Test Generation Using Symbolic Execution

Patrice Godefroid

Microsoft Research
pg@microsoft.com

Abstract

This paper presents a short introduction to automatic code-driven test generation using symbolic execution. It discusses some key technical challenges, solutions and milestones, but is not an exhaustive survey of this research area.

1998 ACM Subject Classification D.2.5 Testing and Debugging, D.2.4 Software/Program Verification

Keywords and phrases Testing, Symbolic Execution, Verification, Test Generation

1 Automatic Code-Driven Test Generation

In this paper, we discuss the problem of *automatic code-driven test generation*:

> Given a program with a known set of input parameters, automatically generate a set of input values that will exercise as many program statements as possible.

Variants of this problem definition can be obtained using other code coverage criteria [48]. An optimal solution to this problem is theoretically impossible since this problem is undecidable in general (for infinite-state programs written in Turing-expressive programming languages). In practice, approximate solutions are sufficient.

Although automating test generation using program analysis is an old idea (e.g., [41]), practical tools have only started to emerge during the last few years. Indeed, the expensive sophisticated program-analysis techniques required to tackle the problem, such as symbolic execution engines and constraint solvers, have only become computationally affordable in recent years thanks to the increasing computational power available on modern computers. Moreover, this steady increase in computational power has in turn enabled recent progress in the engineering of more practical software analysis techniques. Specifically, this recent progress was enabled by new advances in dynamic test generation [29], which generalizes and is more powerful than static test generation, as explained later in this paper.

Automatic code-driven test generation differs from *model-based testing*. Given an abstract representation of the program, called *model*, model-based testing consists in generating tests to check the *conformance* of the program with respect to the model. In contrast, code-driven test generation does not use or require a model of the program under test. Instead, its goal is to generate tests that exercise as many program statements as possible, including assertions inserted in the code if any. Another fundamental difference is that models are usually written in abstract formal modeling languages which are, by definition, more amenable to precise analysis and test generation. In contrast, code-driven test generation has to deal with arbitrary software code and systems for which program analysis is bound to be imprecise, as discussed below.

2 Symbolic Execution

Symbolic execution is a program analysis technique that was introduced in the 70s (e.g., see [41, 5, 14, 53, 39]). 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 [15].

One of the earliest proposals for using static analysis as a kind of symbolic program testing method was proposed by King almost 35 years ago [41]. The idea is to symbolically explore the tree of all computations the program exhibits when all possible value assignments to input parameters are considered. For each *control path* ρ, that is, a sequence of control locations of the program, a *path constraint* ϕ_ρ is constructed that characterizes the input assignments for which the program executes along ρ. All the paths can be enumerated by a search algorithm that explores all possible branches at conditional statements. The paths ρ for which ϕ_ρ is satisfiable are *feasible* and are the only ones that can be executed by the actual program. The solutions to ϕ_ρ characterize the inputs that drive the program through ρ. This characterization is exact provided symbolic execution has perfect precision. Assuming that the theorem prover used to check the satisfiability of all formulas ϕ_ρ is sound and complete, this use of static analysis amounts to a kind of symbolic testing.

A prototype of this system allowed the programmer to be presented with feasible paths and to experiment with assertions in order to force new and perhaps unexpected paths. King noticed that assumptions, now called preconditions, also formulated in the logic could be joined to the analysis forming, at least in principle, an automated theorem prover for Floyd/Hoare's verification method, including inductive invariants for programs that contain loops. Since then, this line of work has been developed further in various ways, leading to various strands of program verification approaches, such as *verification-condition generation* (e.g., [20, 37, 17, 3]), *symbolic model checking* [6] and *bounded model checking* [13].

Symbolic execution is also a key ingredient for *precise* automatic code-driven test generation. While program verification aims at proving the absence of program errors, test generation aims at generating concrete test inputs that can drive the program to execute specific program statements or paths.

Work on automatic code-driven test generation using symbolic execution can roughly be partitioned into two groups: *static* versus *dynamic* test generation.

