# Symbolic Execution for Software Testing: Three Decades Later

Cristian Cadar
Imperial College London
c.cadar@imperial.ac.uk

Koushik Sen
University of California, Berkeley
ksen@cs.berkeley.edu

## Abstract

Recent years have witnessed a surge of interest in symbolic execution for software testing, due to its ability to generate high-coverage test suites and find deep errors in complex software applications. In this article, we give an overview of modern symbolic execution techniques, discuss their key challenges in terms of path exploration, constraint solving, and memory modeling, and discuss several solutions drawn primarily from the authors' own work.

***Categories and Subject Descriptors*** D.2.5 [*Testing and Debugging*]: Symbolic execution

***General Terms*** Reliability

## 1. Introduction

Symbolic execution has gathered 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 [5, 12, 22, 25], it has only recently been made practical, as a result of significant advances in constraint satisfiability [14], and of more scalable dynamic approaches which combine concrete and symbolic execution [9, 20]. In this article, we give an overview of modern symbolic execution techniques, and discuss their challenges in terms of path exploration, constraint solving, and memory modeling. 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 key challenges and proposed solutions by using examples drawn primarily from the authors' own work. For a detailed overview of symbolic execution techniques, we refer the reader to previously published surveys in the area, such as [11, 19, 32, 34].



A key goal of symbolic execution in the context of software testing is to explore as many different program paths as possible in a given amount of time, and for each path to (1) generate a set of concrete input values exercising that path, and (2) 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 and debug 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 more powerful than traditional dynamic execution techniques such as those implemented by popular tools like Valgrind or Purify, which depend on the availability of concrete inputs triggering the error. Finally, unlike certain 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.

## 2. Overview of Classical Symbolic Execution

The key idea behind symbolic execution [12, 25] is to use *symbolic values*, instead of concrete data values as input and to represent the values of program variables as *symbolic expressions* over the symbolic input values. As a result, the output values computed by a program are expressed as a function of the symbolic input values. In software testing, symbolic execution is used to generate a test input for each execution path of a program. An execution path is a sequence of `true` and `false`, where a value of `true` (respectively `false`) at the $i^{\text{th}}$ position in the sequence denotes that the $i^{\text{th}}$ conditional statement encountered along the execution path took the "then" (respectively the "else") branch. All the 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 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

```
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 }
```

