Heuristics for Scalable Dynamic Test Generation

Jacob Burnim
Koushik Sen

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2008-123

http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-123.html

September 19, 2008

Acknowledgement

Thanks to Caltech's UGCS, of the Student Computing Consortium, for providing the computing resources used in this work. This work is supported in part by the NSF Grant CNS-0720906 and a gift from Toyota.

Heuristics for Scalable Dynamic Test Generation

Jacob Burnim and Koushik Sen
EECS Department, UC Berkeley, USA
{jburnim,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.*

1 Introduction

Testing with manually generated test inputs is the predominant technique in industry to ensure software quality — in fact, such manual testing accounts for 50–80% of the typical cost of software development. However, manual test input generation is expensive, error-prone, and usually not exhaustive.

A simple and effective technique for automated test generation is *random testing* [2, 22, 9, 4, 6, 23] (a.k.a fuzz testing). 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 For example, in the experiments reported in this paper, several hours of random testing covers only 10% of the branches in the Vim editor.

Several symbolic techniques for automated test generation [18, 5, 28, 27, 1, 29] have been proposed to address the limitations of manual and random testing. Such techniques provide better coverage of a program's behavior because they try to generate a single test input for each feasible execution path. In symbolic execution, a program is executed on symbolic inputs: the execution of an assignment statement updates the program state with symbolic expressions and the execution of a conditional statement generates a symbolic constraint in terms of the symbolic inputs. Symbolic techniques then generate concrete inputs that satisfy the symbolic constraints generated along each execution path. Such an input forces the program to take that execution path during normal testing.

Recently, *concolic testing* [12, 24] and a related technique [3] have been proposed as a variant of symbolic execution where symbolic execution is performed simultaneously with concrete execution. Specifically, the program is simultaneously executed on concrete and symbolic values, and symbolic constraints generated along the path are simplified using the corresponding concrete values. The symbolic constraints are then used to incrementally generate test inputs for better path coverage by conjoining symbolic constraints for a prefix of the path with the negation of a conditional taken by the execution. The primary advantage of concolic execution over pure symbolic simulation is the presence of concrete (data and address) values, which can be used both to reason precisely about complex data structures as well as to simplify constraints when they go beyond the capabilities of the underlying constraint solver.

In practice, symbolic techniques and concolic testing have been shown to be very effective in unit testing — often yielding nearly 95-100% branch coverage on programs having 100-2000 lines of code. However, both symbolic and concolic testing fail to scale to large programs. This is because the possible number of execution paths that must be considered symbolically is so large that the methods end up exploring only small part of the program path space. This is unfortunate, as concolic techniques hold much promise for larger and complicated pieces of code for which generating test suites with good coverage is of great importance. A natural question is how to devise search strategies that help to achieve branch coverage quickly despite searching only a small fraction of a program's path space.

A key observation in devising such a search strategy is that the number of execution paths required to get full branch coverage is bounded by the total number of branches in the program, which is both finite and significantly smaller than the total number of feasible execution paths. Therefore, to quickly get branch coverage, rather than attempting to systematically generate test inputs for all feasible execution paths, we should try to explore only those paths that would expose some uncovered branch. This motivates our first proposed search strategy, which is guided by the static structure of a program, namely the control-flow graph. In this strategy, we choose branches to negate for the purpose of test generation based on their distance in the control-flow graph 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. In traditional random testing, we get poor branch coverage because we sample the input space uniformly when generating random inputs; many of these inputs lead the program along the same path, whereas only certain rare inputs may lead the program to a critical corner case. This motivates our second search strategy, which attempts to sample uniformly from the space of execution paths, rather than inputs. Our third strategy is a variant of the second which we have found to be more effective in practice.

We have implemented our search strategies for C programs in a prototype test generation tool, CREST. The tool is extensible, open source, and publicly available. We have experimentally evaluated these strategies on three C benchmarks: (1) `replace`, a 600-line text-processing program and the largest program in the Siemens Benchmark Suite [17], (2) GNU `grep` 2.2 [14], a 15K-line open-source regular expression matching tool, and (3) Vim 5.7 [26], a 150K-line open-source text editor.

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. These strategies cover branches more rapidly, and obtain greater overall coverage, than random testing or depth-first search. On the largest benchmark, our control-flow directed search and our second random search achieve more than twice the coverage of the other methods.

