Incremental Symbolic Execution for Automated Test Suite Maintenance

Sarmad Makhdoom Muhammad Adeel Khan Junaid Haroon Siddiqui

Department of Computer Science
LUMS School of Science and Engineering
Lahore 54792, Pakistan
{12030019, 09030014, junaid.siddiqui}@lums.edu.pk

ABSTRACT

Scaling software analysis techniques based on source-code, such as symbolic execution and data flow analyses, remains a challenging problem for systematically checking software systems. In this work, we aim to efficiently apply symbolic execution in increments based on versions of code. Our technique is based entirely on dynamic analysis and patches completely automated test suites based on the code changes. Our key insight is that we can eliminate constraint solving for unchanged code by checking constraints using the test suite of a previous version. Checking constraints is orders of magnitude faster than solving them. This is in contrast to previous techniques that rely on inexact static analysis or cache of previously solved constraints. Our technique identifies ranges of paths, each bounded by two concrete tests from the previous test suite. Exploring these path ranges covers all paths affected by code changes up to a given depth bound. Our experiments show that incremental symbolic execution based on dynamic analysis is an order of magnitude faster than running complete standard symbolic execution on the new version of code.

Categories and Subject Descriptors

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

Keywords

Symbolic execution; incremental analysis; KLEE

1. INTRODUCTION

Symbolic execution is a powerful program analysis technique based on a systematic exploration of (bounded) program paths, which was developed over three decades ago [7, 13]. A key idea used in symbolic execution is to build *path conditions*—given a path, a path condition represents a constraint on the input variables, which is a conjunction of the branching conditions on the path. Thus, a solution to a (feasible) path condition is an input that executes the corresponding path. A common application of symbolic execution is indeed to generate test inputs, say to increase code coverage. Automation of symbolic execution requires constraint solvers or decision procedures [3, 9] that can handle the classes of constraints in the ensuing path conditions.

A lot of progress has been made during the last decade in constraint solving technology, in particular SAT [20] and SMT [3, 9] solving. Moreover, raw computation power is now able to support the complexity of solving formulas that arise in a number of real applications. These technological advances have fueled the research interest in symbolic execution, which today not only handles constructs of modern programming languages and enables traditional analyses, such as test input generation [12, 11, 17, 4], but also has non-conventional applications, for example in checking program equivalence [16], in repairing data structures for error recovery [10], and in estimating power consumption [18].

Despite the advances, a key limiting factor of symbolic execution remains its inherently complex path-based analysis. Due to this complex analysis, symbolic execution takes significant time and its use becomes impractical for integrating in development work flow. A key observation, however, is that in a development work flow, code changes in increments and the increments are often very small. Several recent research projects have attempted to build on this idea. Two approaches have been used. One approach is to use static analysis to find differences in the code flow graph (CFG) and either prune symbolic execution by eliminating paths that will not be changed for sure [15] or use heuristics to guide a manual test suite towards code changes [14]. The other approach is based on caching previous results of symbolic analysis so that future runs are faster [22, 21]. The later technique does not utilize the information about code changes directly and relies on the fact that similar code will result in similar path conditions and hence the solving can greatly benefit from caching prior results.

This paper presents a novel technique for incremental symbolic execution using a pure dynamic analysis approach without caching any prior results. Our key insight is that while solving path conditions is costly, generating and checking them is orders of magnitude faster. Instead of performing an inexact static analysis to eliminate generation of path conditions, we generate path conditions and use the prior test suite to check satisfiability. The tests are kept in memory and efficiently divided along different paths of exploration

ASE'14, September 15-19, 2014, Vasteras, Sweden.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3013-8/14/09 ...$15.00.
http://dx.doi.org/10.1145/2642937.2642961.

Program 1 To find middle of three integers

```
1  int mid(int x, int y, int z) {
2    if (x<y) {
3      if (y<z) return y;
4      else if (x<z) return z;
5      else return x;
6    } else if (x<z) return x;
7    else if (y<z) return z;
8    else return y; }
```

to check against the generated path condition. Storing concrete tests and checking their satisfiability is much more efficient than caching path conditions, canonicalizing them, and comparing them.

