

**The challenges—and great promise—
of modern symbolic execution techniques,
and the tools to help implement them.**

BY CRISTIAN CADAR AND KOUSHIK SEN

Symbolic Execution for Software Testing: Three Decades Later

SYMBOLIC EXECUTION HAS garnered a lot of attention in recent years as an effective technique for generating high-coverage test suites and for finding deep errors in complex software applications. While the key idea behind symbolic execution was introduced more than three decades ago,^{6,12,23} it has only recently been made practical, as a result of significant advances in constraint satisfiability,¹⁶ and of more scalable dynamic approaches that combine concrete and symbolic execution.^{9,19}

Symbolic execution is typically used in software testing to explore as many different program paths as possible in a given amount of time, and for each path to generate a set of concrete input values exercising it, and

check for the presence of various kinds of errors including assertion violations, uncaught exceptions, security vulnerabilities, and memory corruption. The ability to generate concrete test inputs is one of the major strengths of symbolic execution: from a test generation perspective, it allows the creation of high-coverage test suites, while from a bug-finding perspective, it provides developers with a concrete input that triggers the bug, which can be used to confirm the error independently of the symbolic execution tool that generated it.

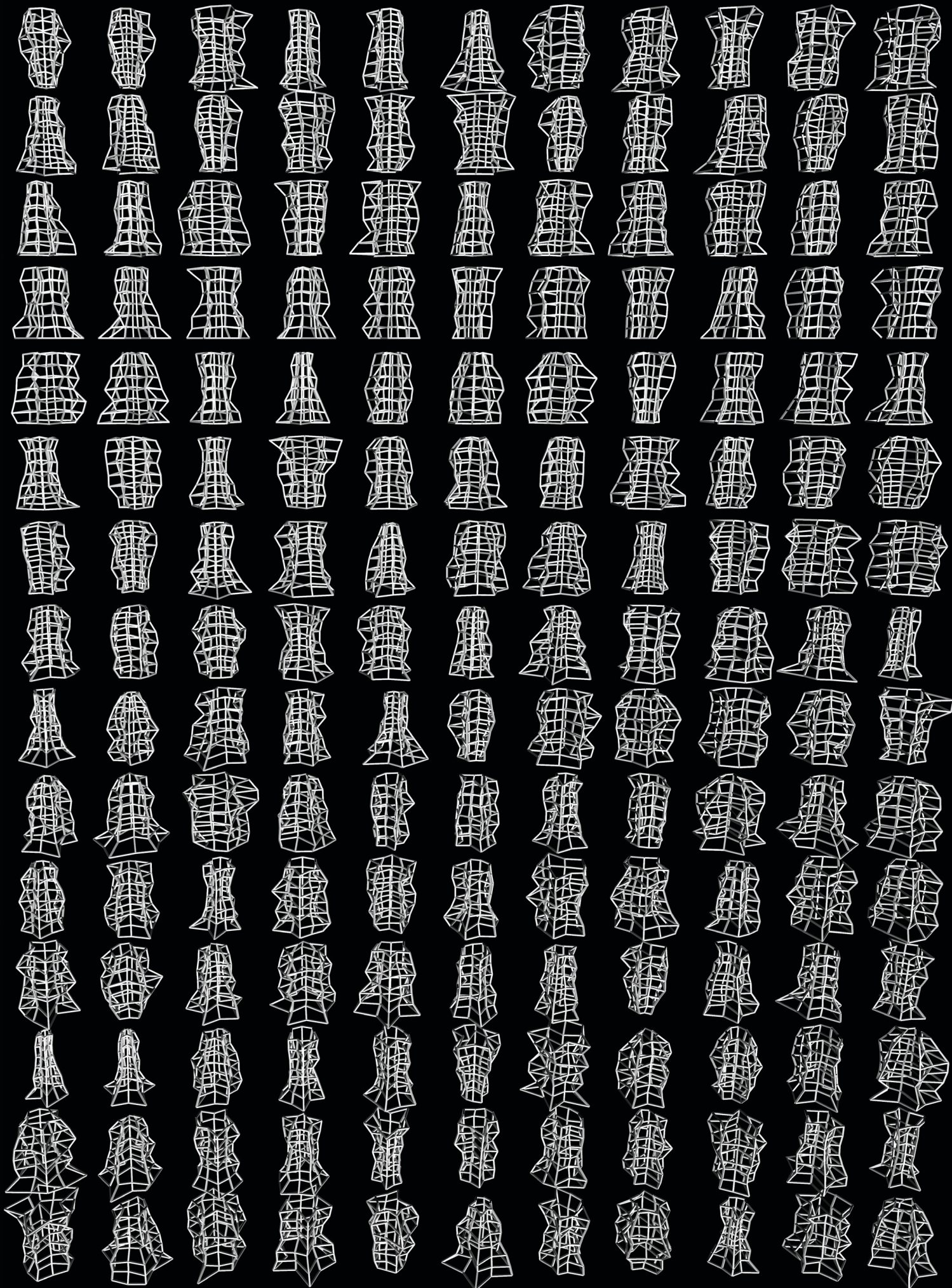
Furthermore, note that in terms of finding errors on a given program path, symbolic execution is much more powerful than traditional dynamic execution techniques such as those implemented by popular tools like Valgrind²⁸ or Purify,²¹ because it has the ability to find a bug if there are *any* buggy inputs on that path, rather than depending on having a concrete input that triggers the bug.