This paper makes the following contributions:

1. We describe two random search strategies that sample the path space of a program rather than the input space. As a result, these strategies achieve superior coverage in our experiments than pure random testing.

2. We present a novel search strategy that utilizes the static structure of a program (i.e. the control-flow graph) to drive dynamic test generation. This search strategy, which greedily picks paths to improve branch coverage, outperforms with respect to branch coverage both concolic testing with a traditional depth-first search and the other search strategies.

3. We have implemented our search techniques in CREST, an extensible, open-source, and publicly-available test generation tool for C programs. We successfully apply the tool to a 150K-line C application, on which the control-flow directed search strategy achieves greater coverage than any other strategy.

2 Background

We now give a brief recapitulation of concolic testing preceded by a description of the programming model.

2.1 Programming Model

We describe our concolic search strategies on a simple imperative programming language. A program P in this language consists of a set of functions $\{f_0, \ldots, f_{n-1}\}$, each consisting of a sequence of statements $f_i = s_{i,0}, \ldots, s_{i,m_i-1}$, with labels $l_{i,0}, \ldots, l_{i,m_i-1}$. One of the functions is distinguished as `main`, the function at which the execution of the program begins.

Each function f_i begins and ends with special statements $\texttt{Entry}_{f_i}$ and $\texttt{Exit}_{f_i}$. All other statements are one of: (1) an input statement $m := \texttt{INPUT}()$, (2) a call $\texttt{Call}(f)$ to some function f, (3) an assignment $m := e$ to memory location m of the value of e, an expression free of side effects, (4) a conditional `if` p `then goto` l, where l is the label of another statement in the same function and p is a predicate free of side effects, (5) an error statement `ERROR`.

For any conditional l : `if` p `goto` l', we call the statements with labels $l{+}1$ and l' the true and false *branches* of l, and call these labels *branch labels*. We require branches and branch labels to be unique. That is, no statement $l : s$ can be the target of more than one conditional `goto`, and a statement immediately following a conditional cannot be the target of any conditional `goto`. This requirement simplifies our technical description by uniquely pairing the two branches of a conditional statement. Thus, we can call the two branches $l + 1 : s$ and $l' : s'$ of conditional l : `if` p `goto` l' *paired branches*, denoted by $\overline{l + 1 : s} = l' : s'$ and $\overline{l' : s'} = l + 1 : s$.

The execution of a program P on inputs I proceeds through a sequence of labeled program statements $p_0, \ldots, p_{k-1}$, with $p_0 = l_{main,0} : \texttt{Entry}_{\texttt{main}}$, the first statement of the `main` function. We call this a *concrete path* or *execution*, denoted $\texttt{ConcretePath}(P, I)$.

2.2 Concolic Execution

Concolic testing performs symbolic execution of the program together with its concrete execution. It maintains a *symbolic memory map* $\mathcal{S}$ and a *symbolic path constraint* Φ in addition to the concrete memory. These are updated during the course of concolic execution. The symbolic memory map is a mapping from concrete memory addresses to symbolic expressions, and the symbolic constraint is a list of first order formula over symbolic input values. The details of the construction of the symbolic memory and constraints is standard [27, 12, 24]: at every statement $l : m := \texttt{INPUT}()$, the symbolic memory map $\mathcal{S}$ introduces a mapping $m \mapsto x_m$ from the address m to a fresh symbolic value x_m, and at every assignment $l : m := e$, the symbolic memory map updates the mapping of m to $\mathcal{S}(e)$, the symbolic expression obtained by evaluating e in the current symbolic memory. The concrete values of the variables (available from the concrete memory map) are used to simplify $\mathcal{S}(e)$ by substituting concrete values for symbolic ones whenever the symbolic expressions go beyond the theory that can be handled by the symbolic decision procedures.

The symbolic constraint Φ is initially an empty set. At every conditional statement l : `if` p `then goto` l', if the execution takes the then branch, the symbolic constraint $(\mathcal{S}(p) \neq 0)$ is appended to Φ and if the execution takes the else branch, the symbolic constraint $(\mathcal{S}(p) = 0)$ is appended to Φ. Thus, a conjunction of the constraints in Φ denotes a logical formula over the symbolic input values that the concrete inputs are required to satisfy to execute the path executed so far.