Paths in changed code do not match any prior tests and have to be fully explored. The first and last such path delineate a *range* that can be explored. This builds on our previous work on *ranged symbolic execution* [19] that introduced the concept of ranges of paths to be explored and used it to enable parallel symbolic execution. Ranged symbolic execution builds on the insight that the *state* of a symbolic execution run can, rather surprisingly, be encoded succinctly by a *test input*—specifically, by the input that executes the last terminating (feasible) path explored by symbolic execution. By defining a fixed *branch exploration ordering*—e.g., taking the true branch before taking the false branch at each non-deterministic branch point during the exploration—an operation already fixed by common implementations of symbolic execution [12, 4, 2], we have that each test input partitions the space of (bounded) paths under symbolic execution into two sets: *explored* paths and *unexplored* paths. Moreover, the branch exploration ordering defines a *linear order* among test inputs; specifically, for any two inputs (that do not execute the same path or lead to an infinite loop), the branching structure of the corresponding paths defines which of the two paths will be explored first by symbolic execution. Thus, an ordered pair of tests, say $\langle \tau, \tau' \rangle$, defines a *range* of (bounded) paths $[\rho_1, \ldots, \rho_k]$ where path ρ_1 is executed by τ and path ρ_k is executed by τ', and for $1 \leq i < k$, path ρ_{i+1} is explored immediately after path ρ_i.

The encoding allows us to separate out the ranges affected by code changes for incremental symbolic execution. We generate and solve path conditions for all test cases in this range and generate new tests in the test suite. For all paths that have a matching test case from prior version, we copy the old test. Any tests that no longer have a matching path get discarded automatically. This enables a totally automated maintenance of symbolic execution generated test suite.

2. ILLUSTRATIVE OVERVIEW

Forward symbolic execution is a technique for executing a program on symbolic values [8, 13]. There are two fundamental aspects of symbolic execution: (1) defining semantics of operations that are originally defined for concrete values and (2) maintaining a *path condition* for the current program path being executed – a path condition specifies necessary constraints on input variables that must be satisfied to execute the corresponding path.

Program 1 contains a function that finds the middle of three integer values. Figure 1 shows the execution tree of

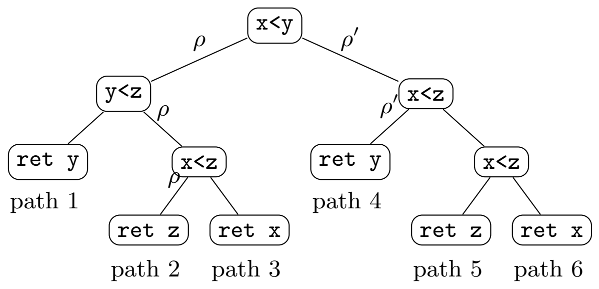

Figure 1: Execution tree of program 1

Program 2 Modified version of mid shown in Program 1

```
1  int mid(int x, int y, int z) {
2    if (x<y) {
3      if (y<z) return y;
4      else if (x<=z) return z;
5      else return x;
6    } else if (x<z) return x;
7    else if (y<z) return z;
8    else return x; }
```

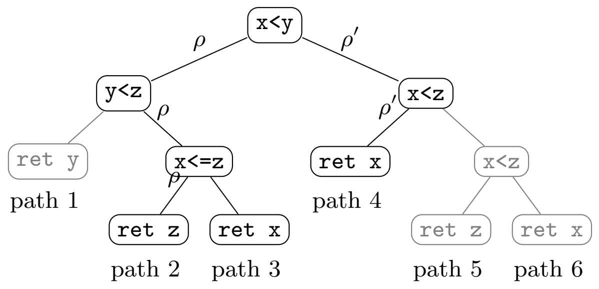

Figure 2: Execution tree of program 2

this program and shows the 6 path conditions explored. The path conditions for each of these paths can be solved using off-the-shelf SAT solvers for concrete tests that exercise the particular path. For example, path 2 can be solved to X=1, Y=3, and Z=2. Following are all the six paths explored:

```
path 1:  [X < Y < Z] L2 -> L3
path 2:  [X < Z < Y] L2 -> L3 -> L4
path 3:  [Z < X < Y] L2 -> L3 -> L4 -> L5
path 4:  [Y < X < Z] L2 -> L6
path 5:  [Y < Z < X] L2 -> L6 -> L7
path 6:  [Z < Y < X] L2 -> L6 -> L7 -> L8
```

Program 2 shows a newer version of the program. Line numbers 4 and 8 are changed in this version of the program. This results in the new execution tree shown in Figure 2. Here we can see the changed path conditions. We need to identify the change in the execution tree in order to perform incremental symbolic execution. We will use our previously generated test suite to find the difference in the execution tree.