Finally, unlike other program analysis techniques, symbolic execution is not limited to finding generic errors such as buffer overflows, but can reason about higher-level program properties, such as complex program assertions.

This article gives an overview of symbolic execution by showing how it

» key insights

- **Modern symbolic execution techniques provide an effective way to automatically generate test inputs for real-world software. Such inputs can achieve high test coverage and find corner-case bugs such as buffer overflows, uncaught exceptions, and assertion violations.**
- **Symbolic execution works by exploring as many program paths as possible in a given time budget, creating logical formula encoding the explored paths, and using a constraint solver to generate test inputs for feasible execution paths.**
- **Modern symbolic execution techniques mix concrete and symbolic execution and benefit from significant advances in constraint solving to alleviate limitations which prevented traditional symbolic execution from being useful in practice for about 30 years.**




works on a simple example and highlighting its main features. We describe a couple of modern approaches to symbolic execution that make it effective for real-world software. Then, we explore the main challenges of symbolic execution, including path explosion, constraint solving, and memory modeling. Finally, we present several representative symbolic execution tools. Note that we do not aim to provide here a comprehensive survey of existing work in the area, but instead choose to illustrate some of the main challenges and proposed solutions by using examples from the authors' own work.


Overview of Classical Symbolic Execution

The key idea behind symbolic execution^{6,12,23} is to use *symbolic values*, instead of concrete data values, as input values, and to represent the values of program variables as *symbolic expressions* over the symbolic values. As a result, the output values computed by a program are expressed as a function of the input symbolic values. In software testing, symbolic execution is used to generate a test input for each feasible execution path of a program. A feasible execution path is a sequence of `true` and `false`, where a value of `true` (respectively `false`) at the i^{th} position in the sequence denotes that the i^{th} conditional statement encountered along the execution path took the “then” (respectively the “else”) branch. All the feasible execution paths of a program can be represented using a tree, called the *execution tree*. For example, the function `testme()` in Figure 1 has three feasible execution paths, which form the execution tree shown in Figure 2. These paths can be executed, for instance, by running the program on the inputs $\{x = 0, y = 1\}$, $\{x = 2, y = 1\}$ and $\{x = 30, y = 15\}$. The goal of symbolic execution is to generate such a set of inputs so that all the feasible execution paths (or as many as possible in a given time budget) can be explored exactly once by running the program on those inputs.

Symbolic execution maintains a symbolic state σ , which maps variables to symbolic expressions, and a symbolic path constraint (or path condition) PC , which is a quantifier-free first-order formula over symbolic expressions. At the beginning of a symbolic execution,



Unlike other program analysis techniques, symbolic execution is not limited to finding generic errors such as buffer overflows, but can reason about higher-level program properties, such as complex program assertions.



σ is initialized to an empty map and PC is initialized to `true`. Both σ and PC are populated during the course of symbolic execution. At the end of a symbolic execution along a feasible execution path of the program, PC is solved using a constraint solver to generate concrete input values. If the program is executed on these concrete input values, it will take exactly the same path as the symbolic execution and terminate in the same way.

For example, symbolic execution of the code in Figure 1 starts with an empty symbolic state and with symbolic path constraint `true`. At every read statement `var = sym_input()` that receives program input, symbolic execution adds the mapping $var \mapsto s$ to σ , where s is a fresh symbolic value. For example, symbolic execution of the first two lines of the `main()` function (lines 16–17) results in $\sigma = \{x \mapsto x_0, y \mapsto y_0\}$, where x_0, y_0 are two initially unconstrained symbolic values. At every assignment $v = e$, symbolic execution updates σ by mapping v to $\sigma(e)$, the symbolic expression obtained by evaluating e in the current symbolic state. For example, after executing line 6, $\sigma = \{x \mapsto x_0, y \mapsto y_0, z \mapsto 2y_0\}$.

At every conditional statement `if (e) S1 else S2`, PC is updated to $PC \wedge \sigma(e)$ (“then” branch), and a fresh path constraint PC' is created and initialized to $PC \wedge \neg \sigma(e)$ (“else” branch). If PC is satisfiable for some assignment of concrete to symbolic values, then symbolic execution continues along the “then” branch with the symbolic state σ and symbolic path constraint PC . Similarly, if PC' is satisfiable, then another instance of symbolic execution is created with symbolic state σ and symbolic path constraint PC' , which continues the execution along the “else” branch; note that unlike in concrete execution, both branches can be taken, resulting in two execution paths. If any of PC or PC' is not satisfiable, symbolic execution terminates along the corresponding path. For example, after line 7 in the example code, two instances of symbolic execution are created with path constraints $x_0 = 2y_0$ and $x_0 \neq 2y_0$, respectively. Similarly, after line 8, two instances of symbolic execution are created with path constraints $(x_0 = 2y_0) \wedge (x_0 > y_0 + 10)$ and $(x_0 = 2y_0) \wedge (x_0 \leq y_0 + 10)$, respectively.