Given a concolic program execution along the path $p_0, p_1, \ldots, p_{k-1}$, concolic testing generates a new test input in the following way. It selects a conditional p_j along the path that was executed such that $0 \leq j < k$. (The selection of j depends on the search strategy.) Let Φ_l be the symbolic path constraint just before executing this instruction and ϕ_e be the constraint generated by the execution of this instruction. Using a decision procedure, concolic testing finds a satisfying assignment I for the constraint $(\bigwedge_{\phi \in \Phi_l} \phi) \wedge \neg\phi_e$. The property of a satisfying assignment is that if these inputs are provided at each input statement, then the new execution will follow the old execution up to the location l, but then take the conditional branch opposite to the one taken by the old execution, i.e., the new execution will be of the form $p_0, \ldots, \overline{p_j}, p'_{j+1}, \ldots, p'_m$. The satisfying assignment I is used as the new input for the next run of the program.

2.3 A Generic Concolic Search Strategy

The previous section described how we can use concolic execution of a program on a given input to generate a new input that would force the program along a different execution path. If we repeatedly perform concolic execution on the newly generated inputs, then we end up generating a set of test inputs. However, in the description of concolic execution, we did not specify how we pick a particular branch where we negate a constraint. An algorithm for selecting a particular branch in each iteration gives a search strategy for exploring the path space of the program under test. We next describe a generic algorithm for exploring the path space. The algorithm can be instantiated with a search strategy and each such instantiation will explore the path space in a different order. Our goal is to find a search strategy that enables us to quickly achieve high branch coverage.

The generic algorithm is given in Algorithm 1. The algorithm maintains a current execution path p, and is parametrized by three components:

1. A criterion for when to terminate the search.

2. A selection process for picking which branch from the current execution path p the search should force next.

3. A procedure for continuing the search given an execution obtained by forcing a branch selected in 2. (Typically this would either be to set the current execution p to be the new concrete path, or to make some sort of recursive call on the new path.)

Algorithm 1 GenericSearchStrategy(program P, path p)

```
while termination conditions are not met do
  $i \leftarrow$ pick a branch from $p$
  if $\exists I$ that forces $P$ through $p_0, \ldots, p_{i-1}, \overline{p_i}$ then
    $q \leftarrow \texttt{ConcretePath}(P, I)$
    process the new execution $q$
  end if
end while
```

In practice, we typically run such a search strategy on an initial execution of either random inputs or of all zeros. In addition to whatever stopping criteria the strategy uses, the search is terminated once a fixed budget of iterations (executions of the program under test) is exhausted. Further, if the search strategy finishes without using the entire iteration budget, it may be restarted on new inputs.

2.4 Depth-First Search

Previous concolic testing approaches [12, 24] have used depth-first search strategies to explore the path space of the program under test. We describe here BoundedDFS, a bounded-depth, depth-first search strategy which we use as a point of comparison in our experimental evaluations. Specifically, BoundedDFS(p, i, depth), explores all executions through the true and false branches of the first depth-many branches at and below statement p_i in p that can be successfully flipped by concolic testing.

Strategy BoundedDFS$(p, i, depth)$ is an instance of the generic search strategy from the previous section. A count, $forced$, is maintained of the number of branches forced along the current execution p, and the search terminates once $forced = depth$ or when there are no further branches along p to select. The branch selection process simply picks the first branch p_j with $j \geq i$ that has not yet been picked. If this branch is successfully forced, yielding new execution q, then BoundedDFS$(q, j+1, depth-1)$ is called recursively and $forced$ is increased by one. The search then continues on the initial path p.

Note that branches which cannot be forced above do not count towards the depth, and thus, barring any paths with fewer than d branches that can be forced, BoundedDFS will successfully force exactly $2^d - 1$ branches.

3 Random Search Strategies

A widely used form of automated testing is random testing. In random testing, the input space is sampled randomly to generate random inputs. Although such testing is sometimes quite effective in practice, it suffers from two key problems. First, many sets of values may lead to the same execution path and are thus *redundant*, and second, the probability of selecting particular inputs that cause some buggy behavior or explore a corner case branch may be astronomically small [22].

We next describe two search strategies for concolic testing that can randomly sample the *path space* of a program rather than its *input space*. By doing so, we avoid the problem of re-execution of redundant execution paths while keeping the inputs random. Moreover, 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.

3.1 Uniform Random Search

A natural notion of random execution path through a program P is an execution for which the true branch and false branch of each symbolic conditional is taken with equal probability. We describe a concolic search algorithm UniformRandomSearch, an instance of the generic search strategy in Section 2.3, which can generate such an execution.