We start by analyzing the modified tree while maintaining a list of all tests satisfying the current paths. We start execution that will first take path 1 in the execution tree. We build the path condition and check against satisfying tests. Since this path is unchanged, one test from previous suite will be attached to this path without solving it again. Any branches that are unsatisfied by any test from previous suite are discarded. The second test input will not remain attached to the second path. The program will take some path but it won't be the one we discovered last time. The path condition would be different this time and no test from previous suite will validate with it. We now know a point where path are starting to diverge. This is the starting point of our range that we need to re-examine for the changes.

We reach the point where an old test takes the same exact path generating the old path condition in the program. In this example that would be the path condition 6 where program will be the same on both versions (path 6). At this point, we mark the previous test case as our last point in the range that is affected by code modification. Now after this analysis we have a range that states that the program is changed from this path condition to next path condition. We will complete symbolic execution only on the area identified in the program that will results in 3 new tests. These tests are those, which were invalidated, in the newer version of the program and results in new path conditions. We then use the SAT solver to evaluate them and add these new values to our test suite. We can safely remove our old test inputs that are invalided in the next run and reuse those that are still valid.

3. TECHNIQUE

In this section we will discuss our approach to dynamically figure out the areas in the execution tree that are modified in the newer version of code. Our presentation assumes a standard bounded depth-first symbolic execution where path exploration is systematic and for each condition, the "true" branch is explored before the "false" branch. Such exploration is standard in commonly used symbolic execution techniques, such as generalized symbolic execution using JPF [12], CUTE [17], and KLEE [4].

In our technique, we first symbolically execute the base version of the program, which results in generation of a comprehensive test suite by solving the path conditions. While executing the new version symbolically we use previous test suite. While in the process of exploring states, we compare and validate each new path condition with the solution we already have in the test suite of base version. If we have a successful comparison, we just add that test case to the new test suite. In incremental testing most of the code remains same so is the execution tree. Comparing new path conditions with the old tests is less costly than solving the path condition for the concrete inputs.

Algorithm 1 is used to find and evaluate affected ranges in the program execution tree. It splits test suite on each branch condition in two sets. One set contains all those tests that are satisfiable if that branch evaluated true and other set contains all those test cases that result in false evaluation of branch condition. At this point, if any of the true or false side gets no test cases and all tests go on the other side then that area was infeasible before and need no further exploration. That is first benefit of using our approach in terms of time saving. If any test case is not satisfiable at this point, we start exploring the program for new paths until some other path in the way matches with our previous test inputs.

Algorithm 1: IncrementalExplore to explore new ranges and not solving the present path conditions in the new program

```
Input: A finite set $TestSuite = \{t_1, t_2, \ldots, t_n\}$ of test cases
for each $b$ in $BranchCondition$ do
    $T_{true} \leftarrow split(TestSuite, b)$;
    $T_{false} \leftarrow split(TestSuite, \neg b)$
    $T_{invalid} \leftarrow TestSuite - (T_{true} \cup T_{false})$
    if $T_{invalid} \neq \emptyset$ then
        if $T_{true} = \emptyset$ then
            $exploreAndSolve(T_{true})$
            $IncrementalExplore(T_{false})$
        if $T_{false} = \emptyset$ then
            $exploreAndSolve(T_{false})$
            $IncrementalExplore(T_{true})$
    else
        $IncrementalExplore(T_{true})$
        $IncrementalExplore(T_{false})$
```

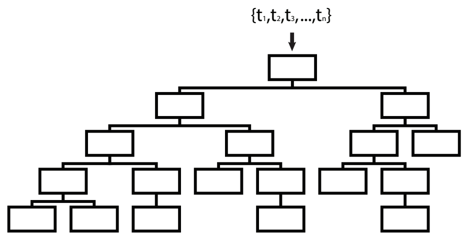

Figure 3: Execution tree of new program without execution

We will discuss it with the help of an example. Figure 3 shows the execution tree of new version of the program which is unknown to the executor before starting execution. We give test cases to the executor before it start exploring the code. On each branching node it will split the test suite into two possible subsets where one set belongs to the left side of the tree and other belongs to the right side. On each branch we also validate the test inputs to see whether they are still satisfiable or not.