**Figure 1.** Simple example to illustrate symbolic execution.

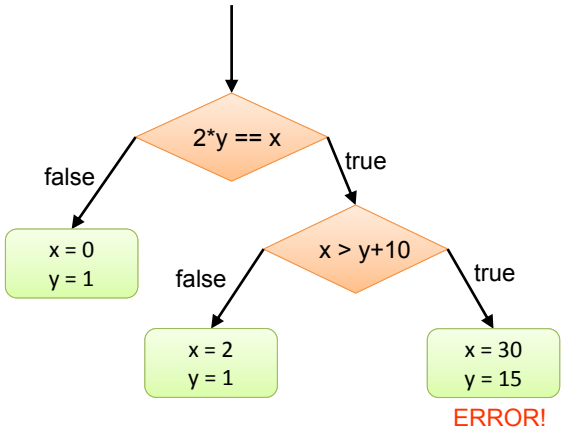


**Figure 2.** Execution tree for the example in Figure 1.

$\{x = 0, y = 1\}, \{x = 2, y = 1\}$ and $\{x = 30, y = 15\}$. The goal is to generate such a set of inputs so that all the execution paths depending on the symbolic input values—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 $\sigma$, which maps variables to symbolic expressions, and a symbolic path constraint $PC$, which is a quantifier-free first-order formula over symbolic expressions. At the beginning of a symbolic execution, $\sigma$ is initialized to an empty map and $PC$ is initialized to *true*. Both $\sigma$ and $PC$ are updated during the course of symbolic execution. At the end of a symbolic execution along an 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 $\sigma$, where $s$ is a fresh (unconstrained) 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 $\sigma$ 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 $\sigma$ and symbolic path constraint $PC$. Similarly, if $PC'$ is satisfiable, then another instance of symbolic execution is created with symbolic state $\sigma$ 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 (e.g., 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 which 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 execution paths, where each execution path is either a sequence of an arbitrary number of `true`s followed by a `false` or a

```
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 3.** Simple example to illustrate infinite number of execution paths.

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

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

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 $\{\texttt{N} \mapsto N_{n+1}, \texttt{sum} \mapsto \sum_{i\in[1,n]} N_i\}$. In practice, one needs to put a limit on the search, e.g., 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 an execution path contains formulas that cannot be (efficiently) solved by a constraint solver. For example, 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 code of the function `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 $x_0 \neq (y_0 y_0) \bmod 50$ and $x_0 = (y_0 y_0) \bmod 50$ after the execution of the first conditional statement. For the second variant, symbolic execution will generate the path constraints $x_0 \neq \texttt{twice}(y_0)$ and $x_0 = \texttt{twice}(y_0)$, 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 which alleviate this problem and generate at least some inputs for the modified programs.

## 3. 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 below two such extensions, and then discuss the key advantages they provide.

**Concolic Testing:** Directed Automated Random Testing (DART) [20], or Concolic testing [37] 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 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 towards an alternative execution path. This process is repeated systematically or heuristically until all execution paths are explored, a user-defined coverage criteria is met, or the time budget expires.

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 discussed in Section 4.1.

**Execution-Generated Testing (EGT):** The EGT approach [9], implemented and extended by the EXE [10] and KLEE [8] 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 `20`, 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 strain on symbolic execution, because the call will be executed concretely). Then, the branch on line 7 will become `if (20 == x)`, and symbolic execution will follow both the "then" side of the branch (where it adds the constraint that $x = 20$), as well as the "else" side (where it adds the constraint that $x \neq 20$). Note that on the "then" path, 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.

Concolic testing and EGT are two instances of modern symbolic execution techniques whose main advantage lies in their ability to mix concrete and symbolic execution. For simplicity, in the rest of the article we will collectively refer to these techniques as "dynamic symbolic execution."

**Imprecision vs. completeness in dynamic symbolic execution:** One of the key advantages in mixing concrete and symbolic execution is that imprecision caused by the interaction with external code or constraint solving timeouts can be alleviated using concrete values.

For example, real applications almost always interact with the outside world, e.g., 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 simply be performed concretely, as in the original program. However, even if some operands are symbolic, dynamic symbolic execution can use one of the possible concrete values of these symbolic operands: in EGT this is done by solving the current path constraint for a satisfying assignment (which can be optimized via the counter-example cache discussed in §4.2), while concolic testing can immediately use the concrete run-time values of those inputs from the current concolic execution.

