` and an `else` branch that are feasible for some inputs, then Firefox could be expected to have on the order of 2^{500} paths, which still far exceeds the number of atoms in the observable Universe [115]. We now discuss solutions to this path explosion problem.

A Generic Symbolic Execution and Search Algorithm. In order to present different trade-offs, we describe the operation of a symbolic execution engine in a more precise manner using the worklist-style Algorithm 1.

The algorithm is parameterized by a function *pickNext* for choosing the next state to expand in a worklist, a function *follow* that returns a decision on whether to follow a branch, and a relation $\sim$ that controls whether program states should be merged or kept separate (more on this later). A program state is a triple (ℓ, pc, s) consisting of a program location ℓ, the path condition pc, and the symbolic store s that maps each variable to either a concrete value or an expression over input variables. In line 1, the worklist w is initialized with a state whose symbolic store maps each variable to itself (we ignore named constants, for simplicity). Here, $\lambda x.e$ denotes the function mapping parameter x to an expression e, with $\lambda(x_1, \ldots x_n).e$ mapping multiple parameters. In each iteration, the algorithm picks a new state from the worklist (line 3).

On encountering an assignment $v := e$ (lines 5–6), the algorithm creates a successor state at the fall-through successor location $succ(\ell)$ of ℓ by updating the symbolic store s with a mapping from v to a new symbolic expression obtained by evaluating e in the context of s, and adds the new state to the set S. At every branch (lines 7–11), the algorithm first checks whether to follow either path and, if so, adds the corresponding condition to the successor state, which in turn is added to S. A symbolic execution engine can decide to not follow a branch if the branch is infeasible or would exceed a limit on loop unrolling. For assertions (line 12–13), the path condition, the symbolic store, and the negated assertion are put in conjunction and checked for satisfiability. Halt statements terminate the analyzed program, so the algorithm just outputs the path condition—a satisfying assignment of this condition can be used to generate a test case for the execution leading to the halt.

In lines 16–21, the new states in S are merged with any matching states in the worklist before being added to the worklist themselves. Two states match if they share the same location and are similar according to relation $\sim$. Merging creates a disjunction of the two path conditions and builds the merged symbolic store from ite expressions that assert one or the other original value, depending on the path taken (line 19).

Input: Choice function *pickNext*, similarity relation $\sim$, branch checker *follow*, and initial location ℓ_0.
Data: Worklist w and set of successor states S.

```
1  w := {(ℓ0, true, λv.v)};
2  while w ≠ ∅ do
3      (ℓ, pc, s) := pickNext(w); S := ∅;
       // Symbolically execute the next instruction
4      switch instr(ℓ) do
5          case v := e                                          // assignment
6              S := {(succ(ℓ), pc, s[v ↦ eval(s, e)])};
7          case if(e) goto ℓ′                                   // conditional jump
8              if follow(pc ∧ s ∧ e) then
9                  S := {(ℓ′, pc ∧ e, s)};
10             if follow(pc ∧ s ∧ ¬e) then
11                 S := S ∪ {(succ(ℓ), pc ∧ ¬e, s)};
12         case assert(e)                                       // assertion
13             if isSatisfiable(pc ∧ s ∧ ¬e) then abort else S := {(succ(ℓ), pc, s)}
14         case halt                                            // program halt
15             print pc;
       // Merge new states with matching ones in w
16     forall (ℓ″, pc′, s′) ∈ S do
17         if ∃(ℓ″, pc″, s″) ∈ w : (ℓ″, pc″, s″) ∼ (ℓ″, pc′, s′) then
18             w := w \ {(ℓ″, pc″, s″)};
19             w := w ∪ {(ℓ″, pc′ ∨ pc″, λv.ite(pc′, s′[v], s″[v]))};
20         else
21             w := w ∪ {(ℓ″, pc′, s′)};
22 print "no errors";
```

Algorithm 1. Generic symbolic execution of programs written in a simple input language with assignments, conditional `goto` statements, assertions, and `halt` statements. For simplicity, this algorithm is just intraprocedural; function calls must be inlined. It can generate precise symbolic function summaries, if invoked per procedure and with a similarity relation that merges all states when the function terminates.

Search Heuristics for Program Loops. Bounded model checkers [36] and extended static checkers [7,54,118] unroll loops up to a certain bound, which can be iteratively increased if an injected unwinding assertion fails. Such unrolling is usually performed by statically rewriting the control-flow graph, but can be fit into Algorithm 1 by defining *follow* to return false for branches that would unroll a loop beyond the bound. By default, symbolic execution explores loops as long as it cannot prove the infeasibility of the loop condition; in the case of an unbounded loop, this can lead to an infinite number of unrollings.

Modern symbolic execution typically performs loop unrolling and aims to be smart about choosing which program state to explore next. Dynamic test generation as implemented in DART [62] starts with an arbitrary initial unrolling of the loop (driven by an original concrete input) and explores different unrollings in subsequent tests, driven by follow-on concrete inputs chosen by DART. In other words, it implements *pickNext* to follow concrete executions, postponing branch alternatives to be covered by a subsequent concrete execution. Simple techniques like bounding the number of constraints injected at each program location are effective practical solutions to limit path explosion [64]. Dynamic test generation as implemented in KLEE [23] employs a search strategy implemented in the function *pickNext* that biases the choice of program states from the worklist against states that perform many repetitions of the same loop. For example, a search strategy optimized for line coverage selects states close to unexplored code and avoids states in deep loop unrollings.

While these code-coverage-based search heuristics do not change the number of execution paths through a target program, they focus the search on different parts of the program. In practice, these heuristics are important to explore diverse parts of the program, to avoid being stuck in specific parts of the search space, and hence to try to maximize code coverage and the number of bugs found given a limited time and space budget.