UniformRandomSearch maintains a position i, initially zero, in its current execution path $p = p_0, \ldots, p_{n-1}$. When selecting a branch, it randomly chooses between terminating the search or picking one of the branches in $p_i, p_{i+1}, \ldots, p_{n-1}$. Specifically, the j^{th} branch is picked with probability 2^{-j}, leaving a probability of 2^{-m} that the search is terminated, where m is the number of branches.

If selected branch p_j can be forced, yielding path q, then the current execution p is replaced by q. In either case, the position i is set to $j + 1$ and the search continued.

We show below that, assuming that our decision procedure is complete for all symbolic branches in the program, UniformRandomSearch generates a path $q = q_0, \ldots, q_{m-1}$ with probability 2^{-k}, where k is the number of symbolic branches along q. Thus, UniformRandomSearch generates paths uniformly at random.

Prop 1. *Suppose while running UniformRandomSearch we have a target execution, $q = q_0, \ldots, q_{m-1}$, a current execution $p = p_0, \ldots, p_{n-1}$, and a current position i such that $p_0, \ldots, p_{i-1} = q_0, \ldots, q_{i-1}$. We prove by induction on k, the number of symbolic branches in $q_i, \ldots, q_{m-1}$, that the algorithm generates path q with probability 2^{-k}. Note that the desired result is the special case when $i = 0$.*

Proof.

- In the base case ($k = 0$), p and q agree on all symbolic branches in q. The remaining execution passes only through conditionals with predicates to which symbolic inputs do not flow. Thus, the execution is completely determined by $q_0, \ldots, q_{i-1}$, so UniformRandomSearch will produce q with probability $1 = 2^0 = 2^{-k}$.

- Suppose that the result holds for 0 to $k - 1$ symbolic branches, and that $q_i, \ldots, q_{m-1}$ has k symbolic branches. Let j be the least $j \geq i$ such that $p_i \neq q_j$. If no such j exists, then $p = q$ and UniformRandomSearch generates q iff it immediately terminates, which occurs with probability 2^{-k}. Otherwise, note that p_j and q_j must be symbolic branches with $\overline{p_j} = q_j$.

  Then, the algorithm generates q iff it next picks branch p_j to force, and then the recursive call generates $q_{j+1}, \ldots, q_{m-1}$. If q_j is the h^{th} symbolic branch in $q_i, \ldots, q_{m-1}$, then by the induction hypothesis, the probability that this occurs is $2^{-h} \cdot 2^{k-h} = 2^{-k}$. □

3.2 Random Branch Search

Algorithm UniformRandomSearch enables us to sample uniformly from the path space of a program under test, but it requires $L/2$ expected runs of the test program to generate a random execution of length L. Although UniformRandomSearch gives a nice theoretical guarantee of a uniformly random search of the path space, we found, after trial-and-error, a simpler random search strategy that is more effective in practice. In the simpler method we simply force a random branch along the current path in each iteration.

This method, RandomBranchSearch, is again an instance of our generic search strategy in Section 2.3. The branch selection procedure simply picks a random branch along the current execution p, and, if the branch can be forced, replaces the current execution with the resulting one. Optionally, the search can restart on new inputs if it fails to uncover any new branches after some number of iterations.

4 Control-Flow Directed Search

The goal of any search strategy in concolic testing is to generate inputs that collectively cover as many branches as possible in the program under test. In this section we describe a concolic search strategy, CfgDirectedSearch, which uses the static structure of the test program to direct the search along short paths to currently uncovered branches. Our experiments demonstrate that this approach can improve branch coverage faster than a traditional depth-first search, which must systematically explore all paths.

At a high level, CfgDirectedSearch constructs a combined *control flow and static call graph* for the program under test. Given a current execution, the algorithm finds short paths through this static graph from branches along the execution to branches that have not yet been covered. It then attempts to force the execution down these short paths.

In the next sections, we formally describe the control flow and static call graph for a program. We then describe a sub-strategy, SearchAlongPath, for forcing an execution along a static path through this graph, and define our notion of the length of such a path. Finally, we describe the full CfgDirectedSearch algorithm, which is built from these components.

4.1 Control Flow and Static Call Graph

We define the *control flow and static call graph* (CFCG) for a program P. This graph captures the possible paths the test program can take to reach any given branch. We use the CFCG during concolic search to determine which branches to force in order to draw closer to uncovered branches.