Besides external code, imprecision in symbolic execution creeps in many other places—such as unhandled operations (e.g., floating-point) or complex functions which cause constraint solver timeouts—and the use of concrete values allows dynamic symbolic execution to recover from that imprecision, albeit at the cost of missing some execution 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 starts with the random input $\{x = 22, y = 7\}$, which generates the symbolic path constraint $x_0 \neq (y_0 y_0) \bmod 50$ along the concrete execution path on this input. If we assume that the constraint solver cannot solve 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 (e.g. `twice` is some third-party closed-source library function or a system call), in which case the path constraint becomes $x_0 \neq \texttt{twice}(y_0)$, 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 by using existing constraint solvers. For example, in the above example, concolic testing replaces $y_0$ by its concrete value 7. This simplifies the path constraint in both programs to $x_0 \neq 49$. By solving the path constraint $x_0 = 49$, concolic testing generates the input $\{x = 49, y = 7\}$ for a previously unexplored execution path. Note that classical symbolic execution cannot easily perform this simplification because the concrete state is not available during symbolic execution.

Dynamic symbolic execution's ability to simplify constraints using concrete values helps it generate test inputs for execution paths for which symbolic execution gets stuck, but this comes with a caveat: due to simplification, it could loose completeness, i.e. they may not be able to generate test inputs for some execution paths. For instance, in our example dynamic symbolic execution 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 operations or external calls are encountered.

## 4. Key Challenges and Some Solutions

We next discuss the key challenges in symbolic execution, and some interesting solutions developed in response to them.

### 4.1 Path Explosion

One of the key challenges of symbolic execution is the huge number of programs paths in all but the smallest programs, which is usually exponential in the number of static branches in the code. As a result, given a fixed time budget, it is critical to explore the most relevant paths first.

First of all, note that symbolic execution implicitly filters out all paths which (1) do not depend on the symbolic input, and (2) are infeasible given the current path constraints. Despite this filtering, path explosion represents one of the biggest challenges facing symbolic execution. There are two key approaches that have been used to address this problem: heuristically prioritizing the exploration of the most promis-

ing paths, and using sound program analysis techniques to reduce the complexity of the path exploration. We discuss each in turn.

**Heuristic techniques.** The key 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.

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 [10], is to favor previously visited statements that were run the fewest number of times.

As another example, heuristics based on random exploration have also proved successful [7, 8]. The key 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.

Another successful approach is to interleave symbolic exploration with random testing [29]. 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.

More recently, symbolic execution was combined with evolutionary search, in which a fitness function is used to drive the exploration of the input space [3, 23, 26, 39]. For example, the Austin tool [26] combines search-based software testing, which uses a suitable fitness function to drive evolutionary search of the test input space, with dynamic symbolic execution to exploit the best of both worlds. Effectiveness of search-based software testing depends on the quality of its fitness function. Several recent promising approaches [3, 23, 39] have exploited concrete state information or symbolic information from dynamic and static analysis to improve fitness functions, which resulted in better test generation. Mutation testing, where the adequacy of a test suite is evaluated by checking its ability to identify various mutations in the program, has also been combined successfully with dynamic symbolic execution [24].

Overall, these novel ways of combining symbolic execution with heuristic search techniques have already shown a lot of promise, and we believe that further advances in this area can play an important role in alleviating the path explosion problem.

**Sound program analysis techniques.** The other key way in which the path explosion problem has been approached was to use various ideas from program analysis and software verification to reduce the complexity of the path exploration in a sound way.

One simple approach that can be used to reduce the number of explored paths is to merge them statically, using *select* expressions that are then passed directly to the constraint solver [13]. While this approach 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.

Compositional techniques improve symbolic execution by caching and reusing the analysis of lower-level functions in subsequent computations [17, 18]. The key idea is to compute function summaries for each tested function—described in terms of pre- and post-conditions over the function's inputs and outputs—and then reuse these summaries in higher-level functions. Lazy test generation [30] is an approach similar to the counterexample-guided refinement paradigm from static software verification. The technique first explores, using dynamic symbolic execution, an abstraction of the function under test by replacing each called function with an unconstrained input.