Summaries and State Merging. In addition to search heuristics, one can also *reduce* the number of paths to be explored by *memoizing and merging* program states reached by different program paths. This general idea of analyzing programs *compositionally* (e.g., [105]) is well-known in interprocedural static program analysis and is key to make it scale to large programs (e.g., [21,29,43,70]). In the context of static analysis, a merged state typically *over*-approximates the individual states that are merged [43,88]; even if the resulting imprecision can be reduced, it cannot be eliminated, thus leading to false positives (i.e., infeasible executions). In contrast, in the context of symbolic execution for test generation, as a matter of principle, a merged state must precisely represent the information from *all* execution paths subsumed by that state *without any* over-approximation. In other words, while compositional static analysis typically computes and memoizes *"may"* over-approximate function summaries, compositional test generation computes *"must"* under-approximate summaries [59].

For test generation, symbolic summaries can be computed at the block, method, function or procedure level, or at arbitrary program points. These can be computed incrementally, one interprocedural path at a time, and then bundled together using disjunctions as shown in lines 17–22 of Algorithm 1. The advantage of this approach is that, instead of being symbolically re-executed over and over again in different calling contexts, each intraprocedural path is symbolically executed only once, and the results of that symbolic execution are memoized using local input-preconditions and output post-conditions. A symbolic summary for a procedure is then simply defined as the disjunction of the summaries of its intraprocedural paths. Whenever a higher-level procedure `foo` calls an already-summarized procedure `bar`, the summary for `bar` is re-used and

included in the current path condition of `foo`. The effect of re-using a symbolic summary is thus to *merge* all the states that can be reached when the summarized procedure returns. Such summaries can be computed in various ways, e.g., in an inner-most-first order [59] or lazily on-demand [3], and can also be used together with "may" summaries generated with static analysis (e.g., [67]).

Unfortunately, the main drawback of using summaries and state merging is that both symbolic execution and constraint solving become more complex and expensive: summaries (i.e., disjunctions of sub-program paths) need to be computed and memoized, which makes the path conditions (which now have more disjunctions) harder to solve.

Therefore, various trade-offs have been proposed. Two extreme trade-offs are (i) no state merging at all, i.e., complete separation of paths [23,24,62], and (ii) complete static state merging, as implemented by verification condition generators [7,36,77,118]. Static state merging combines states at join points after completely encoding all subpaths, i.e., it defines *pickNext* to explore all subpaths leading to a join point before picking any states at the join point, and it defines $\sim$ to contain all pairs of states. In search-based symbolic execution engines with no state merging, *pickNext* can be chosen freely according to the search goal, and $\sim$ is empty.

Some approaches adopt intermediate merging strategies. In the context of bounded model checking (BMC), Ganai and Gupta [56] investigate splitting the verification condition along subsets of paths; this moves BMC a step into the direction of symbolic execution, and corresponds to partitioning the $\sim$ relation. Hansen et al. [72] describe an implementation of static state merging in which they modify the exploration strategy to effectively traverse the control-flow graph in topological order and merge all states that share the same program location. Alas, for two of their three tested examples, the total solving time increases with this strategy due to the added load placed on the constraint solver by the increased complexity of the merged path conditions.

Indeed, the challenge in state merging is that, while merging two program states on the one hand may reduce the number of execution paths by an exponential factor, it could also increase the time the symbolic execution engine spends solving the more complex constraints. The net effect can be positive or negative. [82] proposes a technique for deciding when merging two program states is expected to be benefical vs. not. The approach uses a query count estimation algorithm to compute, during symbolic execution, whether the performance benefit resulting from the reduction in number of execution paths would outweigh the increase in constraint solver time. Experiments showed that this approach consistently achieved several orders of magnitude speedup over the then state of the art.

An alternative to implementing state merging inside the symbolic execution engine is to present this same engine with an equivalent variant of the target program that is easier to symbolically execute. Whereas a usual compiler translates programs into code that executes as quickly as possible on a target CPU, taking into account CPU-specific properties, Overify [116] instead compiles programs

to have the simplest possible control flow, using techniques like jump threading, loop unswitching, transforming conditionally executed side-effect-free statements into speculative branch-free versions, splitting objects into smaller ones to reduce opportunities for pointer aliasing, and others. The net effect is that compiling a program with the `-Overify` option reduces the time of exhaustive symbolic execution by up to almost two orders of magnitude.

3.4 Efficient Constraint Solving

Another key component is the constraint solver being used to solve path constraints. Over the last decade, several tools implementing dynamic test generation for various programming languages, properties and application domains have been developed. Examples of such tools are DART [62], EGT [22], PathCrawler [117], CUTE [107], EXE [24], SAGE [64], CatchConv [93], PEX [113], KLEE [23], CREST [20], BitBlaze [111], Splat [87], Apollo [4], YOGI [67], Kudzu [106], S2E [31], and JDart [85]. The above tools differ by how they perform dynamic symbolic execution (for languages such as C, Java, x86, .NET), by the type of constraints they generate (for theories such as linear arithmetic, bit-vectors, arrays, uninterpreted functions, etc.), and by the type of constraint solvers they use (such as lp_solve, CVClite, STP, Disolver, Yikes, Z3). Indeed, like in traditional static program analysis and abstract interpretation, these important parameters are determined in practice depending on which type of program is to be tested, on how the program interfaces with its environment, and on which properties are to be checked. Moreover, various cost/precision tradeoffs are also possible while generating and solving constraints.

Fortunately, and driven in part by the test generation applications reported in this article, the science and engineering of automated theorem proving has made a lot of progress over the last decade as well. Notably, the last decade witnessed the birth and rise of so-called Satisfiability-Modulo-Theories (SMT) solvers [2,10,16,49,57,95], which can efficiently check satisfiability of complex constraints expressed in rich domains. Such solvers have also become computationally affordable in recent years thanks to the increasing computational power available on modern computers.

