Heuristics for Scalable Dynamic Test Generation

Jacob Burnim
EECS, UC Berkeley
Email: jburnim@cs.berkeley.edu

Koushik Sen
EECS, UC Berkeley
Email: ksen@cs.berkeley.edu

***Abstract*—Recently there has been great success in using symbolic execution to automatically generate test inputs for small software systems. A primary challenge in scaling such approaches to larger programs is the combinatorial explosion of the path space. It is likely that sophisticated strategies for searching this path space are needed to generate inputs that effectively test large programs (by, e.g., achieving significant branch coverage). We present several such heuristic search strategies, including a novel strategy guided by the control flow graph of the program under test. We have implemented these strategies in CREST, our open source concolic testing tool for C, and evaluated them on two widely-used software tools, grep 2.2 (15K lines of code) and Vim 5.7 (150K lines). On these benchmarks, the presented heuristics achieve significantly greater branch coverage on the same testing budget than concolic testing with a traditional depth-first search strategy.**

I. INTRODUCTION

Testing with manually generated inputs is the predominant technique in industry to ensure software quality – such testing accounts for 50–80% of the typical cost of software development. But manual test generation is expensive, error-prone, and rarely exhaustive. Thus, several techniques have been proposed to automatically generate test inputs.

A simple and effective technique for automated test generation is *random testing* [1], [2], [3], [4]. In random testing, the program under test is simply executed on randomly-generated inputs. A key advantage of random testing is that it scales well in the sense that random test input generation takes negligible time. However, random testing is extremely unlikely to test all possible behaviors of a program

A number of symbolic techniques for automated test generation [5], [6] have been proposed to address the limitations of random testing. Such techniques attempt to symbolically execute a program under test along all possible program paths, generating and solving constraints to produce concrete inputs that test each path. Recently, *concolic testing* [7], [8] and a related technique [9] have been proposed which run symbolic execution simultaneously with concrete executions. These approaches are generally more scalable in practice because they can use the concrete program values to reason precisely about complex data structures as well as to simplify intractable constraints.

Although symbolic and concolic techniques have been shown to be very effective in testing smaller programs, these approaches fail to scale to larger programs in which only a tiny fraction of the huge number of possible program paths can be explored. A natural question is how to devise search strategies that could quickly cover a significant portion of the branches in a test program despite searching only a small fraction of the program's path space.

We propose a search strategy that is guided by the static structure of the program under test, namely the control flow graph (CFG). In this strategy, we choose branches to negate for the purpose of test generation based on their distance in the CFG to currently uncovered branches. We experimentally show that this greedy approach to maximizing the branch coverage helps to improve such coverage faster, and to achieve greater final coverage, than the default depth-first search strategy of concolic testing.

We further propose two random search strategies. While in traditional random testing a program is run on random inputs, these two strategies test a program along random execution paths. The second attempts to sample uniformly from the space of possible program paths, while the third is a variant we have found to be more effective in practice.

We have implemented these search strategies in CREST, an open-source prototype test generation tool for C, and experimentally validated the strategies on three benchmarks ranging up to 150K lines of code. Our experiments demonstrate that these search strategies can more effectively search the path space of a test program than either random testing or depth-first concolic search.

II. CONCOLIC SEARCH STRATEGIES

In this section, we contrast our three proposed concolic search strategies with a traditional depth-first search. Due to space constraints, we describe these search strategies by example, leaving the formal details to the accompanying technical report. Also omitted are the now standard details of concolic execution [7], [8].

Figure 1 contains a short program in a C-like imperative language. We use this program as our running example to illustrate the concolic search strategies, treating its two integer inputs x and y as symbolic. For a conditional statement, we call the first statement in the true and false blocks a pair of *branches*. Thus, in the example program the pairs of branches are (l_1, l_2), (l_6, l_7), (l_{10}, l_{13}), and (l_{11}, l_{12}).