3 Static Test Generation

Static test generation (e.g., [41]) 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.*

Unfortunately, this approach is ineffective whenever the program contains statements involving constraints outside the scope of reasoning of the theorem prover, i.e., statements "that cannot be reasoned about symbolically". This limitation is illustrated by the following example [23]:

```
int obscure(int x, int y) {
  if (x == hash(y)) return -1;   // error
  return 0;                      // ok
}
```

Assume the constraint solver cannot "symbolically reason" about the function `hash` (perhaps because it is too complex or simply because its code is not available). This means that the constraint solver cannot generate two values for inputs `x` and `y` that are guaranteed to

satisfy (or violate) the constraint `x == hash(y)`. In this case, static test generation cannot generate test inputs to drive the execution of the program `obscure` through either branch of the conditional statement: static test generation is *helpless* for a program like this. Note that, for test generation, it is not sufficient to know that the constraint `x == hash(y)` is satisfiable for *some* values of `x` and `y`, it is also necessary to generate *specific values* for `x` and `y` that satisfy or violate this constraint.

The practical implication of this simple observation is significant: static test generation is doomed to perform poorly whenever precise symbolic execution is not possible. Unfortunately, this is frequent in practice due to complex program statements (pointer manipulations, floating-point operations, etc.) and calls to operating-system and library functions that are hard or impossible to reason about symbolically with good enough precision.

4 Dynamic Test Generation

A second approach to test generation is *dynamic test generation* (e.g., [42, 49, 36]): it consists of executing the program P, typically starting with some random 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. This process is repeated until a given final statement is reached or a specific program path is executed.

Directed Automated Random Testing [29], or DART for short, is a recent variant of dynamic test generation that blends it with model checking techniques with the goal of systematically executing *all* feasible program paths of a program while detecting various types of errors using run-time checking tools (like Purify, for instance). In DART, each new input vector attempts to force the execution of the program through *some* new path. By repeating this process, such a *directed search* attempts to force the program to sweep through all its feasible execution paths, in a style similar to *systematic testing* and *dynamic software model checking* [22].

In practice, a directed search typically cannot explore all the feasible paths of large programs in a reasonable amount of time. However, it usually does achieve better coverage than pure random testing and, hence, can find new program bugs.

A key observation [29] is that *imprecision in symbolic execution can be alleviated using concrete values and randomization*: whenever 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 concrete values of those inputs.

Let us illustrate this important point with an example. Consider again the program `obscure` given above. Even though it is impossible to generate two values for inputs `x` and `y` such that the constraint `x == hash(y)` is satisfied (or violated), it is easy to generate, for a fixed value of `y`, a value of `x` that is equal to `hash(y)` since the latter can be observed and known at runtime. By picking randomly and then fixing the value of `y`, we first run the program, observe the concrete value c of `hash(y)` for that fixed value of `y` in that run; then, in the next run, we set the value of the other input `x` either to c or to another value, while leaving the value of `y` unchanged, in order to force the execution of the `then` or `else` branches, respectively, of the conditional statement in the function `obscure`. The DART algorithm does all this automatically [29].

In summary, static test generation is unable to generate test inputs to control the execution of the program `obscure`, while dynamic test generation can *easily* drive the executions of

that same program through all its feasible program paths. In realistic programs, imprecision in symbolic execution typically creeps in in many places, and dynamic test generation allows test generation to recover from that imprecision. Dynamic test generation can be viewed as extending static test generation with additional runtime information, and is therefore more general, precise, and powerful.

5 The Quest for Maximum Precision

Dynamic test generation is the most precise general form of code-driven test generation that is known today. It is more precise than static test generation and other forms of test generation such as random, taint-based and coverage-heuristic-based test generation. It is also the most sophisticated, requiring the use of automated theorem proving for solving path constraints. This machinery is more complex and heavy-weight, but may excerise more paths, find more bugs and generate fewer redundant tests covering the same path. Whether this maximum precision is worth the trouble depends on the application domain.