3.5 Parallelization and Testing as a Cloud Service

An orthogonal approach to speed up symbolic execution is *parallelization* of path exploration on a cluster of machines, harnessing its aggregate CPU and memory capabilities (alternatively, one could imagine running a symbolic execution engine on a supercomputer). One way to parallelize symbolic execution is by statically dividing up the task among nodes and have them run independently. However, when running on large programs, this approach leads to high workload imbalance among nodes, making the entire cluster proceed at the pace of the slowest node—if this node gets stuck, for instance, while symbolically executing a loop, the testing process may never terminate. In [18], a method is described for parallelizing symbolic execution on shared-nothing clusters in a

way that scales well. Without changing the exponential nature of the problem, parallel symbolic execution harnesses cluster resources to make it feasible to run automated testing on larger systems than would otherwise be possible.

In essence, software testing reduces to exercising as many paths through a program as possible and checking that certain properties hold along those paths (no crashes, no buffer overflows, etc.). The advances in symbolic execution lead naturally to the "testing as a service" (TaaS) vision [26,66] of (1) offering software testing as a competitive, easily accessible online service, and (2) doing fully automated testing in the cloud, to harness vast, elastic resources toward making automated testing practical for real software. A software-testing service allows users and developers to upload the software of interest, instruct the service what type of testing to perform, click a button, and then obtain a report with the results. For professional uses, TaaS can integrate directly with the development process and test the code as it is written. TaaS can also serve as a publicly available certification service that enables comparing the reliability and safety of software products [25].

Having seen some key challenges in automated test generation and possible solutions, we now describe two recent systems that employ many of these techniques: SAGE (Sect. 4) and S2E (Sect. 5).

4 Whitebox Fuzzing with SAGE

Whitebox fuzzing of file parsers [64] is a "killer app" for automatic test generation using dynamic symbolic execution and constraint solving. Many security vulnerabilities are due to programming errors in code for parsing files and packets that are transmitted over the Internet. For instance, the Microsoft Windows operating system includes parsers for hundreds of file formats. A security vulnerability in any of those parsers may require the deployment of a costly, visible security patch to more than a billion PCs, i.e., millions of dollars [65]. Because of the magnitude of this problem, Microsoft has invested significant resources to hunt security vulnerabilities in its products, and provided the right environment for whitebox fuzzing to mature to an unprecedented level.

Whitebox fuzzing extends dynamic test generation from unit testing to whole-program security testing in three main ways: First, inspired by blackbox fuzzing [55], whitebox fuzzing performs dynamic test generation starting from one or several *well-formed* inputs, which is a heuristic to increase code coverage quickly and give the search a head-start. Second, again like blackbox fuzzing, the focus of whitebox fuzzing is to find *security vulnerabilities*, like buffer overflows, not to check functional correctness; finding such security vulnerabilities can be done fully automatically and does not require an application-specific test *oracle* or functional specification. Third, and more importantly, the main technical novelty of whitebox fuzzing is *scalability*: it extends the scope of dynamic test generation from (small) units to (large) whole programs. Whitebox fuzzing scales to large file parsers embedded in applications with millions of lines of code and execution traces with hundreds of millions of machine instructions.

Since whitebox fuzzing targets large applications, it must scale to long program executions, and such symbolic execution is expensive. For instance, a single symbolic execution of Microsoft Excel with 45,000 input bytes executes nearly a billion x86 instructions. In this context, whitebox fuzzing uses a novel directed search algorithm, dubbed *generational search*, that maximizes the number of new input tests generated from each symbolic execution. Given a path constraint, *all* the constraints in that path are systematically negated one-by-one, placed in a conjunction with the prefix of the path constraint leading to it, and attempted to be solved by a constraint solver. This way, a single symbolic execution can generate thousands of new tests. (In contrast, a standard depth-first or breadth-first search would negate only the last or first constraint in each path constraint, and generate at most one new test per symbolic execution.)

Whitebox fuzzing was first implemented in the tool SAGE, short for Scalable Automated Guided Execution [64]. SAGE was the first tool to perform dynamic symbolic execution at the x86 binary level, which allows it to be used on any program regardless of its source language or build process. It also ensures that "what you fuzz is what you ship," as compilers can perform source-code changes which may impact security.

SAGE uses several optimizations that are crucial for dealing with huge execution traces with billions of machine instructions. To scale to such execution traces, SAGE uses several techniques to improve the speed and memory usage of constraint generation: *symbolic-expression caching* ensures that structurally equivalent symbolic terms are mapped to the same physical object; *unrelated constraint elimination* reduces the size of constraint solver queries by removing the constraints which do not share symbolic variables with the negated constraint; *local constraint caching* skips a constraint if it has already been added to the path constraint; *flip count limit* establishes the maximum number of times constraints generated from a particular program branch can be flipped; using a cheap syntactic check, *constraint subsumption* eliminates constraints logically implied by other constraints injected at the same program branch (mostly likely due to successive iterations of an input-dependent loop) [64].

Since 2008, SAGE has been running in production on hundreds of machines, automatically fuzzing hundreds of applications in Microsoft security testing labs. This is over 500 machine-years and the "largest computational usage ever for any Satisfiability-Modulo-Theories (SMT) solver" according to the authors of the Z3 SMT solver [95], with over four billion constraints processed to date [13].

During this fuzzing, SAGE found many new security vulnerabilities (buffer overflows) in many Windows parsers and Office applications, including image processors, media players, file decoders, and document parsers. Notably, SAGE found roughly one third of *all* the bugs discovered by file fuzzing during the development of Microsoft's Windows 7 [65], saving millions of dollars by avoiding expensive security patches for nearly a billion PCs worldwide. Because SAGE was typically run last, those bugs were missed by everything else, including static program analysis and blackbox fuzzing.

In 2015, SAGE and other popular blackbox fuzzers used internally at Microsoft were packaged into Project Springfield [91], the first commercial cloud fuzzing service (renamed Microsoft Security Risk Detection in 2017). Customers who subscribe to this service can submit fuzzing jobs targeting their own software and benefit from the technology described in this article. No source code or symbols are required. Springfield fuzzing jobs are easy to set up by non-experts, and are processed using the same tools used for over a decade inside Microsoft, for automatic job validation, seed minimization, fuzzing on many machines (leveraging the cloud) each running different fuzzing tools and configurations, and then automatic analysis, triage and prioritization of the bugs found, with results available directly on the Springfield web-site.