If a symbolic execution instance hits an exit statement or an error (for example, the program crashes or violates an assertion), the current instance of symbolic execution is terminated and a satisfying assignment to the current symbolic path constraint is generated, using an off-the-shelf constraint solver. The satisfying assignment forms the *test inputs*: if the program is executed on these concrete input values, it will take exactly the same path as the symbolic execution and terminate in the same way. For example, on our example code we get three instances of symbolic executions that result in the test inputs $\{x = 0, y = 1\}$, $\{x = 2, y = 1\}$, and $\{x = 30, y = 15\}$, respectively.

Symbolic execution of code containing loops or recursion may result in an infinite number of paths if the termination condition for the loop or recursion is symbolic. For example, the code in Figure 3 has an infinite number of feasible execution paths, where each feasible execution path is either a sequence of an arbitrary number of true's followed by a false or a sequence of infinite number of true's. The symbolic path constraint of a path with a sequence of n true's followed by a false is:

$$\left(\bigwedge_{i \in [1, n]} N_i > 0\right) \wedge (N_{n+1} \leq 0)$$

where each N_i is a fresh symbolic value, and the symbolic state at the end of the execution is $\{N \mapsto N_{n+1}, \text{sum} \mapsto \sum_{i \in [1, n]} N_i\}$. In practice, one needs to put a limit on the search (for example, a timeout, or a limit on the number of paths, loop iterations, or exploration depth).

A key disadvantage of classical symbolic execution is that it cannot generate an input if the symbolic path constraint along a feasible execution path contains formulas that cannot be (efficiently) solved by a constraint solver (for example, nonlinear constraints). Consider performing symbolic execution on two variants of the code in Figure 1: in one variant, we modify the `twice` function as in Figure 4; in the other variant, we assume that the code of `twice` is not available. Let us assume that our constraint solver cannot handle non-linear arithmetic. For the first variant, symbolic execution will generate the path constraints $x_0 \neq (y_0 y_0) \% 50$ and $x_0 = (y_0 y_0) \% 50$ after the execution

Figure 1. Simple example illustrating symbolic execution.

```

1  int twice (int v) {
2      return 2*v;
3  }
4
5  void testme (int x, int y) {
6      z = twice (y);
7      if (z == x) {
8          if (x > y+10)
9              ERROR;
10         }
11     }
12 }
13
14 /* simple driver exercising testme () with sym inputs */
15 int main() {
16     x = sym_input ();
17     y = sym_input ();
18     testme (x, y);
19     return 0;
20 }

```

of the first conditional statement. For the second variant, symbolic execution will generate the path constraints $x_0 \neq \text{twice}(y_0)$ and $x_0 = \text{twice}(y_0)$, where `twice` is an uninterpreted function. Since the constraint solver cannot solve any of these constraints, symbolic execution will fail to generate any input for the modified programs. We next describe two modern symbolic execution techniques that alleviate this problem and generate at least some inputs for the modified programs.

Modern Symbolic Execution Techniques

One of the key elements of modern symbolic execution techniques is their ability to mix concrete and symbolic execution. We present here two such extensions, and then discuss the main advantages they provide.

Concolic Testing. Directed Automated Random Testing (DART)¹⁹ or concolic testing³⁵ performs symbolic execution dynamically, while the program is executed on some concrete input values. Concolic testing maintains a concrete state and a symbolic state: the concrete state maps all variables to their concrete values; the symbolic state only maps variables that have non-concrete values. Unlike classical symbolic execution, since concolic execution maintains the entire concrete state of the program along an execution, it needs initial concrete values for its inputs. Concolic testing executes a program starting with some given or

Figure 2. Execution tree for the example in Figure 1.

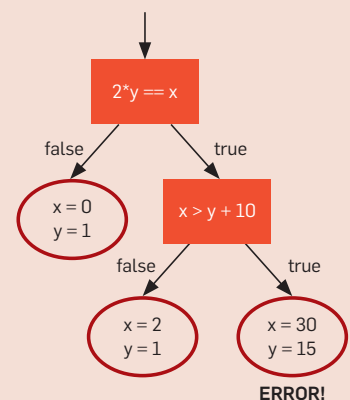


Figure 3. Simple example illustrating an infinite number of feasible execution paths.

```

1  void testme_inf() {
2      int sum = 0;
3      int N = sym_input ();
4      while (N > 0) {
5          sum = sum + N
6          N = sym_input ();
7      }
8  }

```

Figure 4. Simple modification of the example in Figure 1. The function `twice` now performs some non-linear computation.

```

1  int twice (int v) {
2      return (v*v) % 50;
3  }

```

random input, gathers symbolic constraints on inputs at conditional statements along the execution, and then uses a constraint solver to infer variants of the previous inputs in order to steer the next execution of the program toward an alternative feasible execution path. This process is repeated systematically or heuristically until all feasible execution paths are explored or a user-defined coverage criteria is met.