A concolic search strategy operates on full, concrete executions through the test program – e.g. $l_0, l_1, l_5, l_7, l_8, l_3, l_9, l_{13}, l_{14}, l_4$, corresponding to a run on inputs $x = 1$, $y = 0$ – along with symbolic path constraints – e.g. $x > y \wedge y \leq 0 \wedge x \neq 4$. For such an execution, a strategy must select one of the alternate branches

```
    main(x, y) {
l0:     if (x > y)
l1:       z = f(y);
        else
l2:       z = y;
l3:     g(x, z);
l4:     return;
    }

    int f(a) {
l5:     if (a > 0)
l6:       ABORT;
        else
l7:       ;
l8:     return -a;
    }
```

```
     g(a, b) {
l9:      if (a == 4)
l10:       if (2*b > 9)
l11:         ABORT;
           else
l12:         ;
         else
l13:       print b-a;
l14:     return;
     }

     not_called(a) {
l15:     b = f(a)
l16:     print 2*b;
l17:     return;
     }
```

Fig. 1. Example program.

the path could have taken – e.g. l_2, l_6, and l_{10} – and then try to solve modified path constraints to find inputs which lead the program down the new branch.

A. Bounded Depth-First Search.

Suppose our initial execution is on inputs $x = 0$, $y = 0$, yielding the concrete execution $P_0 = l_0, l_2, l_3, l_9, l_{13}, l_{14}, l_4$. This execution passes through two conditional statements, and has path constraints $x \leq y \wedge x \neq 4$.

The depth-first search (DFS) first attempts to force the first branch l_2 to l_1, by solving path constraint $x > y$, yielding, e.g., $x = 1$, $y = 0$. Executing on these inputs gives concrete path $P_1 = l_0, l_1, l_5, l_7, l_8, l_3, l_9, l_{13}, l_{14}, l_4$ with path constraints $x > y \wedge y \leq 0 \wedge x \neq 4$.

The DFS recurses on P_1, forcing the second branch along P_1 from l_7 to l_6 by solving $x > y \wedge y > 0$, perhaps yielding $x = 2$, $y = 1$. These inputs give execution $P_2 = l_0, l_1, l_5, l_6$, which reaches the **ABORT** at l_6. The DFS recurses on P_2, but there are no further branches to explore, so this second recursive call immediately returns. Continuing to process P_1, the DFS will attempt to force l_{13} to l_{10}, and will recurse on the resulting path.

The search ends when all feasible program paths have been explored. For a bound $d > 0$, we can also restrict the search to forcing the first d feasible branches along any path. (A branch along a path is *feasible* if we can solve for inputs for which the branch is and is not taken.) Such a search will explore 2^d execution paths, as long as all paths have at least d feasible branches.

B. Control-Flow Directed Search.

The goal of the control-flow directed search strategy is to use the static structure of the program under test to guide the dynamic search of the program's path space. In particular, to achieve high coverage of the test program, we want to guide the search towards paths which reach previously uncovered branches.

Thus, we construct a weighted, static call and control flow graph for the program under test. First, we build the control flow graph (CFG) for each function, giving the edges from a conditional to its two branches weight one and all other edges weight zero. In the example program, the four methods have edges: $(l_0, l_1)^*$, $(l_0, l_2)^*$, (l_1, l_3), (l_2, l_3), (l_3, l_4) and $(l_5, l_6)^*$, $(l_5, l_7)^*$, (l_7, l_8), and $(l_9, l_{10})^*$, $(l_9, l_{13})^*$, $(l_{10}, l_{11})^*$, $(l_{10}, l_{l_1 2})^*$, (l_{12}, l_{14}), (l_{13}, l_{14}) and (l_{15}, l_{16}), (l_{16}, l_{17}), with the starred edges having weight one. Additionally, we add a zero-weight edge from each call site to the called function: (l_1, l_5), (l_3, l_9), and (l_{15}, l_5).

For some set of uncovered or target branches, we can compute with a breadth-first search the minimum distance from every branch to one of the targets. Given an execution of the test program, CfgDirectedSearch tries to force execution down the branches with smallest distances, leading the search towards the uncovered/target branches.

For example, if only branch l_{11} remained uncovered we would assign distance 0 to l_{11}, distance 1 to l_{10}, distance 2 to l_1 and l_2, and infinite distance to l_6, l_7, l_{12}, and l_{13}. Note that the branches in `f` have infinite distance because there are no edges from called functions back to their call sites – we make the simplifying assumption that all called functions return and that we can ignore the branches traversed inside a function when trying to reach a later branch.

