Symbolic Execution for Software Testing in Practice – Preliminary Assessment

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

Patrice Godefroid
Microsoft Research
pg@microsoft.com

Sarfraz Khurshid
U. Texas at Austin
khurshid@ece.utexas.edu

Corina S. Păsăreanu*
CMU/NASA Ames
corina.s.pasareanu@nasa.gov

Koushik Sen
U.C. Berkeley
ksen@eecs.berkeley.edu

Nikolai Tillmann
Microsoft Research
nikolait@microsoft.com

Willem Visser
Stellenbosch University
visserw@sun.ac.za

ABSTRACT

We present results for the "Impact Project Focus Area" on the topic of symbolic execution as used in software testing. Symbolic execution is a program analysis technique introduced in the 70s that has received renewed interest in recent years, due to algorithmic advances and increased availability of computational power and constraint solving technology. We review classical symbolic execution and some modern extensions such as generalized symbolic execution and dynamic test generation. We also give a preliminary assessment of the use in academia, research labs, and industry.

Categories and Subject Descriptors

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

General Terms

Reliability

Keywords

Generalized symbolic execution, dynamic test generation

1. INTRODUCTION

The ACM-SIGSOFT Impact Project is documenting the impact that software engineering research has had on software development practice. In this paper, we present preliminary results for documenting the impact of research in symbolic execution for automated software testing. Symbolic execution is a program analysis technique that was introduced in the 70s [8, 15, 31, 35, 46], and that has found renewed interest in recent years [9, 12, 13, 28, 29, 32, 33, 40, 42, 43, 50–52, 56, 57].

Symbolic execution is now the underlying technique of several popular testing tools, many of them open-source: NASA's Symbolic (Java) PathFinder[1], UIUC's CUTE and jCUTE[2], Stanford's KLEE[3], UC Berkeley's CREST[4] and BitBlaze[5], etc. Symbolic execution tools are now used in industrial practice at Microsoft (Pex[6], SAGE [29], YOGI[7] and PREfix [10]), IBM (Apollo [2]), NASA and Fujitsu (Symbolic PathFinder), and also form a key part of the commercial testing tool suites from Parasoft and other companies [60].

Although we acknowledge that the impact of symbolic execution in software practice is still limited, we believe that the explosion of work in this area over the past years makes for an interesting story about the increasing impact of symbolic execution since it was first introduced in the 1970s. Note that this paper is not meant to provide a comprehensive survey of symbolic execution techniques; such surveys can be found elsewhere [19, 44, 49]. Instead, we focus here on a few modern symbolic execution techniques that have shown promise to impact software testing in practice.

Software testing is the most commonly used technique for validating the quality of software, but it is typically a mostly manual process that accounts for a large fraction of software development and maintenance. Symbolic execution is one of the many techniques that can be used to automate software testing by automatically generating test cases that achieve high coverage of program executions.

Symbolic execution is a program analysis technique that executes programs with symbolic rather than concrete inputs and maintains a *path condition* that is updated whenever a branch instruction is executed, to encode the constraints on the inputs that reach that program point. Test generation is performed by solving the collected constraints using a constraint solver. Symbolic execution can also be used for bug finding, where it checks for run-time errors or assertion violations and it generates test inputs that trigger those errors.

The original approaches to symbolic execution [8,15,31,35,

[*] We thank Matt Dwyer for his advice

[1] http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc

[2] http://osl.cs.uiuc.edu/~ksen/cute/

[3] http://klee.llvm.org/

[4] http://code.google.com/p/crest/

[5] http://bitblaze.cs.berkeley.edu/

[6] http://research.microsoft.com/en-us/projects/pex/

[7] http://research.microsoft.com/en-us/projects/yogi/

ICSE '11, May 21–28, 2011, Waikiki, Honolulu, HI, USA
Copyright 2011 ACM 978-1-4503-0445-0/11/05 ...$10.00.