For the example in Figure 1, concolic execution will generate some random input, say $\{x = 22, y = 7\}$, and execute the program both concretely and symbolically. The concrete execution will take the “else” branch at line 7 and the symbolic execution will generate the path constraint $x_0 \neq 2y_0$ along the concrete execution path. Concolic testing negates a conjunct in the path constraint and solves $x_0 = 2y_0$ to get the test input $\{x = 2, y = 1\}$; this new input will force the program execution along a different execution path. Concolic testing repeats both concrete and symbolic execution on this new test input. The execution takes a path different from the previous one—the “then” branch at line 7 and the “else” branch at line 8 are now taken in this execution. As in the previous execution, concolic testing also performs symbolic execution along this concrete execution and generates the path constraint $(x_0 = 2y_0) \wedge (x_0 \leq y_0 + 10)$. Concolic testing will generate a new test input that forces the program along an execution path that has not been previously executed. It does so by negating the conjunct $(x_0 \leq y_0 + 10)$ and solving the constraint $(x_0 = 2y_0) \wedge (x_0 > y_0 + 10)$ to get the test input $\{x = 30, y = 15\}$. The program reaches the ERROR statement with this new input. After this third execution of the program, concolic testing reports that all execution paths of the program have been explored and terminates test input generation. Note that in this example, concolic testing explores all the execution paths using a depth-first search strategy; however, one could employ other strategies to explore paths in different orders, as we discuss later.

Execution-Generated Testing (EGT). The EGT approach,⁹ implemented and extended by the EXE¹⁰ and KLEE⁸ tools, works by making a distinction between the concrete and symbolic state of a program. To this end, EGT intermixes

concrete and symbolic execution by dynamically checking before every operation if the values involved are all concrete. If so, the operation is executed just as in the original program. Otherwise, if at least one value is symbolic, the operation is performed symbolically, by updating the path condition for the current path. For example, if line 17 in Figure 1 is changed to $y = 10$, then line 6 will simply call function `twice()` with the concrete argument 10, call which will be executed as in the original program (note that `twice` could perform an arbitrarily complex operation on its input, but this would not place any additional strain on symbolic execution, because the call will be executed concretely). Then, the branch on line 7 will become `if (20 == x)`, and execution will be forked, one instance adding the constraint that $x = 20$ and following the “then” branch, and the other adding the constraint that $x \neq 20$ and following the “else” branch. Note that on the “then” branch, the conditional at line 8 becomes `if (x > 20)`, and therefore its “then” side is infeasible because x is constrained to have value 20 on this path.

Imprecision vs. completeness in concolic testing and EGT. One of the key advantages in mixing concrete and symbolic execution is that imprecision in symbolic execution (due to, for example, interaction with external code, or constraint solving timeouts), can be alleviated using concrete values (and in the case of concolic testing, also randomization).

For example, real applications almost always interact with the outside world, for instance, by calling libraries that are not instrumented for symbolic execution, or by issuing OS system calls. If all the arguments passed to such a call are concrete, the call can be simply performed concretely, as in the original program. However, even if some operands are symbolic, EGT and concolic testing can use one of the possible concrete values of the symbolic arguments: in EGT this is done by solving the current path constraint for a satisfying assignment, while concolic testing can immediately use the concrete runtime values of those inputs from the current concolic execution.

Besides external code, imprecision in symbolic execution creeps into many

other places—such as unhandled instructions (for example, floating-point) or complex functions that cause constraint solver timeouts—and the use of concrete values allows symbolic execution to recover from that imprecision, albeit at the cost of missing some feasible paths, and thus sacrificing completeness.

To illustrate, we describe the behavior of concolic testing on the version of our running example in which the function `twice` returns the non-linear value $(v*v)\%50$ (see Figure 4). Let us assume that concolic testing generates the random input $\{x = 22, y = 7\}$. Then, the symbolic execution will generate the symbolic path constraint $x_0 \neq (y_0 y_0)\%50$ along the concrete execution path on this input. If we assume that the constraint solver cannot handle non-linear constraints, then concolic testing will fail to generate an input for an alternate execution path. We get a similar situation if the source code for the function `twice` is not available (for example, `twice` is some third-party closed-source library function or a system call), in which case the path constraint becomes $x_0 \neq \text{twice}(y_0)$, where `twice` is an uninterpreted function. Concolic testing handles this situation by replacing some of the symbolic values with their concrete values so that the resultant constraints are simplified and can be solved. For instance, in the example, concolic testing replaces y_0 by its concrete value 7. This simplifies the path constraint in both program versions to $x_0 \neq 49$. By solving the path constraint $x_0 = 49$, concolic testing generates the new input $\{x = 49, y = 7\}$ for a previously unexplored execution path. Note that classical symbolic execution cannot perform this simplification because the concrete state is not available during symbolic execution.

EGT can handle this situation in a similar way: when it encounters the statement `return (v*v) % 50` or the external call `z = twice(y)`, it will call the constraint solver on the current symbolic path constraint to generate a satisfying assignment to y_0 , say $y_0 = 7$, replace this value in the symbolic state and in the path constraint, and continue the execution in a partial symbolic state $\{x \mapsto x_0, y \mapsto 7\}$. The tool KLEE optimizes this by keeping a counterexample cache (described later).

Concolic testing and EGT's approach to simplify constraints using concrete values help them generate test inputs for execution paths for which symbolic execution gets stuck, but this approach comes with a caveat: due to simplification, concolic testing and EGT could lose completeness, that is, they may not be able to generate test inputs for some feasible execution paths. For instance, in our example both techniques will fail to generate an input for the path `true, false`. However, this is clearly preferable to the alternative of simply aborting execution when unsupported statements or external calls are encountered.