A related approach to avoid repeatedly exploring the same part of the code is to automatically prune redundant paths during exploration. For example, the RWset technique [4] uses the key insight 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. A similar sound technique works by partitioning inputs into non-interfering blocks which can then be explored separately [31].

### 4.2 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 key bottlenecks in symbolic execution, where it often dominates runtime. In fact, one of the key 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 that exploit the type of constraints generated during the symbolic execution of real programs. We present below 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 for the current execution

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 [10, 15, 37].

**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, 37]. To illustrate this point, we present one such algorithm, namely the counter-example 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.

### 4.3 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 [37]. At the other end are systems like EXE [10], and more recently KLEE [8] and SAGE [15] that model pointers using the theory of arrays with selections and updates implemented by solvers like STP [16] or Z3 [14].

The trade-off between precision and scalability should be determined in light of the code being analyzed (e.g., low-level systems code vs. high-level applications code), and the exact performance difference between different constraint solving theories. In addition, note that in dynamic symbolic execution, one can tune both scalability and precision by customizing the use of concrete values in symbolic formulas.

### 4.4 Handling Concurrency

Large real-world programs are often concurrent. Because of the inherent non-determinism of such programs, testing is notoriously hard. Despite these challenges, dynamic symbolic execution has been effectively used to test concurrent programs, including applications with complex data inputs [35, 36], distributed systems [6, 33], and GPGPU programs [13, 28].

## 5. Tools

Dynamic symbolic execution has been implemented by several tools from both academia and research labs (e.g., [1, 7–10, 20, 21, 37, 38]). 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 below five of these tools, with whom the authors of this article were involved.

### 5.1 DART, CUTE and CREST

DART [20] 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) execution 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) [36, 37] extend DART to handle multi-threaded programs that manipulate dynamic data structures using pointer operations. In multi-threaded 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 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.

CREST [7] 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 (see §4.1). Since being released as open source in May 2008[1], CREST has been downloaded 1500+ times and has been used by several research groups.

### 5.2 EXE and KLEE

EXE [10] 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: e.g., 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, 16], and a series of higher-level ones such as caching and irrelevant constraint elimination.

KLEE [8] is a redesign of EXE, built on top of the LLVM-compiler infrastructure. 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*—e.g., 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, file systems, device drivers and library code.

KLEE was open-sourced in June 2009[2]. The tool has an active user community—with around 200 members on the mailing list and growing—and has been extended by several research groups in a variety of areas.

Several dynamic symbolic execution tools, including KLEE and CREST discussed above, are now available as open-source. These tools have been applied to a variety of applications, including network servers and tools (Berkeley Packet Filter, Avahi, Bonjour); file systems (ext2, ext3, JFS); editors (vi); UNIX utilities (Coreutils, MINIX, Busybox suites); computer vision code (OpenCV), and library code (java.util, Perl Compatible Regular Expressions, libdwarf, libelf). Furthermore, several research teams have built up on top of these tools, and extended them to new problems such as detecting SQL injection vulnerabilities, constructing automatic exploits, testing flash storage platforms, finding errors in wireless sensor networks code, reverse engineering binary device drivers, constructing deterministic multithreaded systems, and testing online games [11].

### 5.3 Other Symbolic Execution Tools in Practice

The industry has also started to adopt symbolic execution. For example, Microsoft has developed SAGE, a dynamic symbolic execution tool for x86 binaries that has discovered several critical bugs in large Windows applications such as image processors and file decoders [21], and also PEX [38], a tool for .NET code, which is now available as a Visual Studio add-in. In addition to Microsoft, several other companies, such as NASA [1], IBM [2], and Fujitsu [27] are developing and using such techniques to test their code.

While the impact of symbolic execution is still limited, the continued stream of innovations in this area will likely increase its applicability and adoption in practice.