46] addressed simple sequential programs with a fixed number of input data of primitive type. Modern approaches, such as generalized symbolic execution (GSE) [33] and jCUTE [51], address multi-threaded programs with complex data structures as inputs. Much of the popularity of symbolic execution applied to large programs is due to recent advances in dynamic test generation [12, 28], extending prior work originating in the 80s and 90s [16, 17, 36] where the symbolic execution is performed at run-time, along concrete program executions. We discuss these techniques in more detail in the next section.

Symbolic execution still suffers from scalability issues due to the large number of paths that need to be analyzed and the complexity of the constraints that are generated. However, algorithmic advances, newly available Satisfiability Modulo Theories (SMT) solvers[8] and more powerful machines have already made it possible to apply such techniques to large programs (with millions lines of code) and to discover subtle bugs in commonly used software – ranging from library code to network and operating systems code – saving millions of dollars (see Section 3).

2. SYMBOLIC EXECUTION

The key idea behind symbolic execution [35] is to use as input values *symbolic values* instead of actual data, and to represent values of program variables as symbolic expressions. As a result, the outputs computed by a program are expressed as a function of the symbolic inputs.

Symbolic execution maintains a symbolic state, which maps variables to symbolic expressions, and a symbolic path constraint PC, a first order quantifier free formula over symbolic expressions. PC accumulates constraints on the inputs that trigger the execution to follow the associated path. At every conditional statement `if (`e`) S1 else S2`, PC is updated with conditions on the inputs to choose between alternative paths. A fresh path condition PC' is created and initialized to $PC \wedge \neg\sigma(e)$ ("else" branch) and PC is updated to $PC \wedge \sigma(e)$ ("then" branch), where $\sigma(e)$ denotes the symbolic predicate obtained by evaluating e in symbolic state σ. Note that unlike in concrete execution, both branches can be taken, resulting in two execution paths. If any of PC or PC' becomes un-satisfiable, symbolic execution terminates along the corresponding path. Satisfiability is checked with a constraint solver.

Whenever symbolic execution along a path terminates (normally or with an error), the current PC is solved and the solution forms the *test inputs*—if the program is executed on these concrete inputs, it will take the same path as the symbolic execution and terminate. Symbolic execution of code containing loops or recursion may result in an infinite number of paths; therefore, in practice, one needs to put a limit on the search, e.g., a timeout or a limit on the number of paths or exploration depth.

2.1 Generalized Symbolic Execution

Generalized symbolic execution (GSE) [33] extends classical symbolic execution with the ability of handling multi-threading and program fragments whose inputs are recursive data structures. GSE performs symbolic execution by leveraging an off-the-shelf model checker, whose built-in capabilities allow handling multi-threading (and other forms of non-determinism). GSE handles input recursive data structures by using *lazy initialization*. GSE starts execution of the method on inputs with *uninitialized* fields and non-deterministically initializes fields when they are first accessed during the method's symbolic execution. This allows symbolic execution of program methods without requiring an a priori bound on the number of input objects. The approach handles input arrays in a similar way. Method preconditions can be used to ensure that fields are initialized to values permitted by the precondition. Partial correctness properties are given as assertions in the program.

On the first access to an un-initialized reference field, GSE non-deterministically initializes it to `null`, to a reference to a new object with un-initialized fields, or to a reference to an object created during a prior initialization step; in this way all the aliasing possibilities in the inputs are treated systematically. Once the field has been initialized, the execution proceeds according to the concrete (non-symbolic) execution semantics. The model-checker systematically handles the non-determinism introduced when creating different heap configurations and when updating path conditions.

2.2 Dynamic Test Generation

Recent work on using symbolic execution for dynamic test case generation—such as Directed Automated Random Testing (DART) [28], EXecution Generated Executions (EGT/EXE) [12, 13] or Concolic Testing (CUTE) [52]—improve classical symbolic execution by making a distinction between the concrete and the symbolic state of a program. The code is essentially run unmodified, and only statements that depend on the symbolic input are treated differently, adding constraints to the current path condition. This dynamic test generation approach has been implemented in various flavors, some of which are discussed in Section 3.