Challenges and Extensions

Here, we discuss the main challenges in symbolic execution, and some interesting solutions and extensions developed in response to them.

Path Explosion. One of the key challenges of symbolic execution is the huge number of program paths in all but the smallest programs, which is usually exponential in the number of static branches in the code. However, note that symbolic execution explores only feasible paths that depend on the symbolic input, which reduces the number of conditionals that spawn new paths. For example, in several experiments on testing a number of medium-sized applications we found that less than 42% of the executed statements depend on the symbolic input, and often less than 20% of the symbolic branches encountered during execution have both sides feasible.¹⁰

Despite this implicit filtering, path explosion represents one of the biggest challenges facing symbolic execution, and given a fixed time budget, it is critical to explore the most relevant paths first. Here, we present a representative selection of the techniques developed to address this problem.

Search heuristics. The main mechanism used by symbolic execution tools to prioritize path exploration is the use of search heuristics. Most heuristics focus on achieving high statement and branch coverage, but they could also be employed to optimize other desired criteria. We describe here several coverage-optimized search heuristics successfully used by current symbolic

Path explosion represents one of the biggest challenges facing symbolic execution.

execution tools.

One particularly effective approach is to use the static control-flow graph (CFG) to guide the exploration toward the path closest (as measured statically using the CFG) from an uncovered instruction.^{7,8} A similar approach, described in Cadar et al.,¹⁰ is to favor statements that were run the fewest number of times.

As another example, heuristics based on random exploration have also proved successful.^{7,8} The main idea is to start from the beginning of the program, and at each symbolic branch for which both sides are feasible to randomly choose which side to explore. Note that this random strategy has a number of important advantages: compared to randomly choosing a path to execute, it avoids starvation when a part of the program rapidly forks many new paths; and compared to randomly generating inputs, it has a higher probability to reach branches that are covered by a very small fraction of the inputs. Furthermore, this strategy favors paths early in the execution, with fewer constraints on the inputs, and thus on reaching new program statements.

Interleaving random and symbolic execution. Another successful approach, which was explored in the context of concolic testing, is to interleave symbolic exploration with random testing.²⁶ This approach combines the ability of random testing to quickly reach deep execution states, with the power of symbolic execution to thoroughly explore states in a given neighborhood.

Pruning redundant paths. An alternative approach to avoid exploring the same lines of code over and over again is to automatically prune redundant paths during exploration. The key insight behind the RWset technique described in Boonstoppel et al.⁵ is that if a program path reaches the same program point with the same symbolic constraints as a previously explored path, then this path will continue to execute exactly the same from that point on and thus can be discarded. This technique is enhanced by an important optimization: when comparing the constraints on the two execution paths, it discards those that depend only on values that will not be subsequently read by the program. Note that the effect of pruning these paths can be


significant, as the number of new paths spawned by the continued execution can be exponential in the number of encountered branches.

Lazy test generation. Lazy test generation²⁷ is an approach similar to the counterexample-guided refinement paradigm from static software verification. The technique first explores, using concolic execution, an abstraction of the function under test by replacing each called function with an unconstrained input. Second, for each (possibly spurious) trace generated by this abstraction, it attempts to expand the trace to a concretely realizable execution by recursively expanding the called functions and finding concrete executions in the called functions that can be stitched together with the original trace to form a complete program execution. Thus, it reduces the burden of symbolic reasoning about interprocedural paths to reasoning about intraprocedural paths (in the exploration phase), together with a localized and constrained search through functions (in the concretization phase).


Static path merging. One simple approach that can be used to reduce the number of paths explored is to merge them statically using select expressions that are then passed directly to the constraint solver.¹³ For example, the statement $x[i] = x[i] > 0 ? x[i] : -x[i]$ can be encoded as $(x[i] = \text{select}(x[i] > 0, x[i], -x[i]))$. If such an expression is computed inside a loop statement with N iterations, this approach can reduce the number of explored paths from 2^N to 1. While merging can be effective in many cases, it is unfortunately passing the complexity to the constraint solver, which as discussed in the next section represents another major challenge of symbolic execution.

Constraint Solving. Despite significant advances in constraint solving technology during the last few years—which made symbolic execution practical in the first place—constraint solving continues to be one of the main bottlenecks in symbolic execution, where it often dominates runtime. In fact, one of the main reasons for which symbolic execution fails to scale on some programs is that their code is generating queries that are blowing up the solver.

As a result, it is essential to implement constraint-solving optimizations



It is essential to implement constraint-solving optimizations that exploit the type of constraints generated during the symbolic execution of real programs.



that exploit the type of constraints generated during the symbolic execution of real programs. We present here two representative optimizations used by existing symbolic execution tools.