How much more precise is dynamic test generation compared to static test generation? In [24], it is shown exactly when the "concretization trick" used in the above `hash` example helps, or when it does not help. This is done formally by simulating the process of simplifying complex symbolic expressions using their runtime values using *uninterpreted functions*. Path constraints are then extended with uninterpreted function symbols representing imprecision during symbolic execution. For test generation, it is shown that those uninterpreted function symbols need to be *universally quantified*, unlike variables representing ordinary program inputs which are *existentially quantified*. In other words, this *higher-order* representation of path constraints forces test generation to be done from *validity proofs* of first-order logic formulas with uninterpreted functions, instead of *satisfiability proofs* of quantifier-free logic formulas (without uninterpreted functions) as usual.

The bottom-line is this: the key property of dynamic test generation that makes it more powerful than static test generation is *only* its ability to observe concrete values and to record those in path constraints. In contrast, the process of simplifying complex symbolic expressions using concrete runtime values can be accurately simulated statically using uninterpreted functions. However, those concrete values are necessary to effectively compute new input vectors, a fundamental requirement in test generation [24].

In principle, static test generation can be extended to concretize symbolic values whenever static symbolic execution becomes imprecise [40]. In practice, this is problematic and expensive because this approach not only requires to detect *all* sources of imprecision, but also requires one call to the constraint solver for each concretization to ensure that every synthesized concrete value satisfies prior symbolic constraints along the current program path. In contrast, dynamic test generation avoids these two limitations by leveraging a specific concrete execution as an automatic fall back for symbolic execution [29].

6 Whitebox Fuzzing (The Killer App)

Another significant recent milestone is the emergence of *whitebox fuzzing* [32] as the current main *"killer app"* for dynamic test generation, and arguably for automatic code-driven test generation in general.

Whitebox fuzzing extends dynamic test generation from unit testing to whole-program security testing. There are three main differences. First, inspired by so-called blackbox fuzzing [21], whitebox fuzzing performs dynamic test generation starting from one or several

well-formed inputs, which is a heuristics 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.

Because whitebox fuzzing targets large applications, symbolic execution must scale to very long program executions, and 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* [32]. SAGE uses several optimizations that are crucial for dealing with huge execution traces. SAGE was also the first tool to perform dynamic symbolic execution at the x86 binary level. Working at the x86 binary level allows SAGE 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.

Over the last few years, whitebox fuzzers have found many new security vulnerabilities (buffer overflows) in many Windows [32] and Linux [46] 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 [33], 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.

Since 2008, SAGE has been running *non-stop* on an average of 100+ machines, automatically fuzzing hundreds of applications in Microsoft security testing labs. This is over 400 machine-years and the *largest computational usage ever for any Satisfiability-Modulo-Theories (SMT) solver*, according to the authors of the Z3 SMT solver [16], with over three billion constraints processed to date.

7 Other Related Work

Over the last several years, other tools implementing dynamic test generation have been developed for various programming languages, properties and application domains. Examples of such tools are DART [29], EGT [9], CUTE [58], EXE [10], Catchconv [47], PEX [60], KLEE [8], CREST [7], BitBlaze [59], Splat [45], Apollo [2], and YOGI [35], to name some. Dynamic test generation has become so popular that it is also sometimes casually referred

to as "execution-generated tests" [9], "concolic testing" [58], or simply "dynamic symbolic execution" [60].

All the above tools differ by how they perform symbolic execution (for languages such as C, Java, x86, .NET, etc.), 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, etc.). Indeed, like in traditional static program analysis and abstract interpretation, these important parameters depend in practice on which type of program is to be tested, on how the program interfaces with its environment, and on the property of interest. Moreover, various cost/precision tradeoffs are also possible, as usual in program analysis.

When building tools like these, there are many other challenges, such as: how to recover from imprecision in symbolic execution [29, 24], how to check efficiently many properties together [10, 31], how to leverage grammars (when available) for complex input formats [44, 27], how to deal with path explosion [23, 1, 4, 45, 35], how to precisely reason about pointers [58, 10, 18], how to deal with inputs of varying sizes [61], how to deal with floating-point instructions [28], how to deal with input-dependent loops [55, 34], which heuristics to prioritize the search in the program's search space [10, 32, 7], how to re-use previous analysis results across code changes [51, 30, 52], how to leverage reachability facts inferred by static program analysis [35], etc. Other recent work has also explored how to target other application areas, such as concurrent programs [57], database applications [19], web applications [2, 54], or device drivers [35, 43].