5 Selective Symbolic Execution with S2E

When the target of testing is not individual applications but *systems* code (e.g., device drivers) or *full system stacks* (i.e., the operating system kernel with drivers, libraries, and applications all together), the dominant considerations become the interaction between the tested software and its environment (Sect. 3.2) and scaling beyond a single application. These challenges motivated the development of the S2E system [31].

S2E is a symbolic execution engine for x86 and ARM binaries. It is built around a modified version of the QEMU [12] virtual machine, and it dynamically dispatches guest machine instructions either to the host CPU for native execution or to a symbolic execution engine (KLEE [23]) embedded inside S2E. Most instructions do not access symbolic state, so they can run natively, while the rest are interpreted symbolically.

The original use case for S2E was testing full system stacks with complex environments (proprietary OS kernel, many interdependent libraries, etc.). An accurate assessment of program behavior requires taking into account every *relevant* detail of the environment, but making *a priori* the choice of what is relevant is hard. So S2E offers "in-vivo" symbolic execution, in which the environment is automatically abstracted on-the-fly, as needed. This is in contrast to "in-vitro" symbolic execution, as done by symbolic execution engines and model checkers that use stubs or models [8,23,97]).

Such in-vivo symbolic execution was used both in industrial and academic settings. Engineers at Intel used S2E to search for security vulnerabilities in UEFI BIOS implementations [11]. DDT [81] used S2E to test closed-source proprietary Microsoft Windows device drivers, without access to the driver source code or inside knowledge of the Windows kernel. DDT found memory leaks, segmentation faults, race conditions, and memory corruption bugs in drivers that had been shipping with Windows for many years. Network researchers used S2E to verify the dataplane of software network routers [47] by using a form of exhaustive symbolic execution on specific elements in the router without having to model the rest of the router.

The key design choice that enables in-vivo symbolic execution is to carefully maintain a unified global state store that simultaneously captures both symbolic and concrete execution state. An execution can thus alternate between concrete and symbolic execution multiple times during the same run, without losing desired correctness and precision. For example, when symbolically executing a driver in-vivo, the initial execution starts out concrete from userspace, is concrete in the kernel, then switches to symbolic when entering the driver, then back to concrete when the driver calls into the kernel, back to symbolic when execution returns inside the driver, and so on. In this way, "interesting" program paths are explored in the software of interest (the driver, in this example) without symbolically executing the rest of the real environment.

The unified state store consists of machine state (memory, CPU registers and flags, system clock, devices, etc.) that is shared between the symbolic execution engine and the virtual machine. S2E is responsible for the transparent conversions of concrete state to symbolic state and vice versa, governed by an *execution consistency model*. For example, under a so-called "locally consistent execution" model, when a driver calls `kmalloc` with a symbolic argument λ to allocate kernel memory, S2E automatically picks a concrete value that satisfies the path constraint, say 64, and executes `kmalloc` concretely in the kernel. Once the kernel returns the concrete 64-byte memory buffer, S2E returns to the driver a symbolic two-valued pointer $p = 0 \mid \&buffer$ capturing the two possible outcomes of the `kmalloc` call. S2E then augments the path constraint with $\lambda = 64$ and continues symbolic execution in the driver. Depending on consistency model, if λ is later involved in a branch condition with one of the branches made infeasible by the $\lambda = 64$ constraint, S2E can return to the original `kmalloc` call site to re-concretize λ to an additional value that enables that branch, and repeat the call.

Execution consistency models in S2E are analogous to memory consistency models [94]. The traditional assumption about system execution is that the state at any point in time is consistent, i.e., there exists a feasible concrete-execution path from the system's start state to the system's current state. This is what S2E calls "strictly consistent concrete execution," and this is the strongest of all execution consistency models. Relaxing this assumption results in the definition of five additional consistency models [32], each offering different trade-offs. For example, "strictly consistent unit-level execution" corresponds to the consistency model that governs how DART [62] and EXE [24] handle the environment. The "locally consistent execution" mentioned above is the one DDT [81] employed for the interface between device drivers and the OS kernel.

An interesting concept in S2E is that of *symbolic hardware*. It corresponds to "overapproximately consistent execution," and allows virtualized hardware to return unconstrained symbolic values. DDT [81] and SymDrive [104] used this execution consistency model for the hardware interface, in order to test drivers for hardware error paths that are difficult to exercise, and to make up for the all-together absence of the hardware.

One of S2E's optimizations is *lazy concretization*: concretize symbolic values on-demand, only when concretely-running code is about to branch on a condition that depends on that value. This makes it possible to carry a lot of data through the layers of the system stack without conversion. For example, when a program writes a buffer of symbolic data to the filesystem, there are usually no branches in the kernel or the disk device driver that depend on the data per se, so S2E can pass the buffer through unconcretized and write it in symbolic form to the virtual disk, from where it can later be read back in its symbolic form, thus avoiding the loss of precision inherent in concretization.

Over time, S2E turned into a general platform for software analysis, and saw a number of surprising use cases. For example, RevNIC [30] employed "overapproximately consistent execution" to automatically reverse-engineer proprietary Windows device drivers and produce equivalent drivers for different platforms. In [32], S2E was used to develop a comprehensive performance profiler to measure instruction count, cache misses, TLB misses, and page faults for arbitrary memory hierarchies along all paths of a program. A side effect of S2E's design as a virtual machine is that it can be used not only for proprietary software but also for self-modifying, JITed, and/or obfuscated and packed/encrypted binaries. This made S2E well suited for malware analysis in a commercial setting [41]. It was also used to develop Chef [17], a tool for turning the vanilla interpreter of a dynamically interpreted language (like Python) into a sound and complete symbolic execution engine for that language. As a final example, two of the seven systems competing in the finals of DARPA's Cyber Grand Challenge in 2016 were based on S2E. This competition was an all-machine computer security tournament, where each competing machine had to autonomously analyze computer programs, find security vulnerabilities, fix them, and launch attacks on other competitors. As part of Galactica (one of the DARPA competitors), S2E launched 392 successful attacks during the competition, twice as many as the competition's all-around winner.