In order to understand the concept of finding ranges in the program, lets say one of the test cases reaches the leaf node in the execution tree (as shown in figure 4). We compare and validate this test case, and lets say, it turns out to be successfully validated. We then move to the next test case from the test suite, while executing the next test case we discover that next test case is not validating. These validations fail due to change in the path conditions and the execution tree. We mark this test case as the start of our range; range which needs to be explored for new test inputs.

We keep on testing the program until all other test cases either fail or we do not find any path in the exploration that matches with the one we already have. Lets say, in figure 5, we come across a test case which successfully validates with the current path condition. This is the point where we end our range. So our range starts from the left most test case to the last valid test case as shown in figure 5. That range would have all the new states that need to be explored. And

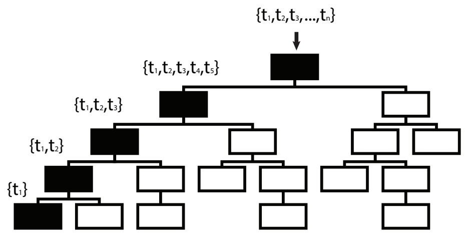

Figure 4: Execution tree of new program with one valid test reaching the leaf

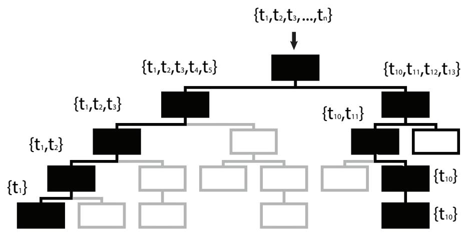
$\{t_1,t_2,t_3,...,t_n\}$
$\{t_1,t_2,t_3,t_4,t_5\}$
$\{t_{10},t_{11},t_{12},t_{13}\}$
$\{t_1,t_2,t_3\}$
$\{t_{10},t_{11}\}$
$\{t_1,t_2\}$
$\{t_{10}\}$
$\{t_1\}$
$\{t_{10}\}$

Figure 5: Range for modified execution tree with two bounded test cases

any tests present in this area should be solved with the solver for concrete inputs. These tests are added to the new test suite because either they were not present or they were invalidated.

We can get such multiple ranges in the execution tree of modified program. As we have already stated that the symbolic execution for incremental testing is in order while exploring the program state space so these ranges would be non-overlapping and we can safely run them in parallel on multiple machines to achieve more speed up for incremental testing.

We modified state-of-the-art tool KLEE [4] that operates on the LLVM [1] bit code for our implementation of our technique. KLEE takes LLVM bit code, performs symbolic analysis, and produces test cases along with their path condition files. Test cases generated by the KLEE contains the actual values to use for symbolic data in order to reproduce the path that KLEE followed. These can be used for obtaining code coverage as well as for reproducing the bugs. We used these test inputs, called seeds, generated by KLEE, as an input test suite to the new program.

If we execute KLEE by providing seeds, it stores them in hash maps. While executing a program symbolically it takes all these seeds side by side. On every branching condition in the program, it divides the seeds for both branches. Any number of seeds that belong to the true side of the branch are associated with it and vice versa. On each step it removes those seeds that are no longer validating with current state of the program. A branch is considered infeasible if all seeds go on the other side.

We took advantage of this seeding system in KLEE to implement our incremental technique. For every branch seeds are compared and divided. If all seeds are validated, it means there is no change in the program. But if any seed fails to validate, it marks our starting point of the range. We start exploring for new states from that point until we find some other seed in the way that was present before and still validating. We modified KLEE so that instead of just checking and moving forward when seeds are supplied, when some seeds fail to validate it starts exploring the program further. We then make new path conditions and solve them in the end for concrete values. These new path conditions and concrete values are added to the new test suite of the program so that they can serve as seeds for the next incremental run.