For a program P consisting of functions $f_1, \ldots, f_n$, the combined *control flow and static call graph* $CFCG_P$ is a directed graph whose vertices are the statements (equivalently, labels) of P, and with edges from each statement $l_{i,j} : s_{i,j}$ to its immediate successors:

$$\begin{cases} \text{none} & \text{if } s_{i,j} = \texttt{Exit}_{f_i} \\ l' \text{ and } l_{i,j+1} & \text{if } s_{i,j} = \texttt{if } p \texttt{ goto } l' \\ \texttt{Entry}_f \text{ and } l_{i,j+1} & \text{if } s_{i,j} = \texttt{Call}(f) \\ l_{i,j+1} & \text{otherwise} \end{cases}$$

Note that $CFCG_P$ is exactly the union of the traditional, per-function control flow graphs for $f_0, \ldots, f_{n-1}$, with an added *call edge* from each call call site to the entry point of the called function.

We will call a path in $CFCG_P$ a *static path*. Note that because $CFCG_P$ contains no edges from functions back to their call sites, a static path through some statement $l : \texttt{Call}(f)$ must either skip entirely over the body of f or enter f and never leave. Thus, a static path captures only a fragment of a possible execution of P. Specifically, a static path from $\texttt{Entry}_{\texttt{main}}$ to some statement s of P gives a sequence of statements and function calls an execution could take to reach s, but omits the concrete path through any function call that does not enclose s.

We next describe a relation *matches* between dynamic executions and static paths. We will use this relation to describe our CFG-directed search algorithm. Formally, we say that a subsequence $p_i, \ldots, p_j$ of an execution $p = p_0, \ldots, p_{n-1}$ *matches* some static path $S = s_0, \ldots, s_{m-1}$ iff $p_i = s_0$ and either:

- $p_{i+1}, \ldots, p_j$ matches $s_1, \ldots, s_{m-1}$
- $p_i, \ldots, p_j = \texttt{Call}(f), \texttt{Entry}_f, \ldots, \texttt{Exit}_f, p_{i'}, \ldots, p_j$ and $p_{i'}, \ldots, p_j$ matches $s_1, \ldots, s_{m-1}$

Further, we define an empty static path to be matched by any subsequence of an execution.

4.2 Dynamic Search Along a Static Path

We describe a concolic search algorithm which, given an execution of program P and a static path S, attempts to force the execution of P to follow S. Our overall control-flow-directed search strategy will consist essentially of selecting short static paths to uncovered branches and then using this procedure to solve for an execution that reaches the uncovered branch.

Formally, SolveAlongPath(p, i, S) is a procedure which takes as input an execution $p = p_0, \ldots, p_{n-1}$ through program P, a branch p_i on p, and a static path $S = s_0, \ldots, s_{m-1}$ in $CGCG_P$. If successful, it returns a path q such that $p_0, \ldots, p_{i-1} = q_0, \ldots, q_{i-1}$ and the remainder $q_i, q_{i+1}, \ldots$ of q matches S.

Procedure SearchAlongPath is an instance of the generic concolic search strategy given in Section 2.3. In a call to SearchAlongPath(p, i, S), we find the longest matching subsequences $p_i, \ldots, p_{i+k-1}$ and $s_0, \ldots, s_{j-1}$ of execution p and static path S, respectively. If S is completely matched, the search terminates successfully. Otherwise, if p_{i+k} and s_j are branches, and $\overline{p_{i+k}} = s_j$, then we select branch p_{i+k} to be forced. If forcing is successful, returning a new path p', we set $p \leftarrow p'$ and allow the search to continue. If either one of p_{i+k} or s_j is not a branch or the forcing does not succeed, the search terminates in failure.

Each successful forcing above increases the number of branches in S that are matched. Thus, SearchAlongPath will terminate after having selected and forced a branch no more times than the number of branches in S. The number of branches along S is therefore a measure of the difficulty of forcing a concrete execution to match S.

Motivated by the above observation, we define a distance metric on $CFCG_P$ to capture this difficulty of forcing execution along the static graph. Each edge (s, t) is given weight 1 iff t is a branch and weight 0 otherwise. Then, the weight of static path $p_{i-1}, s_0, \ldots, s_{m-1}$ equals the number of branches along S, and thus bounds the cost of

SearchAlongPath(p, i, S). The distance $d(s, t)$ from statement s to t is then the minimum weight over all paths from s to t in $CFCG_P$.