S2E is currently an open-source project [51] and is also at the heart of several commercial cybersecurity products. S2E illustrates how automated test generation can morph into a variety of other forms of program analysis.

6 Other Approaches to Automated Test Generation

As mentioned in the introduction, this paper is not a survey on automatic test generation. We do mention briefly here some other notable test-generation techniques.

Model-Based Testing. Given an abstract representation of the program, called *model*, model-based testing consists of generating tests by analyzing the model in order to check the *conformance* of the program with respect to the model (e.g., [114]). Such models are usually program specifications written by hand, but they can also be generated automatically using machine-learning techniques (e.g., see [69,102] and the article on automata learning in this volume [74]). In contrast, the code-driven test-generation techniques discussed in this article do

not use or require a model of the program under test. Instead, their goal is to generate tests that exercise as many program statements as possible, including assertions inserted in the code.

Grammar-Based Fuzzing. Most popular blackbox random fuzzers for security testing support some form of grammar representation to specify the input format of the application under test, e.g., Peach [99] and SPIKE [100], among many others [112]. Such grammars are typically written by hand, and this process is laborious, time consuming, and error-prone. Nevertheless, grammar-based fuzzing is the most effective fuzzing technique known today for fuzzing applications with complex structured input formats, like web-browsers which must take as inputs web-pages including complex HTML documents and JavaScript code. Work on grammar-based test input generation started in the 1970's [71,101]. Test generation from a grammar is usually either random [40,89,110] or exhaustive [83]. Imperative generation [33,42] is a related approach in which a custom-made program generates the inputs (in effect, the program encodes the grammar). Grammar-based fuzzing can also be combined with whitebox fuzzing [61,86].

Search-Based Test Generation. Test generation can be viewed as a search and optimization problem (e.g., how to maximize code coverage), and various heuristics and search techniques have been proposed, for instance, using genetic algorithms and simulated annealing (e.g., [90]). The fuzzing heuristics using code-coverage feedback mentioned earlier are related to these techniques. These techniques have also been applied to other software engineering problems, including other testing-related problems such as test case minimization and test case prioritization [90].

Exploit Generation. A targetted form of security testing is exploit generation: given a program, automatically find vulnerabilities and generate exploits for them. Systems like Mayhem [28] have used pre-conditioned symbolic execution to find and exploit zero-day security bugs [6], and work prior to that augmented such test generation with knowledge from security patches in order to reverse-engineer the exploits against which those patches were defending [15].

Combinatorial Testing. Given a program and a set of input parameters, combinatorial testing aims at generating efficiently a set of test inputs which cover all pairs of input parameters (e.g., [39]). Generalizations from pairs to arbitrary k-tuples have also been proposed. In practice, these techniques are used provided the number of input parameters is sufficiently small, e.g., for configuration parameters [39].

Concurrency Testing. Systematic testing techniques and algorithms have also been proposed for concurrent software (e.g., [50,53,58,96]). Such techniques explore the possible interleavings of multiple processes or threads using a runtime scheduler with the goal of finding concurrency-related bugs such as deadlocks and race conditions.

Runtime Verification. Runtime verification tools (e.g., [48,73]) monitor at run-time the behavior of a program and compare this behavior against a

high-level specification, typically represented as a finite-state automaton or a temporal-logic formula. These tools can be viewed as extensions of runtime checking tools mentioned earlier (like Purify and AddressSanitizer), and are complementary to test generation.

Program Verification. Over the past few decades, symbolic execution, constraint generation and automated theorem proving have also been developed further in various ways for program verification, such as verification-condition generation (e.g., [9,46]), symbolic model checking [19] and bounded model checking [35]. Program verification aims at proving the absence of program errors, while test generation aims at generating concrete test inputs that can drive the program to execute specific program statements or paths. A detailed technical comparison can be found in [63]. In practice, symbolic execution, constraint generation and solving are typically not sound and complete, and *fully automatic* program verification remains elusive for large complex software.

7 Conclusion

This article presented an introduction and overview of automated test generation for software. We discussed how test generation using dynamic symbolic execution can be more precise than static test generation and other forms of test generation such as random, taint-based and coverage-heuristic-based test generation. This test generation approach is also more sophisticated, requiring the use of automated theorem proving for solving path constraints. This machinery is more complex and heavy-weight, but may exercise more program paths, find more bugs and generate fewer redundant tests covering the same path. Whether this better precision is worth the trouble depends on the application domain.

Research on automatic test generation has been carried out over many years and is still an active area of research. The techniques described in this article have been implemented in tens of tools. The application of those tools has, collectively, found thousands of new bugs, many of them critical from a reliability or security point of view, in many different application domains.

While this progress is significant and enoucouraging, there is room for further improvements. Automated test generation tools have been successfully applied to several application domains so far, but they are not being used routinely yet by most software developers and testers. New application domains are arising for which the techniques described here need significant rethinking. For example the models automatically learned by machine-learning algorithms (e.g., used in self-driving vehicles) are unlike regular programs, and so testing them automatically requires new approaches. More work is required to further lower the cost of automated test generation (e.g., in terms of computation and memory) while increasing the value it provides (e.g., by providing actionable information on how to fix the bugs instead of just providing test cases). Automated testing can benefit from human insights (both in terms of providing test criteria and in prioritizing test case search), and the potential of combining human intelligence

with fuzzing and symbolic execution has yet to be realized. We also see opportunities for machine learning in automated testing, such as in learning input grammars from examples and leveraging these grammars for generating tests. Finally, the goal of automated test generation is to enable software engineers to produce better software faster, and further research is required on how best to integrate testing tools in software development processes, in particular modern agile processes and continuous integration, and how to tighten the feedback loop for fixing bugs.