Given the above distances and execution $x = 1$, $y = 0$ with path $P_0 = l_0, l_1, l_5, l_7, l_8, l_3, l_9, l_{13}, l_{14}, l_4$ with path constraints $x > y \wedge y \leq 0 \wedge x \neq 4$, the CFG-directed search immediately tries to force branch l_{13} to l_{10} because l_{10} has the minimum distance, 1, among the possible alternate branches l_2, l_6, and l_{10}. Solving $x > y \wedge y \leq 0 \wedge x = 4$ yields e.g. $x = 4$, $y = 0$ and $P_1 = l_0, l_1, l_5, l_7, l_8, l_3, l_9, l_{10}, l_{12}, l_{14}$ and $x > y \wedge y \leq 0 \wedge x = 4 \wedge -2y \leq 9$. The search will then force l_{12} to l_{11}, because l_{11} has distance 0. It will solve $x > y \wedge y \leq 0 \wedge x = 4 \wedge -2y > 9$ to get, e.g., $x = 4$, $y = -5$, which drive the program to **ABORT** at l_{11}.

Unlike depth-first search, the search can skip over the branches in `main` and `f` because it uses the static structure of the program to guide the search more directly towards a relevant part of the path space.

As presented so far, the CFG-directed search is greedy, always forcing execution down the branch with minimal distance to a target. In practice, however, the search may drive execution through a branch l with some distance d, but then find that none of the paths from l to a target branch are feasible. We need mechanisms both for revising our distances for branches – i.e. heuristically updating our local estimates for how hard it is to reach a target branch – and for backtracking or restarting the search. These details can be found in the technical report.

C. Uniform Random Search.

Taking inspiration from the effectiveness of widely-used random testing, in which a program is executed on *random inputs*, we propose a search strategy which executes a program along *random paths*. Such a strategy avoids the problem in random testing that often many inputs are used that lead to the same execution paths and are thus *redundant*. Further, for branches that are reachable by only a very small fraction of the inputs, random execution paths can often cover such branches with much higher probability than random inputs.

Given some path P, the UniformRandomSearch strategy will walk down the path, forcing each branch with probability $1/2$. For example, suppose the initial path is $P_0 = l_0, l_2, l_3, l_9, l_{13}, l_{14}, l_4$, corresponding to inputs $x = 0$, $y = 0$. The search considers the first branch l_2 and flips a coin – if the result is heads it will force the execution from l_2 to l_1. Suppose it is heads. Then, solving the path constraints will give, e.g., $x = 1$, $y = 0$ and new path $P_1 = l_0, l_1, l_5, l_7, l_8, l_3, l_9, l_{13}, l_{14}, l_4$.

The search will then move on to the second branch l_7 (of P_1). Suppose the coin is tails this time, and then heads for the third branch l_{13}. Solving the path constraints yields, e.g., $x = 4$, $y = 0$, and path P_2 through l_{10}: $P_2 = l_0, l_1, l_5, l_7, l_8, l_3, l_9, l_{10}, l_{12}, l_{14}$. Finally, suppose the coin is tails for the final branch l_{12}.

It can be shown that UniformRandomSearch will produce some particular execution with L feasible branches with probability 2^{-L}, running the solver and test program an expected $L/2$ times.

D. Random Branch Search.

Although the previous strategy in a certain sense samples the path space uniformly at random, it requires many runs of the program under test to do so. We found, after trial-and-error, an even simpler random search strategy that is more effective in practice.

In this strategy, RandomBranchSearch, we simply pick one of the branches along the current path at random, and then force the execution to not take the branch. The strategy just repeats this step over and over, possibly with random restarts, taking some random walk through the path space.