Note that procedure SearchAlongPath is not complete because it does not explicitly search over all executions through the functions that the static path skips over. A search over all such executions, however, is not necessarily even finite.

4.3 CFG-Directed Search Algorithm

Using the procedure SearchAlongPath from the previous section, we describe a search algorithm CfgDirectedSearch which attempts to systematically increase the branch coverage of a program under test by driving execution down short static paths to currently uncovered branches.

Algorithm CfgDirectedSearch is an instance of the generic search strategy given is section 2.3. As with the other strategies described, the search terminates in failure if it runs out of branches to select, it exhausts its budget of test iterations, or if it uncovers no new branches after some set number of iterations. We describe the branch selection procedure and processing of new execution paths below.

Selection of Branches. The algorithm selects branches b to force which have the shortest static paths from their paired branches $\bar{b}$ to a currently uncovered branch. Specifically, using the distance metric on $CFCG_P$ from the previous section, we define:

$$\mathit{UncoveredDistance}(\bar{b}) \;=\; \min_{b' \text{uncovered}} d(\bar{b}, b')$$

Further, for each branch $\bar{b}$ we track *tries*$(\bar{b})$, the number of times that we have previously flipped the execution from b to $\bar{b}$ during the current search. When selecting a branch from $p = p_0, \ldots, p_{n-1}$, we randomly pick a p_i with minimal *UncoveredDistance*$(\overline{p_i})$ + *tries*$(\overline{p_i})$.

Note that we can compute *UncoveredDistance*(b) for all branches b with a single run of Dijkstra's Algorithm on $CFCG_P$ with its edges reversed. We initialize *UncoveredDistance*$(b) = 0$ for all uncovered branches b and then Dijkstra's Algorithm will find, for each branch b', the minimum distance in $CFCG_P$ from b' to any uncovered branch. (Recall that each edge (s, t) in $CFCG_P$ has weight 1 iff t is a branch and weight zero otherwise.) We recompute these quantities whenever a new branch is covered.

The addition of the *tries*$(\overline{p_i})$ term is a heuristic used to address two problems encountered in practice. We justify this branch selection heuristic below, after first describing how the CfgDirectedSearch algorithm processes new paths found when forcing its selected branches.

Processing of New Paths. If we are successful in forcing some branch p_i selected above, yielding an execution q, we then use SearchAlongPath$(q, i + 1, S)$ to try to drive the execution down *all* static paths S of weight *UncoveredDistance*$(\overline{p_i})$ + *tries*$(\overline{p_i})$ or less from $\overline{p_i}$ to currently uncovered branches. If any call to SearchAlongPath succeeds, we find some execution path which hits a previously uncovered branch.

In this case, we update the current concrete path to equal the new path, reset *tries*(b), and recompute *UncoveredDistance*(b), before continuing CfgDirectedSearch. Thus, our control-flow directed search is a *local* search, as it essentially restarts on every newly-discovered branch and never explicitly revisits older paths.

If instead SearchAlongPath$(q, i + 1, S)$ does not succeed for any static path S of length up to *UncoveredDistance*$(\overline{p_i})$ + *tries*$(\overline{p_i})$, then we increase *tries*$(\overline{p_i})$ by one and return to branch selection.

Note that, for any fixed k, the number of weight-k static paths (i.e. with k or fewer branches) in $CFCG_P$ from $\overline{p_i}$ is finite because every cycle in $CFCG_P$ has length at least one. This is because every cycle must contain either a conditional `goto` or a recursive call, and any non-degenerate recursive call must be guarded by a conditional statement.

In practice, this search over all weight-k paths is efficient both because the number of such paths tends to be small for small k and because we can force along multiple static paths simultaneously. For example, if we wish to attempt to force the execution from some statement along static paths $S = s_0, s_1, s_2, s_3$ and $S' = s_0, s_1, s_2, \overline{s_3}$, we need only force once along the common prefix s_0, s_1, s_2.

Branch Selection Heuristic. The simplest way to prioritize the branches $b \in p$ above would be to simply pick them in increasing order of *UncoveredDistance*$(\bar{b})$. But there are two practical problems with this approach. First, the search can get stuck repeatedly forcing a small number of branch statements which appear along the current execution many times and which have short but infeasible paths to uncovered branches. Second, for many branches p_i along p, the shortest *feasible* path through $CFCG_P$ from $\overline{p_i}$ to an uncovered branch may be slightly longer than *UncoveredDistance*$(\overline{p_i})$. With this simplest approach, static paths longer than *UncoveredDistance*$(\overline{p_i})$ will never explicitly be explored.