Irrelevant constraint elimination. The vast majority of queries in symbolic execution are issued in order to determine the feasibility of taking a certain branch side. For example, in the concolic variant of symbolic execution, one branch predicate of an existing path constraint is negated and then the resulting constraint set is checked for satisfiability in order to determine if the program can take the other side of the branch, corresponding to the negated constraint. An important observation is that in general a program branch depends only on a small number of program variables, and therefore on a small number of constraints from the path condition. Thus, one effective optimization is to remove from the path condition those constraints that are irrelevant in deciding the outcome of the current branch. For example, let the path condition be $(x + y > 10) \wedge (z > 0) \wedge (y < 12) \wedge (z - x = 0)$ and suppose we want to generate a new input by solving $(x + y > 10) \wedge (z > 0) \wedge \neg(y < 12)$, where $\neg(y < 12)$ is the negated branch condition whose feasibility we are trying to establish. Then it is safe to eliminate the constraint on z , because this constraint cannot influence the outcome of the $y < 12$ branch. The solution of this reduced constraint set will give new values for x and y , and we use the value of z from the current execution to generate the new input. More formally, the algorithm computes the transitive closure of all the constraints on which the negated constraint depends, by looking whether they share any variables between them. The extra complication is in dealing with pointer dereferences and array indexing, which is discussed in detail in Cadar et al.¹⁰ and Sen et al.³⁵

Incremental solving. One important characteristic of the constraint sets generated during symbolic execution is that they are expressed in terms of a fixed set of static branches from the program source code. For this reason, many paths have similar constraint sets, and thus allow for similar solutions; this fact can be exploited to improve the speed of constraint solving

by reusing the results of previous similar queries, as done in several systems such as CUTE and KLEE.^{8,35} To illustrate this point, we present one such algorithm, namely the counterexample caching scheme used by KLEE.⁸ In KLEE, all query results are stored in a cache that maps constraint sets to concrete variable assignments (or a special *No solution* flag if the constraint set is unsatisfiable). For example, one mapping in this cache could be $(x + y < 10) \wedge (x > 5) \Rightarrow \{x = 6, y = 3\}$. Using these mappings, KLEE can quickly answer several types of similar queries, involving subsets and supersets of the constraint sets already cached. For example, if a subset of a cached constraint set is encountered, KLEE can simply return the cached solution, because removing constraints from a constraint set does not invalidate an existing solution. Moreover, if a superset of a cached constraint set is encountered, KLEE can quickly check if the cached solution still works, by plugging in those values into the superset. For example, KLEE can quickly check that $\{x = 6, y = 3\}$ is still a valid solution for the query $(x + y < 10) \wedge (x > 5) \wedge (y \geq 0)$, which is a superset of $(x + y < 10) \wedge (x > 5)$. This latter technique exploits the fact that in practice, adding extra constraints often does not invalidate an existing solution.

Memory Modeling. The precision with which program statements are translated into symbolic constraints can have a significant influence on the coverage achieved by symbolic execution, as well as on the scalability of constraint solving. For example, using a memory model that approximates fixed-width integer variables with actual mathematical integers may be more efficient, but on the other hand may result in imprecision in the analysis of code depending on corner cases such as arithmetic overflow—which may cause symbolic execution to miss paths, or explore infeasible ones.

Another example are pointers. On the one end of the spectrum is a system like DART that only reasons about concrete pointers, or systems like CUTE and CREST that support only equality and inequality constraints for pointers, which can be efficiently solved.³⁵ At the other end are systems like EXE, and more recently KLEE and SAGE^{10,17,35} that model pointers using the theory of

arrays with selections and updates implemented by solvers like STP or Z3.^{15,18}

The trade-off between precision and scalability should be determined in light of the code being analyzed (for example, low-level systems code vs. high-level applications code), and the exact performance difference between different constraint solving theories. Note that the trade-off between precision and scalability is possible in modern symbolic execution techniques because we can customize the use of concrete values in symbolic formulas and thereby tune both scalability and precision.

Handling Concurrency. Large real-world programs are often concurrent. Because of the inherent non-determinism of such programs, testing is notoriously difficult. Concolic testing was successfully combined with a variant of partial order reduction to test concurrent programs effectively.^{31–34} This combined method provides one of the first techniques to effectively test concurrent programs with complex data inputs.

Tools

Dynamic symbolic execution has been implemented by several tools from both academia and research labs.^{1,7–10,19,20,35,37} These tools support a variety of languages, including C/C++, Java, and the x86 instruction set, implement several different memory models, target different types of applications, and make use of several different constraint solvers and theories. We discuss here five of these tools, with whom the authors of this article have been involved.

DART, CUTE, and CREST. DART¹⁹ is the first concolic testing tool that combines dynamic test generation with random testing and model checking techniques with the goal of systematically executing all (or as many as possible) feasible paths of a program, while checking each execution for various types of errors. DART was first implemented at Bell Labs for testing C programs, and has inspired many other extensions and tools since.

CUTE (A Concolic Unit Testing Engine) and jCUTE (CUTE for Java)^{31,33,35} extend DART to handle multithreaded programs that manipulate dynamic data structures using pointer opera-

tions. In multithreaded programs, CUTE combines concolic execution with dynamic partial order reduction to systematically generate both test inputs and thread schedules.

CUTE and jCUTE were developed at University of Illinois at the Urbana-Champaign for C and Java programs, respectively. Both tools have been applied to several popular open source software including the java.util library of Sun JDK 1.4.