More broadly, many other papers discussing test generation and program verification have been published over the last 30+ years. It would be virtually impossible to survey them all. We only highlighted here some key technical problems and recent milestones. We encourage the reader to consult other recent surveys, such as [11, 56, 50, 25], which present different, yet also partial, points of view.

8 Conclusion

Automatic code-driven test generation aims at proving existential properties of programs: does there exist a test input that can exercise a specific program branch or statement, or follow a specific program path, or trigger a bug? Test generation dualizes traditional program verification and static program analysis aimed at proving universal properties which holds for all program paths, such as "there are no bugs of type X in this program".

Symbolic reasoning about large programs is bound to be imprecise. If perfect bit-precise symbolic reasoning was possible, static program analysis would detect standard programming errors without reporting false alarms. How to deal with this imprecision is a fundamental problem in program analysis. Traditional static program verification builds "may" over-approximations of the program behaviors in order to prove correctness, but at the cost of reporting false alarms. Dually, automatic test generation requires "must" under-approximations in order to drive program executions and find bugs without reporting false alarms, but at the cost of possibly missing bugs.

Test generation is only one way of proving existential reachability properties of programs, where specific concrete input values are generated to exercise specific program paths. More generally, such properties can be proved using so-called *must abstractions* of programs [26], without necessarily generating concrete tests. A must abstraction is defined as a program abstraction that preserves existential reachability properties of the program. Sound path constraints are particular cases of must abstractions [35]. Must abstractions can also be

built backwards from error states using static program analysis [12, 38]. This approach can detect program locations and states provably leading to error states (no false alarms), but may fail to prove reachability of those error states back from whole-program initial states, and hence may miss bugs or report unreachable error states.

Most tools mentioned in the previous section are research prototypes, aimed at exploring new ideas, but they are not used on a daily basis by ordinary software developers and testers. Finding other "killer apps" for these techniques, beyond whitebox fuzzing of file and packet parsers, is *critical* in order to sustain progress in this research area.

References

1. S. Anand, P. Godefroid, and N. Tillmann. Demand-Driven Compositional Symbolic Execution. In *Proceedings of TACAS'2008 (14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems)*, volume 4963 of *Lecture Notes in Computer Science*, pages 367–381, Budapest, April 2008. Springer-Verlag.
2. S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. M. Paradkar, and M. D. Ernst. Finding Bugs in Web Applications Using Dynamic Test Generation and Explicit-State Model Checking. *IEEE Trans. Software Eng.*, 36(4):474–494, 2010.
3. M. Barnett, B. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proceedings of FMCO'2005 (4th International Symposium on Formal Methods for Components and Objects)*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer-Verlag, September 2006.
4. P. Boonstoppel, C. Cadar, and D. Engler. RWset: Attacking path explosion in constraint-based test generation. In *TACAS'08*, April 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. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of LICS'1990 (5th Symposium on Logic in Computer Science)*, pages 428–439, Philadelphia, June 1990.
7. J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *ASE'08*, 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. In *Proceedings of SPIN'2005 (12th International SPIN Workshop on Model Checking of Software)*, volume 3639 of *Lecture Notes in Computer Science*, San Francisco, August 2005. Springer-Verlag.
10. C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically Generating Inputs of Death. In *ACM CCS*, 2006.
11. C. Cadar, P. Godefroid, S. Khurshid, C.S. Pasareanu, K. Sen, N.Tillmann, and W. Visser. Symbolic Execution for Software Testing in Practice – Preliminary Assessment. In *ICSE'2011*, Honolulu, May 2011.
12. S. Chandra, S. J. Fink, and M. Sridharan. Snugglebug: A Powerful Approach to Weakest Preconditions. In *Proceedings of PLDI'2009 (ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation)*, Dublin, June 2009.
13. E. M. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded Model Checking Using Satisfiability Solving. *Formal Methods in System Design*, 19(1):7–34, 2001.
14. L. A. Clarke. A program testing system. In *Proc. of the 1976 annual conference*, pages 488–491, 1976.
15. L. A. Clarke and D. J. Richardson. Applications of symbolic evaluation. *Journal of Systems and Software*, 5(1):15–35, 1985.