The addition of the *tries*$(\bar{b})$ term helps to address both of these issues. Because we increase *tries*$(\bar{b})$ each time we force some $p_i = b$ but fail to find any uncovered branches, we will not get stuck forcing all other occurrences $p_j = b$ of the branch before trying other branches with similar *UncoveredDistance*'s. Further, the next time some $p_j = b$ is forced, longer static paths will be explored. Thus, we can view the increase in *tries*$(\bar{b})$ as revising upwards our estimate of the least *feasible* distance from $\bar{b}$ to some uncovered branch.

This heuristic could likely be improved by maintaining more detailed, global information about when CfgDirectedSearch has failed to find uncovered branches below each source branch, rather than tracking only simple counts and discarding them whenever a new branch is found. Further, although increasing *tries*$(\bar{b})$ after forcing some $p_i = b$ leads to the algorithm searching along longer static paths the next time it forces some other branch $p_j = b$, it never returns to p_i to explore longer paths. However, the current heuristic is sufficient to achieve significant branch coverage in practice.

5 Evaluation

We have implemented our search strategies in CREST, a prototype test generation tool for C programs. We experimentally evaluate the effectiveness of these strategies on `replace`, the largest program in the Siemens Benchmark Suite, and on two popular open-source applications, `grep` 2.2 [14] and Vim 5.7 [26]. All experiments were run on 2GHz Core2 Duo servers with 2GB of RAM and running Debian GNU/Linux.

With these experiments, we aim to validate the hypotheses that: (1) on a fixed testing budget, our random and control-flow directed search strategies can yield higher overall branch coverage than traditional random testing or concolic testing with a depth-first search strategy, and (2) that relative performance of these methods improves for programs with larger path spaces.

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 symbolic execution to dominate processing done by the strategies themselves. All unconstrained inputs were initially set to zero.

For both `grep` and Vim, the way we instrument and run the tested programs restricts the set of possible program behaviors. Thus, in addition to reporting *absolute* branch coverage, we report *relative* coverage – the 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.

5.1 Implementation

CREST, our open-source test-generation tool, consists of three main pieces: an OCaml instrumentation tool, a C++ concolic execution library, and a C++ search strategy framework.

CIL [21], an OCaml application for parsing, transforming, and analyzing C code, is used to instrument the source of the program under test for concolic testing and to extract the control flow and static call graph. The library performs concolic execution, as described in Section 2.1, simultaneously with the concrete execution of the program.

The search strategy framework provides primitives for writing a concolic search strategy, such as methods for flipping and solving path constraints using the Yices [7] SMT solver. We implemented the search strategies described in this paper in this framework in 1000 lines of C++ code.

Note that CREST does not currently perform any static reasoning about calls through function pointers. Thus, we may encounter execution paths that do not correspond to any static paths in the control flow and static call graph, potentially leading to dynamic distances between branches that are smaller than the static distances in the $CFCG$.

5.2 Siemens Benchmark Suite

The Siemens Benchmark Suite [17] contains seven programs for benchmarking bug finding and program analysis tools. `replace`, the largest of these benchmarks, is a 600-line text processing program, with 200 branches after instrumentation.

`replace` reads a source and destination pattern and a line of text, and replaces the source pattern with the destination pattern in the line. In our experiments, we restricted the two patterns to 10 symbolic characters, and the line of text to 20 symbolic characters for all strategies but the depth-first search. To have a fair comparison, we used 5-character patterns with the DFS strategy, so that with a depth of 14 the search could reach the pattern matching code. Further, a minor optimization was needed for the CFG-directed search on this benchmark to handle the small program size, the details of which are omitted for space reasons.

Figure 1 is a plot of the coverage achieved by the various strategies over 3000 iterations, averaged over three trials. In a single minute of testing, all of the concolic search strategies were able to cover 80% of the branches in `replace`. In fact, in an additional couple of minutes the best concolic runs achieved 85% or even 90% branch coverage. This is close to the best possible, as a brief inspection shows that many of the remaining branches are clearly infeasible.

These experiments provide some evidence that our search strategies can obtain branch coverage at a greater rate than random testing or depth-first concolic search. `replace` is small enough, however, that a depth-first search is able to exhaustively explore a large portion of the path space. We need to test on larger benchmarks to evaluate whether our search strategies can provide significant branch coverage where an exhaustive, depth-first search cannot.