References

1. Aizatsky, M., Serebryany, K., Chang, O., Arya, A., Whittaker, M.: Announcing OSS-Fuzz: Continuous fuzzing for open source software (2016). https://testing.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html
2. Alberti, F., Ghilardi, S., Sharygina, N.: Decision procedures for flat array properties. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 15–30. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_2
3. Anand, S., Godefroid, P., Tillmann, N.: Demand-driven compositional symbolic execution. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 367–381. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_28
4. Artzi, S., Kiezun, A., Dolby, J., Tip, F., Dig, D., Paradkar, A.M., Ernst, M.D.: Finding bugs in web applications using dynamic test generation and explicit-state model checking. IEEE Trans. Softw. Eng. **36**(4), 474–494 (2010)
5. Arya, A., Neckar, C.: Fuzzing for security (2012). https://blog.chromium.org/2012/04/fuzzing-for-security.html
6. Avgerinos, T., Cha, S.K., Hao, B.L.T., Brumley, D.: AEG: automatic exploit generation. In: Network and Distributed System Security Symposium (2011)
7. Babic, D., Hu, A.J.: Calysto: scalable and precise extended static checking. In: International Conference on Software Engineering (2008)
8. Ball, T., Bounimova, E., Levin, V., Kumar, R., Lichtenberg, J.: The static driver verifier research platform. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 119–122. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_11
9. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: a modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006). https://doi.org/10.1007/11804192_17
10. Barrett, C., et al.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14
11. Bazhaniuk, O., Loucaides, J., Rosenbaum, L., Tuttle, M.R., Zimmer, V.: Symbolic execution for BIOS security. In: USENIX Workshop on Offensive Technologies (2015)
12. Bellard, F.: QEMU, a fast and portable dynamic translator. In: USENIX Annual Technical Conference (2005)
13. Bounimova, E., Godefroid, P., Molnar, D.: Billions and billions of constraints: whitebox fuzz testing in production. In: International Conference on Software Engineering (2013)

14. Boyer, R.S., Elspas, B., Levitt, K.N.: SELECT - a formal system for testing and debugging programs by symbolic execution. SIGPLAN Not. **10**, 234–245 (1975)
15. Brumley, D., Poosankam, P., Song, D., Zheng, J.: Automatic patch-based exploit generation is possible: techniques and implications. In: IEEE Symposium on Security and Privacy (2008)
16. Brummayer, R., Biere, A.: Boolector: an efficient SMT solver for bit-vectors and arrays. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 174–177. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00768-2_16
17. Bucur, S., Kinder, J., Candea, G.: Prototyping symbolic execution engines for interpreted languages. In: International Conference on Architectural Support for Programming Languages and Operating Systems (2014)
18. Bucur, S., Ureche, V., Zamfir, C., Candea, G.: Parallel symbolic execution for automated real-world software testing. In: ACM EuroSys European Conference on Computer Systems (2011)
19. Burch, J., Clarke, E., McMillan, K., Dill, D., Hwang, L.: Symbolic model checking: 10^{20} states and beyond. In: Symposium on Logic in Computer Science (1990)
20. Burnim, J., Sen, K.: Heuristics for scalable dynamic test generation. In: International Conference on Automated Software Engineering (2008)
21. Bush, W., Pincus, J., Sielaff, D.: A static analyzer for finding dynamic programming errors. Softw. Practice Exp. **30**(7), 775–802 (2000)
22. Cadar, C., Engler, D.: Execution generated test cases: how to make systems code crash itself. In: Godefroid, P. (ed.) SPIN 2005. LNCS, vol. 3639, pp. 2–23. Springer, Heidelberg (2005). https://doi.org/10.1007/11537328_2
23. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Symposium on Operating System Design and Implementation (2008)
24. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: automatically generating inputs of death. In: Conference on Computer and Communication Security (2006)
25. Candea, G.: A software certification service. In: Symposium on Operating System Design and Implementation (2008). "Research Vision" talk session
26. Candea, G., Bucur, S., Zamfir, C.: Automated software testing as a service. In: Symposium on Cloud Computing (2010)
27. CERT: CERT database of security vulnerabilities (2017). http://www.cert.org/vulnerability-analysis/knowledgebase/
28. Cha, S.K., Avgerinos, T., Rebert, A., Brumley, D.: Unleashing Mayhem on binary code. In: IEEE Symposium on Security and Privacy (2012)
29. Chen, H., Dean, D., Wagner, D.: Model checking one million lines of C code. In: Network and Distributed System Security Symposium (2004)
30. Chipounov, V., Candea, G.: Reverse engineering of binary device drivers with RevNIC. In: ACM EuroSys European Conference on Computer Systems (2010)
31. Chipounov, V., Kuznetsov, V., Candea, G.: S2E: a platform for in-vivo multi-path analysis of software systems. In: International Conference on Architectural Support for Programming Languages and Operating Systems (2011)
32. Chipounov, V., Kuznetsov, V., Candea, G.: The S2E platform: design, implementation, and applications. ACM Trans. Comput. Syst. **30**(1), 2 (2012). Special issue: Best papers of ASPLOS
33. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: ACM SIGPLAN International Conference on Functional Programming (2000)