Comparing and validating the seeds are very easy processes because KLEE uses hashes for seeds as well as the program states, so comparisons would not take much of the computing powers. To validate a seed it just make sure that previous input values for the test case still satisfy the current path condition. This is also not a very CPU intensive job as compared to finding new values by giving path conditions to the solver.

4. EVALUATION

To evaluate incremental symbolic execution we used the GNU core utilities[1] – the basic file, shell and text manipulation core utilities for GNU operating system. Coreutils are medium sized programs between 2000–6000 lines of code. Some of these program performs basic tasks with lot of error checks and thus forms a deep search tree while other performs multiple functions and form a broad search tree. Moreover, because these are included in the every GNU operating systems and they evolve much during the passage of time. We have much iteration available that are the results of bug fixes and feature improvements.

Coreutils were also used in the evaluation of the KLEE [4] symbolic execution tool. As we build incremental symbolic execution using KLEE, Coreutils provide a good benchmark for comparison with KLEE.

We ran KLEE on all Coreutils for one hour and chose the 85 utilities for which KLEE covered maximum paths in this time. We chose version 7.1 as the base version of the Coreutils and then incrementally test one minor update release that is version 7.2.

Figure 6 shows a plot of the execution time of all 85 utilities with and without using our incremental approach on version 7.2 (in incremental seeds of version 7.1 were used). Time taken for each utility with incremental approach is shown as squares in the plot while time taken without using incremental approach is shown as circles. Our incremental approach is about 73% faster and we are saving about 80% solver time by giving 67% less queries.

In regular symbolic execution, major portion of time is consumed at the solver side where it finds the solution of the path condition. In our approach we are reducing the number of queries sent to the solver. Large number of test cases are reused in the process of building new test suite. Only few test cases were solved for concrete inputs using solver. These calls are very less in the incremental testing. This reduced number of calls to the solver is the cause of time saving in the incremental testing.

5. RELATED WORK

Clarke [8] and King [13] pioneered traditional symbolic execution for imperative programs with primitive types. Symbolic execution guided by concrete inputs has been a topic of extensive investigation during the last seven years. DART [11] combines concrete and symbolic execution to collect the

[1]http://www.gnu.org/s/Coreutils

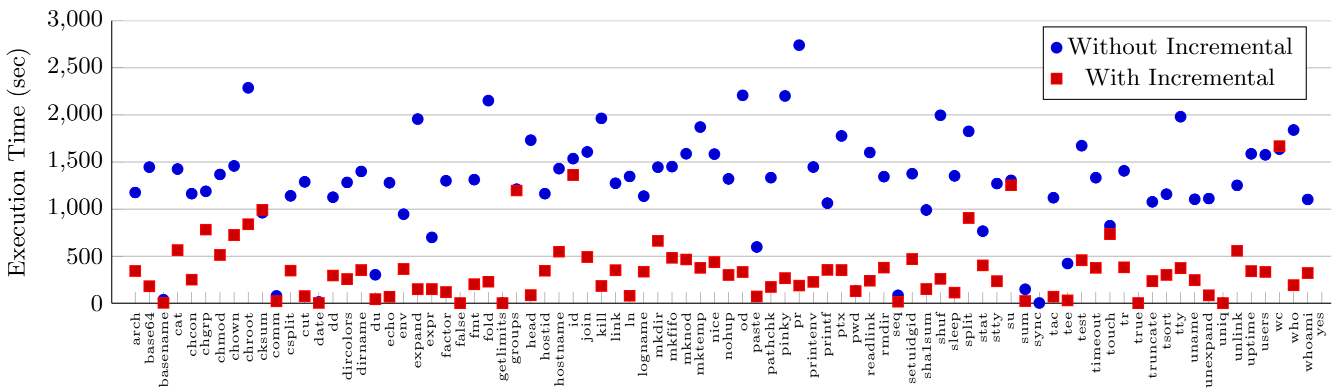

Figure 6: Execution time of 85 program from GNU Coreutils suite of Unix utilities version 7.2 using incremental and non-incremental approach. Square boxes show the time for symbolic execution time by using incremental testing providing seeds from version 7.1. Circle shows the execution time of version 7.2 without using incremental technique.