III. Evaluation and Implementation

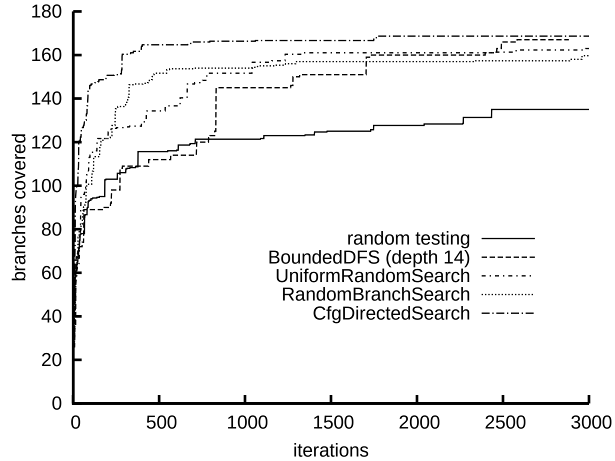
We have implemented our search strategies in CREST, a prototype test generation tool for C. CREST uses CIL [10] to instrument C source files and to extract control-flow graphs, and Yices [11] to solve path constraints. CREST is open source software and is available at http://crest.googlecode.com/.

We experimentally evaluated the effectiveness of our search strategies by running CREST on `replace` (600 lines of C code), the largest program in the Siemens Benchmark Suite, and two popular open-source applications, `grep` 2.2 (15K lines) and `Vim` 5.7 (150K lines).

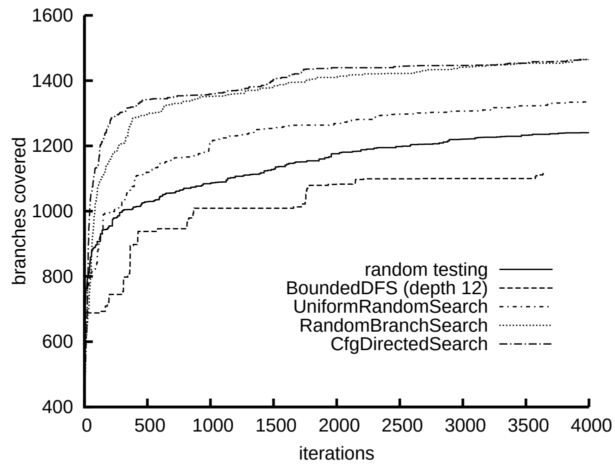
For each benchmark, we compare the performance of the different search strategies over a fixed number of iterations – i.e. runs of the instrumented program. We believe this is an appropriate measure for the testing budget, because, for larger programs, we expect the cost of concrete and symbolic execution to dominate processing done by the strategies themselves.

Experiments were run on 2GHz Core2 Duo servers with 2GB of RAM and running Debian GNU/Linux. All unconstrained variables are initially set to zero.

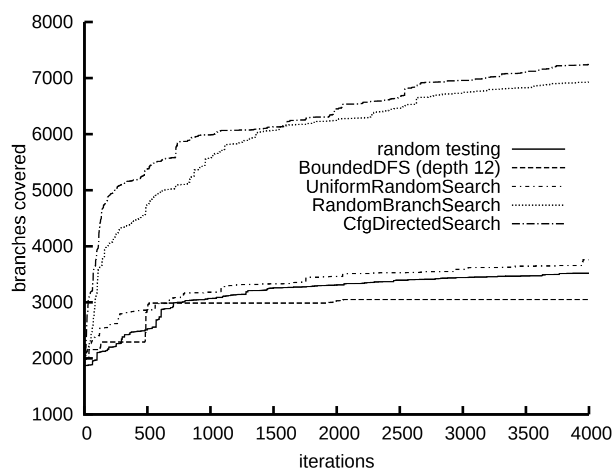
For both `grep` and `Vim`, the way we instrument and run the tested programs (e.g. with fixed-size inputs and fixed



(a) `replace`, with 40 symbolic characters as input (30 for DFS). Contains 200 branches, all of which are reachable. CREST runs 50+ iterations/sec.



(b) `grep` 2.2, with 60 characters as input (45 for DFS). Contains 4184 branches, an estimated 2854 of which are reachable given our testing. CREST runs 40 iterations/sec.



(c) `Vim` 5.7, with 20 characters as input. Contains 39166 branches, an estimated 23400 of which are reachable given our testing. 2-3 sec/iteration.