34. Clang Users Manual: Undefined behavior sanitizer (2017). http://clang.llvm.org/docs/UsersManual.html
35. Clarke, E.M., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Formal Methods Syst. Des. **19**(1), 7–34 (2001)
36. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24730-2_15
37. Clarke, L.A.: A program testing system. In: ACM Annual Conference (1976)
38. Clarke, L.A., Richardson, D.J.: Applications of symbolic evaluation. J. Syst. Softw. **5**(1), 15–35 (1985)
39. Cohen, D.M., Dalal, S.R., Parelius, J., Patton, G.C.: The combinatorial design approach to automatic test generation. IEEE Softw. **13**(5), 83–88 (1996)
40. Coppit, D., Lian, J.: Yagg: an easy-to-use generator for structured test inputs. In: International Conference on Automated Software Engineering (2005)
41. Cyberhaven Inc: Cyberhaven product line. http://cyberhaven.io/
42. Daniel, B., Dig, D., Garcia, K., Marinov, D.: Automated testing of refactoring engines. In: Symposium on the Foundations of Software Engineering (2007)
43. Das, M., Lerner, S., Seigle, M.: ESP: path-sensitive program verification in polynomial time. In: International Conference on Programming Language Design and Implementation (2002)
44. Dhurjati, D., Kowshik, S., Adve, V.: Safecode: enforcing alias analysis for weakly typed languages. In: International Conference on Programming Language Design and Implementation (2006)
45. Dijkstra, E.W.: Notes on Structured Programming. In: Structured Programming. Academic Press, Cambridge (1972)
46. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM **18**(8), 453–457 (1975)
47. Dobrescu, M., Argyraki, K.: Software dataplane verification. In: Symposium on Networked Systems Design and Implementation (2014)
48. Drusinsky, D.: The temporal rover and the ATG rover. In: Havelund, K., Penix, J., Visser, W. (eds.) SPIN 2000. LNCS, vol. 1885, pp. 323–330. Springer, Heidelberg (2000). https://doi.org/10.1007/10722468_19
49. Dutertre, B., de Moura, L.: A fast linear-arithmetic solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006). https://doi.org/10.1007/11817963_11
50. Edelstein, O., Farchi, E., Goldin, E., Nir, Y., Ratsaby, G., Ur, S.: Framework for testing multi-threaded Java programs. Concurrency Comput.: Practice Exp. **15**(3–5), 485–499 (2003)
51. EPFL and Cyberhaven Inc: S2E software distribution. http://s2e.systems/
52. Etoh, H.: Propolice: GCC extension for protecting applications from stack-smashing attacks (2000). https://www.researchgate.net/publication/243483996_GCC_extension_for_protecting_applications_from_stack-smashing_attacks
53. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: Symposium on Principles of Programming Languages (2005)
54. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: International Conference on Programming Language Design and Implementation (2002)
55. Forrester, J.E., Miller, B.P.: An empirical study of the robustness of windows NT applications using random testing. In: USENIX Windows System Symposium (2000)

56. Ganai, M.K., Gupta, A.: Tunneling and slicing: towards scalable BMC. In: Design Automation Conference (2008)
57. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 519–531. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73368-3 52
58. Godefroid, P.: Model checking for programming languages using VeriSoft. In: Symposium on Principles of Programming Languages (1997)
59. Godefroid, P.: Compositional dynamic test generation. In: Symposium on Principles of Programming Languages (2007)
60. Godefroid, P.: Higher-order test generation. In: International Conference on Programming Language Design and Implementation (2011)
61. Godefroid, P., Kiezun, A., Levin, M.Y.: Grammar-based whitebox fuzzing. In: International Conference on Programming Language Design and Implementation (2008)
62. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: International Conference on Programming Language Design and Implementation (2005)
63. Godefroid, P., Lahiri, S.K.: From program to logic: an introduction. In: LASER Summer School (2012)
64. Godefroid, P., Levin, M., Molnar, D.: Automated whitebox fuzz testing. In: Network and Distributed System Security Symposium (2008)
65. Godefroid, P., Levin, M., Molnar, D.: SAGE: whitebox fuzzing for security testing. Commun. ACM **55**(3) (2012)
66. Godefroid, P., Molnar, D.: Fuzzing in the cloud. Technical report MSR-TR-2010-29, Microsoft Research, March 2010
67. Godefroid, P., Nori, A., Rajamani, S., Tetali, S.: Compositional may-must program analysis: unleashing the power of alternation. In: Symposium on Principles of Programming Languages (2010)
68. Gupta, N., Mathur, A.P., Soffa, M.L.: Generating test data for branch coverage. In: International Conference on Automated Software Engineering (2000)
69. Hagerer, A., Hungar, H., Niese, O., Steffen, B.: Model generation by moderated regular extrapolation. In: Kutsche, R.-D., Weber, H. (eds.) FASE 2002. LNCS, vol. 2306, pp. 80–95. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45923-5 6
70. Hallem, S., Chelf, B., Xie, Y., Engler, D.: A system and language for build-ing system-specific static analyses. In: International Conference on Programming Language Design and Implementation (2002)
71. Hanford, K.: Automatic generation of test cases. IBM Syst. J. **9**(4) (1970)
72. Hansen, T., Schachte, P., Søndergaard, H.: State joining and splitting for the symbolic execution of binaries. In: Bensalem, S., Peled, D.A. (eds.) RV 2009. LNCS, vol. 5779, pp. 76–92. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04694-0 6
73. Havelund, K., Rosu, G.: Monitoring Java programs with Java PathExplorer. In: International Conference on Runtime Verification (2001)
74. Howar, F., Jonsson, B., Vaandrager, F.: Combining black-box and white-box techniques for learning register automata. In: Steffen, B., Woeginger, G. (eds.) Computing and Software Science. LNCS, vol. 10000. Springer, Heidelberg (2018)
75. Howard, M., Lipner, S.: The Security Development Lifecycle. Microsoft Press (2006)