## 6. 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 suite that achieve high program coverage; and provide per-path correctness guarantees. While more research is needed in this area, 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 [4, 8–10, 13, 40]. 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.

## References

[1] S. Anand, C. S. Păsăreanu, and W. Visser. JPF-SE: a symbolic execution extension to Java PathFinder. In *TACAS'07*, 2007.

[2] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst. Finding bugs in dynamic web applications. In *ISSTA'08*, July 2008.

[3] A. I. Baars, M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, P. Tonella, and T. E. J. Vos. Symbolic search-based testing. In *ASE'11*, 2011.

[1] Available at `http://code.google.com/p/crest`

[2] Available at `http://klee.llvm.org`

[4] P. Boonstoppel, C. Cadar, and D. Engler. RWset: Attacking path explosion in constraint-based test generation. In *TACAS'08*, Mar–Apr 2008.

[5] R. S. Boyer, B. Elspas, and K. N. Levitt. SELECT – a formal system for testing and debugging programs by symbolic execution. *SIGPLAN Not.*, 10:234–245, 1975.

[6] S. Bucur, V. Ureche, C. Zamfir, and G. Candea. Parallel symbolic execution for automated real-world software testing. In *EuroSys'11*, Apr 2011.

[7] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *ASE'08*, Sept. 2008.

[8] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI'08*, Dec 2008.

[9] C. Cadar and D. Engler. Execution generated test cases: How to make systems code crash itself (invited paper). In *SPIN'05*, Aug 2005.

[10] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. EXE: Automatically generating inputs of death. In *CCS'06*, Oct–Nov 2006. An extended version appeared in ACM TISSEC 12:2, 2008.

[11] C. Cadar, P. Godefroid, S. Khurshid, C. Pasareanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice – preliminary assessment. In *ICSE Impact'11*, May 2011.

[12] L. A. Clarke. A program testing system. In *Proc. of the 1976 annual conference*, pages 488–491, 1976.

[13] P. Collingbourne, C. Cadar, and P. H. Kelly. Symbolic crosschecking of floating-point and SIMD code. In *EuroSys'11*, Apr 2011.

[14] L. De Moura and N. Bjørner. Satisfiability modulo theories: introduction and applications. *Commun. ACM*, 54:69–77, Sept. 2011.

[15] B. Elkarablieh, P. Godefroid, and M. Y. Levin. Precise pointer reasoning for dynamic test generation. In *ISSTA'09*, 2009.

[16] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *CAV'07*, July 2007.

[17] P. Godefroid. Compositional dynamic test generation. In *POPL'07*, Jan. 2007.

[18] P. Godefroid. Higher-order test generation. In *PLDI'11*, 2011.

[19] P. Godefroid, P. de Halleux, M. Y. Levin, A. V. Nori, S. K. Rajamani, W. Schulte, and N. Tillmann. Automated software testing using program analysis. *IEEE Softw.*, 25:30–37, September 2008.

[20] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *PLDI'05*, June 2005.

[21] P. Godefroid, M. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. In *NDSS'08*, Feb. 2008.

[22] W. Howden. Symbolic testing and the DISSECT symbolic evaluation system. *IEEE Transactions on Software Engineering*, 3(4):266–278, 1977.

[23] K. Inkumsah and T. Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *ASE'08*, 2008.

[24] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Trans. Software Eng.*, 37(5):649–678, 2011.

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

[26] K. Lakhotia, P. McMinn, and M. Harman. An empirical investigation into branch coverage for C programs using CUTE and AUSTIN. *J. Systems and Software*, 83(12):2379–91, 2010.

[27] G. Li, I. Ghosh, and S. P. Rajan. KLOVER: a symbolic execution and automatic test generation tool for C++ programs. In *CAV'11*.

[28] G. Li, P. Li, G. Sawaya, and I. Ghosh. GKLEE: Concolic verification and test generation for GPUs. In *PPoPP'12*.

[29] R. Majumdar and K. Sen. Hybrid concolic testing. In *ICSE'07*, May 2007.

[30] R. Majumdar and K. Sen. Latest: Lazy dynamic test input generation. Technical Report UCB/EECS-2007-36, EECS Department, University of California, Berkeley, Mar 2007.

[31] R. Majumdar and R.-G. Xu. Reducing test inputs using information partitions. In *CAV'09*, LNCS, pages 555–569, 2009.

[32] C. S. Pasareanu and W. Visser. A survey of new trends in symbolic execution for software testing and analysis. *STTT*, 11(4):339–353, 2009.

[33] R. Sasnauskas, J. A. B. Link, M. H. Alizai, and K. Wehrle. Kleenet: automatic bug hunting in sensor network applications. In *IPSN'10*, Apr 2010.

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

[35] K. Sen and G. Agha. Automated systematic testing of open distributed programs. In *FASE'06*, 2006.

[36] K. Sen and G. Agha. CUTE and jCUTE : Concolic unit testing and explicit path model-checking tools. In *CAV'06*, 2006.

[37] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *ESEC/FSE'05*, Sep 2005.

[38] N. Tillmann and J. de Halleux. Pex - white box test generation for .NET. In *TAP'08*, Apr 2008.

[39] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *DSN'09*, pages 359–368, 2009.

[40] J. Yang, C. Sar, P. Twohey, C. Cadar, and D. Engler. Automatically generating malicious disks using symbolic execution. In *IEEE Symposium on Security and Privacy*, May 2006.