16 L. de Moura and N. Bjorner. Z3: An Efficient SMT Solver. In *Proceedings of TACAS'2008 (14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340, Budapest, April 2008. Springer-Verlag.

17 E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18:453–457, 1975.

18 B. Elkarablieh, P. Godefroid, and M.Y. Levin. Precise Pointer Reasoning for Dynamic Test Generation. In *Proceedings of ISSTA'09 (ACM SIGSOFT International Symposium on Software Testing and Analysis)*, pages 129–139, Chicago, July 2009.

19 M. Emmi, R. Majumdar, and K. Sen. Dynamic Test Input Generation for Database Applications. In *Proceedings of ISSTA'2007 (International Symposium on Software Testing and Analysis*, pages 151–162, 2007.

20 R. Floyd. Assigning Meaning to Programs. In *Mathematical Aspects of Computer Science*, pages 19–32. XIX American Mathematical Society, 1967.

21 J. E. Forrester and B. P. Miller. An Empirical Study of the Robustness of Windows NT Applications Using Random Testing. In *Proceedings of the 4th USENIX Windows System Symposium*, Seattle, August 2000.

22 P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proceedings of POPL'97 (24th ACM Symposium on Principles of Programming Languages)*, pages 174–186, Paris, January 1997.

23 P. Godefroid. Compositional Dynamic Test Generation. In *Proceedings of POPL'2007 (34th ACM Symposium on Principles of Programming Languages)*, pages 47–54, Nice, January 2007.

24 P. Godefroid. Higher-Order Test Generation. In *Proceedings of PLDI'2011 (ACM SIGPLAN 2011 Conference on Programming Language Design and Implementation)*, pages 258–269, San Jose, June 2011.

25 P. Godefroid, P. de Halleux, M. Y. Levin, A. V. Nori, S. K. Rajamani, W. Schulte, and N. Tillmann. Automating Software Testing Using Program Analysis. *IEEE Software*, 25(5):30–37, September/October 2008.

26 P. Godefroid, M. Huth, and R. Jagadeesan. Abstraction-based Model Checking using Modal Transition Systems. In *Proceedings of CONCUR'2001 (12th International Conference on Concurrency Theory)*, volume 2154 of *Lecture Notes in Computer Science*, pages 426–440, Aalborg, August 2001. Springer-Verlag.

27 P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based Whitebox Fuzzing. In *Proceedings of PLDI'2008 (ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation)*, pages 206–215, Tucson, June 2008.

28 P. Godefroid and J. Kinder. Proving Memory Safety of Floating-Point Computations by Combining Static and Dynamic Program Analysis. In *Proceedings of ISSTA'2010 (ACM SIGSOFT International Symposium on Software Testing and Analysis)*, pages 1–11, Trento, July 2010.

29 P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Proceedings of PLDI'2005 (ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation)*, pages 213–223, Chicago, June 2005.

30 P. Godefroid, S. K. Lahiri, and C. Rubio-Gonzalez. Statically Validating Must Summaries for Incremental Compositional Dynamic Test Generation. In *Proceedings of SAS'2011 (18th International Static Analysis Symposium)*, volume 6887 of *Lecture Notes in Computer Science*, pages 112–128, Venice, September 2011. Springer-Verlag.

31 P. Godefroid, M.Y. Levin, and D. Molnar. Active Property Checking. In *Proceedings of EMSOFT'2008 (8th Annual ACM & IEEE Conference on Embedded Software)*, pages 207–216, Atlanta, October 2008. ACM Press.

32 P. Godefroid, M.Y. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. In *Proceedings of NDSS'2008 (Network and Distributed Systems Security)*, pages 151–166, San Diego, February 2008.

33 P. Godefroid, M.Y. Levin, and D. Molnar. SAGE: Whitebox Fuzzing for Security Testing. *Communications of the ACM*, 55(3):40–44, March 2012.