76. Howden, W.: Symbolic testing and the DISSECT symbolic evaluation system. IEEE Trans. Softw. Eng. **3**(4) (1977)
77. Ivančić, F., Yang, Z., Ganai, M.K., Gupta, A., Shlyakhter, I., Ashar, P.: F-Soft: software verification platform. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 301–306. Springer, Heidelberg (2005). https://doi.org/10.1007/11513988_31
78. Jelinek, J.: Fortify_source: Object size checking to prevent (some) buffer overflows. https://gcc.gnu.org/ml/gcc-patches/2004-09/msg02055.html
79. King, J.C.: Symbolic execution and program testing. J. ACM **19**(7) (1976)
80. Korel, B.: A dynamic approach of test data generation. In: IEEE Conference on Software Maintenance (1990)
81. Kuznetsov, V., Chipounov, V., Candea, G.: Testing closed-source binary device drivers with DDT. In: USENIX Annual Technical Conference (2010)
82. Kuznetsov, V., Kinder, J., Bucur, S., Candea, G.: Efficient state merging in symbolic execution. In: International Conference on Programming Language Design and Implementation (2012)
83. Lämmel, R., Schulte, W.: Controllable combinatorial coverage in grammar-based testing. In: Uyar, M.Ü., Duale, A.Y., Fecko, M.A. (eds.) TestCom 2006. LNCS, vol. 3964, pp. 19–38. Springer, Heidelberg (2006). https://doi.org/10.1007/11754008_2
84. Libfuzzer–a library for coverage-guided fuzz testing. http://llvm.org/docs/LibFuzzer.html
85. Luckow, K., Dimjašević, M., Giannakopoulou, D., Howar, F., Isberner, M., Kahsai, T., Rakamarić, Z., Raman, V.: JDart: a dynamic symbolic analysis framework. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 442–459. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_26
86. Majumdar, R., Xu, R.: Directed test generation using symbolic grammars. In: International Conference on Automated Software Engineering (2007)
87. Majumdar, R., Xu, R.-G.: Reducing test inputs using information partitions. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 555–569. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_41
88. Mauborgne, L., Rival, X.: Trace partitioning in abstract interpretation based static analyzers. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 5–20. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31987-0_2
89. Maurer, P.: Generating test data with enhanced context-free grammars. IEEE Softw. **7**(4) (1990)
90. McMinn, P.: Search-based software test data generation: a survey. Int. J. Softw. Test. Verification Reliab. **14**(2) (2004)
91. Microsoft: Project springfield. https://www.microsoft.com/springfield/
92. Miller, B., Fredriksen, L., So, B.: An empirical study of the reliability of UNIX utilities. Commun. ACM **33**(12) (1990)
93. Molnar, D., Wagner, D.: Catchconv: symbolic execution and run-time type inference for integer conversion errors. Technical report EECS-2007-23, U.C. Berkeley (2007)
94. Mosberger, D.: Memory consistency models. ACM SIGOPS Oper. Syst. Rev. **27**(1) (1993)
95. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24

96. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: International Conference on Programming Language Design and Implementation (2007)
97. Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A., Neamtiu, I.: Finding and reproducing Heisenbugs in concurrent programs. In: Symposium on Operating System Design and Implementation (2008)
98. Offutt, A.J., Jin, Z., Pan, J.: The dynamic domain reduction procedure for test data generation. Softw. Practice Exp. **29**(2) (1999)
99. Peach fuzzer (2017). http://www.peachfuzzer.com/
100. Spike fuzzer (2017). http://resources.infosecinstitute.com/fuzzer-automation-with-spike/
101. Purdom, P.: A sentence generator for testing parsers. BIT Numer. Math. **12**(3) (1972)
102. Raffelt, H., Merten, M., Steffen, B., Margaria, T.: Dynamic testing via automata learning. Int. J. Softw. Tools Technol. Transf. **11**(4) (2009)
103. Ramamoorthy, C., Ho, S.B., Chen, W.: On the automated generation of program test data. IEEE Trans. Softw. Eng. **2**(4) (1976)
104. Renzelmann, M.J., Kadav, A., Swift, M.M.: Symdrive: testing drivers without devices. In: Symposium on Operating System Design and Implementation (2012)
105. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: Symposium on Principles of Programming Languages (1995)
106. Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., Song, D.: A symbolic execution framework for javascript. In: IEEE Symposium on Security and Privacy, pp. 513–528 (2010)
107. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: Symposium on the Foundations of Software Engineering (2005)
108. Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D.: AddressSanitizer: a fast address sanity checker. In: USENIX Annual Technical Conference (2012)
109. Serebryany, K., Iskhodzhanov, T.: ThreadSanitizer - data race detection in practice. In: Workshop on Binary Instrumentation and Applications (2009)
110. Sirer, E., Bershad, B.: Using production grammars in software testing. In: Conference on Domain-Specific Languages (1999)
111. Song, D., et al.: BitBlaze: a new approach to computer security via binary analysis. In: Sekar, R., Pujari, A.K. (eds.) ICISS 2008. LNCS, vol. 5352, pp. 1–25. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89862-7_1
112. Sutton, M., Greene, A., Amini, P.: Fuzzing: Brute Force Vulnerability Discovery. Addison-Wesley Professional, Boston (2007)
113. Tillmann, N., de Halleux, J.: Pex–white box test generation for .NET. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 134–153. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-79124-9_10
114. Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model-based testing approaches. Int. J. Softw. Test. Verification Reliab. **22**(5) (2012)
115. Villanueva, J.C.: How many atoms are there in the universe? (2015). http://www.universetoday.com/36302/atoms-in-the-universe/
116. Wagner, J., Kuznetsov, V., Candea, G.: -OVERIFY: optimizing programs for fast verification. In: Workshop on Hot Topics in Operating Systems (2013)

117. Williams, N., Marre, B., Mouy, P., Roger, M.: PathCrawler: automatic generation of path tests by combining static and dynamic analysis. In: Dal Cin, M., Kaâniche, M., Pataricza, A. (eds.) EDCC 2005. LNCS, vol. 3463, pp. 281–292. Springer, Heidelberg (2005). https://doi.org/10.1007/11408901_21
118. Xie, Y., Aiken, A.: Scalable error detection using Boolean satisfiability. In: Symposium on Principles of Programming Languages (2005)
119. Zalewski, M.: American Fuzzy Loop (2017). http://lcamtuf.coredump.cx/afl/