Fig. 2. Branch coverage achieved on the three benchmarks by the different search strategies and by random testing over a fixed number of iterations. All plots are averages over three runs.

command-line arguments) restricts the set of possible program behaviors. Thus, in addition to reporting *absolute* branch coverage, we report *relative* coverage – the (estimated) fraction of *reachable* branches covered. We estimate the number of reachable branches by summing the branches from each function that was reached by any test run.

A. Replace.

`replace` is a 600-line text processing program, the largest in the Siemens Benchmark Suite [12]. We ran `replace` with 10-symbolic-character source and destination patterns (5 for bounded DFS) and a 20-symbolic-character line of text in which to substitute. A minor optimization is needed for the CFG-directed search on this benchmark, to handle the small program size, the details of which are omitted for space reasons.

As can be seen in Figure 2(a), in a single minute of testing all concolic search strategies were able to cover 80% of the branches in `replace`. In fact, in an additional couple of minutes of testing the best concolic runs achieve 85% or even 90% branch coverage.

B. GNU Grep 2.2

GNU `grep` is a 15K-line open source C program for text search with regular expressions. We instrument `grep` 2.2 to match a length-20 symbolic pattern (length-5 for DFS) against 40 symbolic characters, using all the default matching options.

Figure 2(b) shows that in only a couple of minutes the most effective strategies are able to cover nearly 60% of the estimated reachable branches. In particular, note that the CFG-directed search and the random branch search outperform both random testing and a depth-first concolic search by a significant margin.

C. Vim 5.7.

`Vim` 5.7 is a 150K-line open source text editor. We replace the `safe_vgetc` and `vgetc` functions with one which returns up to 40 characters of symbolic input. These functions provide the inputs to most, but not all modes in Vim. We were thus unable to test Ex mode and several other parts of the editor.

Figure 2(c) shows that in 2-3 hours of testing the most effective search strategies covered nearly a third of the estimated reachable branches. In particular, the CFG-directed search and random branch search achieve more than twice the coverage of the other methods. Further, these two strategies obtain coverage very rapidly, achieving at iterations 100 and 150, respectively, greater coverage than the other strategies do in 4000 iterations.

IV. Conclusions

We believe that a combination of static and dynamic analyses can help automated test generation to achieve significant branch coverage on large software systems. Our experimental results suggest that sophisticated search strategies, particularly those driven by static information such as a programs control flow graph, can enable concolic execution to achieve greater coverage on larger, real-world programs.

Acknowledgments

We would like to thank Caltech UGCS, of the Student Computing Consortium, for providing the computing resources used in this work. This work is supported in part by the NSF Grants CNS-0720906, CCF-0747390, and a gift from Toyota.

References

[1] D. Bird and C. Munoz, "Automatic Generation of Random Self-Checking Test Cases," *IBM Systems Journal*, vol. 22, no. 3, pp. 229–245, 1983.

[2] 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*, 2000.

[3] C. Csallner and Y. Smaragdakis, "JCrasher: an automatic robustness tester for Java," *Software: Practice and Experience*, vol. 34, pp. 1025–1050, 2004.

[4] C. Pacheco and M. D. Ernst, "Eclat: Automatic generation and classification of test inputs," in *19th European Conference Object-Oriented Programming*, 2005.

[5] J. C. King, "Symbolic Execution and Program Testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[6] L. Clarke, "A system to generate test data and symbolically execute programs," *IEEE Trans. Software Eng.*, vol. 2, pp. 215–222, 1976.

[7] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *Proc. of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[8] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," in *5th joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'05)*. ACM, 2005.

[9] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler, "EXE: Automatically generating inputs of death," in *ACM Conference on Computer and Communications Security (CCS 2006)*, 2006.

[10] G. Necula, S. McPeak, S. Rahul, and W. Weimer, "CIL: Intermediate language and tools for analysis and transformation of C programs," in *Proceedings of Conference on Compiler Construction*, 2002.

[11] B. Dutertre and L. M. de Moura, "A fast linear-arithmetic solver for DPLL(T)," in *Computer Aided Verification*, ser. LNCS, vol. 4144, 2006, pp. 81–94.

[12] J. Harrold and G. Rothermel, "Siemens programs, HR variants," http://www.cc.gatech.edu/aristotle/Tools/subjects/.