A significant scalability challenge for this technique is how to handle the exponential number of paths in the code. Recent extensions have tried to address this challenge by using heuristics to guide path exploration [13, 29], interleaving symbolic execution with random testing [40], caching function summaries for later use by higher-level functions [26] or eliminating redundant paths by analyzing the values read and written by the program [7].

Dynamic test generation based on symbolic execution has been implemented in a variety of tools [9, 11–13, 28, 29, 41, 52, 57]. We present several of them in the following section.

3. TOOLS AND IMPACT

In this section we present several recent tools that are based on symbolic execution, together with a preliminary assessment of their impact in practice. In the very limited scope of this paper, it is impossible to review here all the relevant tools. Instead, we focus on a few representative ones that implement the different flavors of symbolic execution presented in the previous section. Albeit incomplete, we do hope that this list convinces the reader of the growing impact of symbolic execution in practice.

JPF–SE and Symbolic (Java) PathFinder. The original GSE framework was developed for Java programs and used NASA's Java PathFinder (JPF) model checker as an enabling technology (see JPF–SE [1]), although GSE can be made to work with other model checkers and imperative languages. Since JPF is a general purpose model checker,

[8] `http://www.smtcomp.org/`

GSE benefits from its collection of built-in state space exploration capabilities, such as different search strategies (e.g., heuristic search) as well as partial order and symmetry reductions; (abstract) state matching can be used to avoid performing redundant work [59]. A similar tool [23] uses the Bogor model checking framework, instead of JPF, while yet another approach uses SPIN [54], for checking parallel numeric applications.

GSE originally leveraged JPF using a source-to-source program transformation, but a more recent implementation of GSE, Symbolic PathFinder (SPF) [43], takes a different approach: instead of running the instrumented program on the standard JPF JVM, SPF implements a non-standard interpretation of Java bytecode using a modified JPF JVM, thereby performing symbolic execution more directly. Symbolic JPF stores symbolic information in *attributes* associated with the JPF concrete states, and it supports mixed concrete/symbolic execution.

SPF can analyze both *Java bytecode* and statechart *models*, e.g., Simulink/Stateflow, Standard UML, or Rhapsody UML, via automatic translation into bytecode. SPF can handle mixed integer and real constraints, and complex mathematical constraints, via heuristic solving. A parallel version also exists [56].

SPF is part of the JPF project[9] (open-sourced since 2003) and it has been applied at NASA in various projects, such as test case generation for the Orion control software (where it helped uncover subtle bugs [43]), fault tolerant protocols, NextGen (TSAFE) aviation software or robot executives. SPF has been extended with a symbolic string analysis at Fujitsu, where it is being used for testing web applications[10]. MIT's tool JFuzz [32], a concolic whitebox fuzzer for Java, is built on top of SPF and is freely available from the JPF web site.

DART. DART [28], short for "Directed Automated Random Testing", blends 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 executes a program starting with some given or random concrete inputs, 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 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.

A key observation in DART 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 run-time values of those inputs. In those cases, symbolic execution degrades gracefully by leveraging concrete values into a form of partial symbolic execution. DART was first implemented at Bell Labs for testing C programs, and has inspired many other extensions and tools since.

CUTE and jCUTE. CUTE (A Concolic Unit Testing Engine) and jCUTE (CUTE for Java) [50–52] extends DART to handle multi-threaded programs that manipulates dynamic data structures using pointer operations. CUTE avoids imprecision due to pointer analysis by representing and solving *pointer constraints* approximately. 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 in University of Illinois at Urbana-Champaign for C and Java programs, respectively. Both tools have been applied to test several open-source software including `java.util` library of Sun' JDK 1.4 and bugs detected by these tools have been made available to the developers. Concolic testing has also been studied in different courses at several universities.