CREST⁷ is an open source tool for concolic testing of C programs. CREST is an extensible platform for building and experimenting with heuristics for selecting which paths to explore. Since being released as open source in May 2008,³ CREST has been downloaded 1,500+ times and has been used by several research groups. For example, CREST has been employed to build tools for augmenting existing test suites to test newly changed code³⁸ and detect SQL injection vulnerabilities,²⁹ has been modified to run distributed on a cluster for testing a flash storage platform,²² and has been used to experiment with more sophisticated concolic search heuristics.³

Concolic testing has also been studied in different courses at several universities.

EXE and KLEE. EXE¹⁰ is a symbolic execution tool for C designed for comprehensively testing complex software, with an emphasis on systems code. To deal with the complexities of systems code, EXE models memory with bit-level accuracy. This is needed because systems code often treats memory as untyped bytes, and observes a single memory location in multiple ways: for example, by casting signed variables to unsigned, or treating an array of bytes as a network packet, inode, or packet filter through pointer casting. As importantly, EXE provides the speed necessary to quickly solve the constraints generated by real code, through a combination of low-level optimizations implemented in its purposely designed constraint solver STP,^{10,18} and a series of higher-level ones such as caching and irrelevant constraint elimination.

KLEE⁸ is a redesign of EXE, built on top of the LLVM²⁴ compiler infra-

a Available at <http://code.google.com/p/crest/>

structure. Like EXE, it performs mixed concrete/symbolic execution, models memory with bit-level accuracy, employs a variety of constraint solving optimizations, and uses search heuristics to get high code coverage. One of the key improvements of KLEE over EXE is its ability to store a much larger number of concurrent states, by exploiting sharing among states at the object-, rather than at the page-level as in EXE. Another important improvement is its enhanced ability to handle interactions with the outside environment—for example, with data read from the file system or over the network—by providing models designed to explore all possible legal interactions with the outside world.

As a result of these features, EXE and KLEE have been successfully used to check a large number of different software systems, including network servers and tools (Berkeley Packet Filter, Avahi, Bonjour, among others);^{10,36} file systems (ext2, ext3, JFS);³⁹ MINIX device drivers (Sound Blaster 16, Lance, PCI);⁵ Unix utilities (Coreutils, MINIX, Busybox suites);⁸ and computer vision code.¹³ They exposed bugs and vulnerabilities in all of these software systems, and constructed concrete inputs triggering them. For example, EXE generated actual disk images that when mounted under various file systems cause the Linux kernel to panic.³⁹ EXE and KLEE were also able to successfully generate high-coverage regression suites: when run on the 89 stand-alone tools of the Coreutils utility suite, KLEE generates tests achieving on average over 90% line coverage, significantly beating an extensive manual regression suite built incrementally by developers over more than 15 years.

KLEE was open sourced in June 2009.^b The tool has an active user community—with approximately 200 members on the mailing list and growing—and several research groups have built upon it in a variety of areas, ranging from wireless sensor networks³⁰ to automated debugging,⁴⁰ reverse engineering of binary device drivers,¹ exploit generation,² online gaming,⁴ testing and verification for GPUs,²⁵ and deterministic multithreading.¹⁴

^b Available at <http://klee.lvm.org/>

Conclusion

Symbolic execution has become an effective program testing technique, providing a way to automatically generate inputs that trigger software errors ranging from low-level program crashes to higher-level semantic properties; generate test suites that achieve high program coverage; and provide per-path correctness guarantees. While more research is needed to scale symbolic execution to very large programs, existing tools have already proved effective in testing and finding errors in a variety of software, varying from low-level network and operating systems code to higher-level applications code.

Acknowledgments

The EGT, EXE, and KLEE projects are joint work with Dawson Engler and several other researchers.^{5,8–10,13,36,39} Daniel Dunbar is the main author of the KLEE system. The DART and concolic testing projects are joint work with several researchers including Gul Agha, Jacob Burnim, Patrice Godefroid, Nils Klarlund, Rupak Majumdar, and Darko Marinov. C

References

- Anand, S., Păsăreanu, C.S. and Visser, W. JPF-SE: A symbolic execution extension to Java PathFinder. In *Proceedings of TACAS'07*.
- Avgerinos, T., Cha, S.K., Hao, B.L.T. and Brumley, D. AEG: Automatic exploit generation. In *Proceedings of NDSS'11*, (Feb. 2011).
- Baluda, M., Braione, P., Denaro, G. and Pezzè, M. Structural coverage of feasible code. In *Proceedings of AST'10*.
- Bethea, D., Cochran, R. and Reiter, M. Server-side verification of client behavior in online games. In *Proceedings of NDSS'10*, 2010.
- Boonstoppel, P., Cadar, C. and Engler, D. RWset: Attacking path explosion in constraint-based test generation. In *Proceedings of TACAS'08*, (Mar–Apr 2008).
- Boyer, R.S., Elspas, B. and Levitt, K.N. SELECT—A formal system for testing and debugging programs by symbolic execution. *SIGPLAN Not.* 10 (1975), 234–245.
- Burnim, J. and Sen, K. Heuristics for scalable dynamic test generation. In *Proceedings of ASE'08*, (Sept. 2008).
- Cadar, C., Dunbar, D. and Engler, D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of OSDI'08*, (Dec 2008).
- Cadar, C. and Engler, D. Execution generated test cases: How to make systems code crash itself (invited paper). In *Proceedings of SPIN'05*, (Aug 2005).
- Cadar, C., Ganesh, V., Pawlowski, P., Dil, D. and Engler, D. EXE: Automatically generating inputs of death. In *Proceedings of CCS'06*, (Oct–Nov 2006). An extended version appeared in *ACM TISSEC* 12, 2 (2008).
- Chipounov, V. and Candea, G. Reverse engineering of binary device drivers with RevNIC. In *Proceedings of EuroSys'10*, (Apr 2010).
- Clarke, L.A. A program testing system. In *Proceedings of the 1976 Annual Conference*, 488–491.
- Collingbourne, P., Cadar, C. and Kelly, P.H. Symbolic crosschecking of floating-point and SIMD code. In *Proceedings of EuroSys'11*, (Apr 2011).
- Cui, H., Wu, J., Tsai, C. and Yang, J. Stable deterministic multithreading through schedule memoization. In *Proceedings of OSDI'10*.