branch conditions along the execution path. DART negates the last branch condition to construct a new path condition that can drive the function to execute on another path. DART focuses only on path conditions involving integers. CUTE [17] extends DART to handle constraints on references. EGT [5] and EXE [6] also use the negation of branch predicates and symbolic execution to generate test cases. They increase the precision of symbolic pointer analysis to handle pointer arithmetic and bit-level memory locations. KLEE [4] is the most recent tool from the EGT/EXE family. KLEE is open-sourced and has been used by a variety of users in academia and industry. KLEE works on LLVM byte code [1]. It works on unmodified programs written in C/C++ and has been shown to work for many off the shelf programs. Ranged symbolic execution uses KLEE as an enabling technology.

5.1 Incremental Symbolic Execution based on Static Analysis

Directed incremental symbolic execution [15] leverages differences among program versions to optimize symbolic execution of *affected* paths that may exhibit modified behavior. The basic motivation is to avoid symbolically executing paths that have already been explored in a previous program version that was symbolically executed. A reachability analysis is used to identify affected locations, which guide the symbolic exploration.

Directed Incremental Symbolic Execution (DiSE) [15] uses static analysis to find the code blocks that are effected by a change and then uses this information in dynamic analysis to prune the execution tree for symbolic execution. DiSE only generate affected path conditions because it preforms symbolic execution after statically analyzing the both programs CFGs. If a user wants a complete test suite using DiSE, he/she needs to check what path conditions get obsolete, which is not clear how to do. Our proposed technique uses dynamic analysis alone. The range for symbolic execution is formed by validating path conditions generated by a previous execution. The benefit of a pure dynamic technique is that we can estimate more accurately whereas DiSE to over-estimate the effect of a change.

Katch [14] combines static and dynamic analysis for increased coverage of software patches. Katch uses heuristics basic on static analysis to select tests from a manual test suite that are most likely to hit modified code and then it uses heuristics based on dynamic analysis to change the test inputs to increase chances of hitting modified code. The dynamic analysis is built upon symbolic execution. Since the problem of code reachability is undecidable in general, Katch hopes that the heuristics will lead to modified code in most cases. Our idea of incremental symbolic execution does not depend a manual test suite and instead does bounded exhaustive symbolic execution. Our claim is that if the fault can be found using standard symbolic execution, we will be able to find it in less time. Or, in other words, our incremental symbolic execution can explore the program up to a deeper depth bound in the same time due to the time saved by incremental execution.

5.2 Incremental Symbolic Execution based on caching solutions to path constraints

Green [21] is a cache for storing path conditions and their solutions. In Green caching technique they slice the path condition for the purpose of reducing it to be checked for satisfiability, then they perform heuristic based canonization to maximize the chance of finding matches with other path conditions and lastly they store this form in the store. On finding matches they server the results from their store instead of calling the solver. While Green is not an incremental symbolic execution tool in itself, it enables more efficient checking in incremental situations. The reason of that benefit is that in an incremental setting, many path conditions will match from the last execution and thus the solving can be avoided. Path conditions are complex structures and even though Green canonicalizes them, it requires some work to compare them and map the variables. As the cache size increases, comparing becomes more difficult. Validating a path condition, on the other hand, is a quick operation and requires no lookup. Green is, therefore, also orthogonal to our work and can be combined such that our technique find affected ranges using validation and then a given range is executed using Green such that any path conditions that have occurred before need not be solved.

Memoized symbolic execution [22] re-uses results of a previous run of symbolic execution by storing them in a trie-based data structure and re-using them by maintaining and updating the trie in the next run of symbolic execution on the modified program. The savings achieved by Memoise for regression analysis relies on the position of the change, and may vary quite a lot between various kinds of changes. We believe our technique is more effective due to low cost of storing concrete tests versus storing and comparing symbolic execution results in tries.

6. CONCLUSION

In this work, we showed how to scale symbolic execution for efficiently analyzing program increments. The key novelty of our work is to apply symbolic execution in increments with only dynamic analysis. Our approach does not use any form of static analysis or caching of previous results for completely automated incremental testing.