CREST. CREST [9] 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 test for programs with far too many executions paths to exhaustively explore. Since being released as an open source in May 2008 [11], CREST has been downloaded 1500+ times and has been used by several research groups. For example, CREST has been used to build tools for augmenting existing test suites to test newly-changed code [62] and for detecting SQL injection vulnerabilities [47], has been modified to run distributed on a cluster for testing a flash storage platform [34], and has been used to experiment with more sophisticated concolic search heuristics [4]. CREST has also been used in teaching courses at few universities.

SAGE: Automated Whitebox Fuzzing. Whitebox fuzzing [29] is a recent approach to *security* testing which extends the scope of systematic dynamic test generation from unit testing to *whole-application* testing. Whitebox fuzzing is able to scale to large file parsers embedded in applications with millions of lines of code and execution traces with billions of machine instructions, such as Microsoft Excel. Several key technical innovations made this possible: new techniques for symbolically executing very long execution traces with billions of program instructions, for symbolic execution at the x86 assembly level, for compact representation of path constraints, new parallel state-space search algorithms like the generational search, and new support in SMT solvers (such as Z3 [22]) for test generation.

Whitebox fuzzing was first implemented in SAGE [29] and since adopted in several other tools, such as CatchConv, Fuzzgrind, Immunity, etc. Over the last couple of years, whitebox fuzzers have found many new security vulnerabilities (buffer overflows) in Windows [29] and Linux [42] applications, including codecs, image viewers and media players. Notably, SAGE found roughly one third of *all* the bugs discovered by file fuzzing during the development of Microsoft's Windows 7 [27], saving millions of dollars by avoiding expensive security patches for nearly a billion PCs worldwide. Since 2008, SAGE has been continually running on an average of 100+ machines automatically "fuzzing" hundreds of applications in a dedicated security testing lab. To date, this represents the largest computational usage ever for any SMT solver, according to the authors of the Z3 SMT solver [22].

[9] `http://babelfish.arc.nasa.gov/trac/jpf`

[10] `http://www.fujitsu.com/global/news/pr/archives/month/2010/20100112-02.html`

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

Pex. Pex [57] implements Dynamic Symbolic Execution to generate test inputs for .NET code, supporting languages such as C#, VisualBasic, and F#. Pex extends the basic approach in several unique ways: While Pex can use concrete values to simplify constraints, Pex usually faithfully represents the semantics of almost all .NET instructions symbolically, including safe and unsafe code, as well as instructions that refer to the object oriented .NET type system, such as type tests and virtual method invocations. Pex uses the SMT solver Z3 [22] to compute models, i.e. test inputs, for satisfiable constraint systems. Pex uses approximations for theories for which Z3 has no precise decision procedures, e.g. for string [6] and floating point arithmetic [38]. Pex supports the generation of test inputs of primitive types as well as (recursive) complex data types, for which Pex automatically computes a factory method which creates an instance of a complex data type by invoking a constructor and a sequence of methods, whose parameters are also determined by Pex. Pex combines several search strategies which select the order in which different execution paths are attempted, in order to achieve high code coverage quickly [61]. In addition to the test case generation capabilities, Pex comes with a mock and stub framework, which makes it easy to write and reuse models for .NET libraries. [21]. Pex enables Parameterized Unit Testing [58], an extension of traditional unit testing.

Pex is a Visual Studio 2010 Power Tool[12]. It is used by several groups within Microsoft. Externally, Pex is available under academic and commercial licenses. The stand-alone Pex tool has been downloaded more than 40,000 times. Anyone can try out Pex in the browser[13], where visitors let Pex analyze more than 250,000 programs within the first five months of the launch of the website.

EXE. EXE [13] 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 [13, 25], and a series of higher-level ones such as caching and irrelevant constraint elimination.

As a result of these features, EXE was able to automatically generate high-coverage test suites, and to discover deep bugs and security vulnerabilities in a variety of complex code, ranging from library code to file systems, packet filters, device drivers and network servers and tools [7, 12, 13, 63].

KLEE. KLEE [11] is a redesign of EXE, built on top of the LLVM [39] 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 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.