34 P. Godefroid and D. Luchaup. Automatic Partial Loop Summarization in Dynamic Test Generation. In *Proceedings of ISSTA'2011 (ACM SIGSOFT International Symposium on Software Testing and Analysis)*, pages 23–33, Toronto, July 2011.

35 P. Godefroid, A.V. Nori, S.K. Rajamani, and S.D. Tetali. Compositional May-Must Program Analysis: Unleashing The Power of Alternation. In *Proceedings of POPL'2010 (37th ACM Symposium on Principles of Programming Languages)*, pages 43–55, Madrid, January 2010.

36 N. Gupta, A. P. Mathur, and M. L. Soffa. Generating Test Data for Branch Coverage. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering*, pages 219–227, September 2000.

37 C. A. R. Hoare. An Axiomatic Approach to Computer Programming. *Communications of the ACM*, 12(10):576–580, 1969.

38 J. Hoenicke, K. R. M. Leino, A. Podelski, M. Schaf, and Th. Wies. It's doomed; we can prove it. In *Proceedings of 2009 World Congress on Formal Methods*, 2009.

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

40 S. Khurshid, C. Pasareanu, and W. Visser. Generalized Symbolic Execution for Model Checking and Testing. In *TACAS'03*, April 2003.

41 J. C. King. Symbolic Execution and Program Testing. *Journal of the ACM*, 19(7):385–394, 1976.

42 B. Korel. A Dynamic Approach of Test Data Generation. In *IEEE Conference on Software Maintenance*, pages 311–317, San Diego, November 1990.

43 V. Kuznetsov, V. Chipounov, and G. Candea. Testing closed-source binary device drivers with DDT. In *USENIX ATC'10*, June 2010.

44 R. Majumdar and R. Xu. Directed Test Generation using Symbolic Grammars. In *ASE*, 2007.

45 R. Majumdar and R. Xu. Reducing test inputs using information partitions. In *CAV'09*, pages 555–569, 2009.

46 D. Molnar, X. C. Li, and D. Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *Proc. of the 18th Usenix Security Symposium*, Aug 2009.

47 D. Molnar and D. Wagner. Catchconv: Symbolic execution and run-time type inference for integer conversion errors, 2007. UC Berkeley EECS, 2007-23.

48 G. J. Myers. *The Art of Software Testing*. Wiley, 1979.

49 A. J. Offutt, Z. Jin, and J. Pan. The Dynamic Domain Reduction Procedure for Test Data Generation. *Software Practice and Experience*, 29(2):167–193, 1997.

50 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.

51 S. Person, M. B. Dwyer, S. G. Elbaum, and C. S. Pasareanu. Differential symbolic execution. In *SIGSOFT FSE*, pages 226–237, 2008.

52 S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed Incremental Symbolic Execution. In *PLDI'2011*, pages 504–515, San Jose, June 2011.

53 C.V. Ramamoorthy, S.-B.F. Ho, and W.T. Chen. On the automated generation of program test data. *IEEE Trans. on Software Engineering*, 2(4):293–300, 1976.

54 P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A Symbolic Execution Framework for JavaScript. In *IEEE Symposium on Security and Privacy*, pages 513–528, 2010.

55 P. Saxena, P. Poosankam, S. McCamant, and D. Song. Loop-Extended Symbolic Execution on Binary Programs. In *ISSTA'2009*, pages 225–236, Chicago, July 2009.

56 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.

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

58 K. Sen, D. Marinov, and G. Agha. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of FSE'2005 (13th International Symposium on the Foundations of Software Engineering)*, Lisbon, September 2005.

59 D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *ICISS'2008*, December 2008.

60 N. Tillmann and J. de Halleux. Pex - White Box Test Generation for .NET. In *Proceedings of TAP'2008 (2nd International Conference on Tests and Proofs)*, volume 4966 of *Lecture Notes in Computer Science*, pages 134–153. Springer-Verlag, April 2008.

61 R. Xu, , P. Godefroid, and R. Majumdar. Testing for Buffer Overflows with Length Abstraction. In *Proceedings of ISSTA'08 (ACM SIGSOFT International Symposium on Software Testing and Analysis)*, pages 27–38, Seattle, July 2008.