- De Moura, L. and Björner, N. Z3: An efficient SMT solver. In *Proceedings of TACAS'08*, (Mar–Apr 2008).
- De Moura, L. and Björner, N. Satisfiability modulo theories: introduction and applications. *Commun. ACM* 54, 9 (Sept. 2011), 69–77.
- Elkarablieh, B., Godefroid, P. and Levin, M.Y. Precise pointer reasoning for dynamic test generation. In *Proceedings of ISSTA'09*.
- Ganesh, V. and Dill, D.L. A decision procedure for bit-vectors and arrays. In *Proceedings of CAV'07*, (July 2007).
- Godefroid, P., Klarlund, N. and Sen, K. DART: Directed Automated Random Testing. In *Proceedings of PLDI'05*, (June 2005).
- Godefroid, P., Levin, M., and Molnar, D. Automated whitebox fuzz testing. In *Proceedings of NDSS'08*, (Feb. 2008).
- Hastings, R. and Joyce, B. Purify: Fast detection of memory leaks and access errors. In *Proceedings of Winter USENIX Conference*, 1992.
- Kim, Y., Kim, M., and Dang, N. Scalable distributed concolic testing: A case study on a flash storage platform. In *Proceedings of ICTAC'10*, 199–213.
- King, J.C. Symbolic execution and program testing. *Commun. ACM* 19, 7 (July 1976), 385–394.
- Lattner, C. and Adve, V. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proceedings of CGO'04*, (Mar 2004).
- Li, G., Li, P., Sawaga, G., Gopalakrishnan, G., Ghosh, I. and Rajan, S.P. GKLEE: Concolic verification and test generation for GPUs. In *Proceedings of PPOPP'12*.
- Majumdar, R. and Sen, K. Hybrid concolic testing. In *Proceedings of ICSE'07*, (May 2007).
- Majumdar, R. and Sen, K. Latest: Lazy dynamic test input generation. Technical Report UCB/ECS-2007-36. EECs Department, University of California, Berkeley, Mar. 2007.
- Nethercote, N. and Seward, J. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science* 89, 2 (2003).
- Ruse, M., Sarkar, T. and Basu, S. Analysis & detection of SQL injection vulnerabilities via automatic test case generation of programs. In *Proceedings of SAINT'10*, (July 2010).
- Sasnauskas, R., Link, J.A.B., Alizai, M.H., and Wehrle, K. Kleenet: Automatic bug hunting in sensor network applications. In *Proceedings of IPSN'10*, (Apr 2010).
- Sen, K. Scalable Automated Methods for Dynamic Program Analysis. Ph.D. thesis. University of Illinois at Urbana-Champaign, June 2006.
- Sen, K. and Agha, G. Automated systematic testing of open distributed programs. In *Proceedings of FASE'06*, 2006.
- Sen, K. and Agha, G. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *Proceedings of CAV'06*.
- Sen, K. and Agha, G. A race-detection and flipping algorithm for automated testing of multi-threaded programs. In *Proceedings of HVC*, (2006).
- Sen, K., Marinov, D. and Agha, G. CUTE: A concolic unit testing engine for C. In *Proceedings of ESEC/FSE'05*, (Sept. 2005).
- Song, J., Ma, T., Cadar, C. and Pietzuch, P. Rule-based verification of network protocol implementations using symbolic execution. In *Proceedings of ICCCN'11*, (May 2011).
- Tillmann, N. and de Halleux, J. Pex—White box test generation for .NET. In *Proceedings of TAP'08*, (Apr. 2008).
- Xu, Z., Kim, Y., Kim, M., Rothermel, G. and Cohen, M.B. Directed test suite augmentation: Techniques and trade-offs. In *Proceedings of FSE'10*, (Nov. 2010).
- Yang, J., Sar, C., Twohey, P., Cadar, C. and Engler, D. Automatically generating malicious disks using symbolic execution. In *IEEE Symposium on Security and Privacy*, (May 2006).
- Zamfir, C. and Candea, G. Execution synthesis: A technique for automated software debugging. In *Proceedings of EuroSys'10*, (Apr 2010).

Cristian Cadar (c.cadar@imperial.ac.uk) is a lecturer in the Department of Computing at Imperial College London.

Koushik Sen (ksen@cs.berkeley.edu) is an associate professor in the Department of Electrical Engineering and Computer Science at the University of California, Berkeley.