KLEE has been open-sourced in June 2009[14]. Since then, it has been downloaded by a variety of users from both the academia and the industry. In particular, it has been used and extended by several research groups, with some of these extensions being contributed back to the main branch. These users have applied KLEE to a variety of areas, ranging from wireless sensor networks [48] to automated debugging [64], reverse engineering and testing of binary device drivers [14, 37], exploit generation [3], online gaming [5], and schedule memoization in multithreaded code [20].

4. CONCLUSION

In this paper we focused on modern symbolic execution techniques that have become popular in recent years, specifically to enable systematic testing for bug finding; we also reviewed the associated tools and their impact in practice. Symbolic execution has also served as a basis for formal verification [18, 24, 30], which is not a focus of this paper.

We outline here some of the challenges to symbolic execution and its wider adoption in software engineering practice. A significant scalability challenge for symbolic execution is how to handle the exponential number of paths in the code. Further advances in compositional techniques [26], pruning redundant paths [7], and heuristics search [9, 40] are needed. Parallelization can also help [13, 29, 53, 56] since the paths generated by symbolic execution can be analyzed independently. Incremental techniques that leverage program differences to focus symbolic execution also hold promise for checking programs as they evolve [45].

Real applications often require solving complex, nonlinear mathematical constraints that are undecidable or very hard to solve; new heuristic techniques are necessary to solve such problems [38, 55]. Test case generation for web applications and security problems requires solving string constraints and combinations of numeric and string constraints. Progress in these areas would significantly extend the impact of symbolic execution to new application domains.

Acknowledgements

Sen's work was supported in part by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227), by NSF Grants CNS-0720906 and CCF-0747390. The CREST tool is developed and maintained by Jacob Burnim. Khurshid's work is supported in part by NSF Grants CCF-0845628, CNS-0958231, and IIS-0438967, and by AFOSR Grant FA9550-09-1-0351.

[12] http://msdn.microsoft.com/en-us/vstudio/bb980963.aspx

[13] http://pexforfun.com

[14] KLEE is available at http://klee.llvm.org.

5. REFERENCES

[1] S. Anand, C. S. Păsăreanu, and W. Visser. JPF-SE: a symbolic execution extension to Java PathFinder. In *TACAS'07*, pages 134–138, 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] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. AEG: Automatic exploit generation. In *NDSS'11*, Feb 2011.

[4] M. Baluda, P. Braione, G. Denaro, and M. Pezzè. Structural coverage of feasible code. In *AST'10*, pages 59–66. ACM, 2010.

[5] D. Bethea, R. Cochran, and M. Reiter. Server-side verification of client behavior in online games. In *NDSS'10*, Feb–Mar 2010.

[6] N. Bjørner, N. Tillmann, and A. Voronkov. Path feasibility analysis for string-manipulating programs. In *TACAS'09*, Mar. 2009.

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

[8] 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.

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

[10] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software–Practice and Experience*, 30(7):775–802, 2000.

[11] 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.

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

[13] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. EXE: Automatically generating inputs of death. In *CCS'06*, Oct–Nov 2006.

[14] V. Chipounov and G. Candea. Reverse engineering of binary device drivers with RevNIC. In *EuroSys'10*, Apr 2010.

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

[16] L. A. Clarke and D. J. Richardson. *Program Flow Analysis: Theory and Application (Symbolic Evaluation Methods for Program Analysis)*. S. Muchnick and N. Jones, Prentice Hall, 1981.

[17] L. A. Clarke and D. J. Richardson. Applications of symbolic evaluation. *Journal of Systems and Software*, 5(1):15–35, 1985.

[18] A. Coen-Porisini, G. Denaro, C. Ghezzi, and M. Pezzé. Using symbolic execution for verifying safety-critical systems. In *ESEC/FSE'01*, 2001.

[19] P. D. Coward. Symbolic execution systems – a review. *Softw. Eng. J.*, 3:229–239, November 1988.

[20] H. Cui, J. Wu, C. che Tsai, and J. Yang. Stable deterministic multithreading through schedule memoization. In *OSDI'10*, 2010.