The key idea of our technique is that we can use previously generated test cases for identification of code changes in the new version. Checking constraints is easy compared to solving them. As an enabling technology we leveraged the open-source tool KLEE – a state-of-art tool for symbolic execution. Experimental results using GNU Coreutils show that our approach provides a significant speedup for incremental testing. Incremental execution finished in 73% less time by giving 67% less queries to the solver for 85 programs.

7. REFERENCES

[1] V. Adve, C. Lattner, M. Brukman, A. Shukla, and B. Gaeke. LLVA: A Low-level Virtual Instruction Set Architecture. In *Proc. 36th International Symposium on Microarchitecture (MICRO)*, pages 205–216, 2003.

[2] S. Anand, C. S. Păsăreanu, and W. Visser. JPF-SE: a Symbolic Execution Extension to Java PathFinder. In *Proc. 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 134–138, 2007.

[3] C. Barrett and C. Tinelli. CVC3. In *Proc. 19th International Conference on Computer Aided Verification (CAV)*, pages 298–302, 2007.

[4] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proc. 8th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 209–224, 2008.

[5] C. Cadar and D. Engler. Execution Generated Test Cases: How to make systems code crash itself. In *Proc. International SPIN Workshop on Model Checking of Software* , pages 2–23, 2005.

[6] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically Generating Inputs of Death. In *Proc. 13th Conference on Computer and Communications Security (CCS)*, pages 322–335, 2006.

[7] L. A. Clarke. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Transactions on Software Engineering* , 2(3):215–222, May 1976.

[8] L. A. Clarke. *Test Data Generation and Symbolic Execution of Programs as an aid to Program Validation.* PhD thesis, University of Colorado at Boulder, 1976.

[9] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, 2008.

[10] B. Elkarablieh, I. Garcia, Y. L. Suen, and S. Khurshid. Assertion-based Repair of Complex Data Structures. In *Proc. 22nd International Conference on Automated Software Engineering (ASE)*, pages 64–73, 2007.

[11] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Proc. 2005 Conference on Programming Languages Design and Implementation (PLDI)*, pages 213–223, 2005.

[12] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized Symbolic Execution for Model Checking and Testing. In *Proc. 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 553–568, 2003.

[13] J. C. King. Symbolic Execution and Program Testing. *Communications ACM*, 19(7):385–394, July 1976.

[14] C. C. Paul Dan Marinescu. KATCH: High-Coverage Testing of Software Patches. In *Proc. 12th joint meeting of the European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*, 2013.

[15] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed Incremental Symbolic Execution. In *Proc. 2011 Conference on Programming Languages Design and Implementation (PLDI)*, pages 504–515, 2011.

[16] D. A. Ramos and D. R. Engler. Practical, Low-Effort Equivalence Verification of Real Code. In *Proc. 23rd International Conference on Computer Aided Verification (CAV)*, pages 669–685, 2011.

[17] K. Sen, D. Marinov, and G. Agha. CUTE: A Concolic Unit Testing Engine for C. In *Proc. 5th joint meeting of the European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 263–272, 2005.

[18] C. Seo, S. Malek, and N. Medvidovic. Component-Level Energy Consumption Estimation for Distributed Java-Based Software Systems. In *Proc. 11th International Symposium on Component-Based Software Engineering*, pages 97–113, 2008.

[19] J. H. Siddiqui and S. Khurshid. Scaling Symbolic Execution using Ranged Analysis. In *Proc. 27th Annual Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2012.

[20] N. Sörensson and N. Een. An Extensible SAT-solver. In *Proc. 6th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 502–518, 2003.

[21] W. Visser, J. Geldenhuys, and M. B. Dwyer. Green: Reducing, Reusing and Recycling Constraints in Program Analysis. In *Proc. 2012joint meeting of the European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*, 2012.

[22] G. Yang, C. S. Păsăreanu, and S. Khurshid. Memoized Symbolic Execution. In *Proc. 2012 International Symposium on Software Testing and Analysis (ISSTA)*, ISSTA 2012, pages 144–154, New York, NY, USA, 2012. ACM.