[21] J. de Halleux and N. Tillmann. Moles: Tool-assisted environment isolation with closures. In *TOOLS'10*, June–July 2010.

[22] L. de Moura and N. Bjorner. Z3: An Efficient SMT Solver. In *TACAS'08*, Mar–Apr 2008.

[23] X. Deng, J. Lee, and Robby. Bogor/Kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems. In *ASE'06*, 2006.

[24] L. K. Dillon. Using symbolic execution for verification of Ada tasking programs. *ACM TOPLAS*, 12, October 1990.

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

[26] P. Godefroid. Compositional dynamic test generation. In *Proc. of the ACM POPL*, Jan 2007.

[27] P. Godefroid. Software model checking improving security of a billion computers. In *SPIN'09*, June 2009.

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

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

[30] L. J. Harrison and R. A. Kemmerer. An interleaving symbolic execution approach for the formal verification of Ada programs with tasking. In *Ada Applications and Environments'88*, 1988.

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

[32] K. Jayaraman, D. Harvison, V. Ganesh, and A. Kiezun. jFuzz: A concolic whitebox fuzzer for Java. In *In NFM'09*, Apr. 2009.

[33] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *TACAS'03*, Apr. 2003.

[34] Y. Kim, M. Kim, and N. Dang. Scalable distributed concolic testing: a case study on a flash storage platform. In *ICTAC'10*, pages 199–213, 2010.

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

[36] B. Korel. A Dynamic Approach of Test Data Generation. In *IEEE Conference on Software Maintenance*, Nov 1990.

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

[38] K. Lakhotia, N. Tillmann, M. Harman, and J. de Halleux. Flopsy - search-based floating point constraint solving for symbolic execution. In *ICTSS'10*, pages 142–157, 2010.

[39] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *CGO'04*, Mar 2004.

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

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

[42] D. Molnar, X. C. Li, and D. Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *USENIX Security'09*, Aug 2009.

[43] C. Pasareanu, P. Mehlitz, D. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *ISSTA'08*, July 2008.

[44] 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.

[45] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. In *PLDI'11 (to appear)*, June 2011. (to appear).

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

[47] M. Ruse, T. Sarkar, and S. Basu. Analysis & detection of sql injection vulnerabilities via automatic test case generation of programs. *IEEE/IPSJ Int. Sym. Applications and the Internet*, 2010.

[48] 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.

[49] 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). May 2010.

[50] K. Sen. *Scalable Automated Methods for Dynamic Program Analysis*. PhD thesis, University of Illinois at Urbana-Champaign, June 2006.

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

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

[53] J. H. Siddiqui and S. Khurshid. ParSym: Parallel symbolic execution. In *ICSTE'10*, Oct. 2010.

[54] S. F. Siegel, A. Mironova, G. S. Avrunin, and L. A. Clarke. Combining symbolic execution with model checking to verify parallel numerical programs. *ACM TOSEM*, 17:10:1–10:34, May 2008.

[55] M. Souza, M. Borges, M. d'Amorim, and C. Păsăreanu. Coral: Solving complex constraints for symbolic pathfinder. In *NFM'11 (to appear)*, Apr. 2011.

[56] M. Staats and C. S. Pasareanu. Parallel symbolic execution for structural test generation. In *ISSTA'10*, July 2010.

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

[58] N. Tillmann and W. Schulte. Parameterized unit tests. In *ESEC/FSE'05*, Sept. 2005.

[59] W. Visser, C. S. Păsăreanu, and R. Pelánek. Test input generation for java containers using state matching. In *ISSTA'06*, July 2006.

[60] T. Xie. Improving automation in developer testing: State of practice. Technical report, North Carolina State University, 2009.

[61] T. Xie, N. Tillmann, P. de Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *DSN'09*, June-July 2009.

[62] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. B. Cohen. Directed test suite augmentation: techniques and tradeoffs. In *FSE'10*, Nov. 2010.

[63] 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.

[64] C. Zamfir and G. Candea. Execution synthesis: A technique for automated software debugging. In *EuroSys'10*, Apr 2010.