5.3 GNU Grep

GNU `grep` is a widely-used open-source tool for text search with regular expressions [14]. `grep` 2.2 contains roughly 15K lines of C code and, after instrumentation, 2142 conditional statements with 4184 branches.

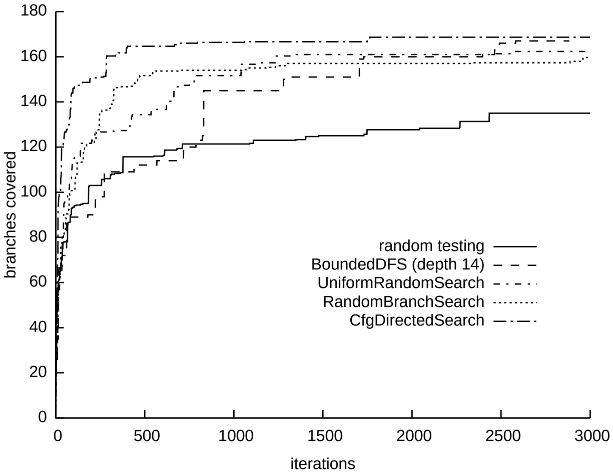

Figure 1. Branch coverage achieved by different search strategies on `replace`, averaged over three runs. `replace` contains 200 branches, all of which are reachable.

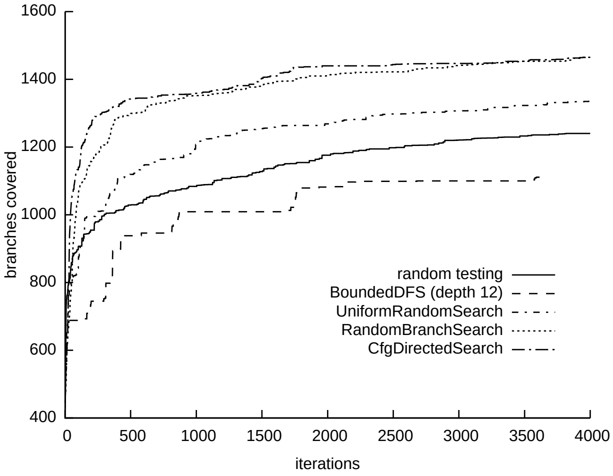

Figure 2. Branch coverage achieved by different search strategies on `grep` 2.2, averaged over three runs. `grep` 2.2 contains 4184 branches, an estimated 2854 of which are reachable given our instrumentation.

In our experiments, we modified `grep` 2.2 to match a length-m symbolic pattern against n symbolic characters. We ran the instrumented `grep` with no arguments except for a pattern string and a single file to search, and we thus tested neither `grep`'s "fixed" or "extended" modes, nor most of its processing of command-line arguments or multiple input files. This limits the achievable branch coverage, so we report relative as well as absolute coverage.

For all search strategies but the depth-first search (DFS), we used $m = 20$ and $n = 40$ for the symbolic input sizes. These limits keep concolic execution of the instrumented `grep` reasonably efficient – roughly 40 runs per second – while providing sufficient freedom to exercise most possible program behaviors. In order to have a fair comparison against the DFS strategy, we used $m = 5$ and $n = 40$ for the depth-first search, so that a depth-12 search was able to reach past the `grep` code for parsing and pre-processing the pattern and into the matching code.

Figure 2 is a plot of the coverage achieved by the various strategies over 4000 iterations, averaged over three trials. In a couple of minutes, the most effective search strategies were able to cover more than a third of the 4184 branches — nearly 60% of the estimated 2854 reachable branches.

Notice that the control-flow directed search and both random searches outperformed traditional random testing and depth-first search. In particular, the CFG-directed and random-branch strategies increased coverage very rapidly in the first 200 to 300 iterations. In fact, even if we increased the input size to $n = 200$ and $m = 400$, slowing execution down to only 4 iterations per second, random testing came close to, but still could not match, the performance of the CFG-directed and random-branch searches with only $n = 20$ and $m = 40$. Similarly, allowing the depth-first search a depth of 14 and nearly 16K iterations closed only half the gap to these two strategies.

Note also that the depth-12 DFS terminated in fewer than 2^{12} iterations because some executions contained fewer than 12 feasible symbolic branches.

5.4 The VIM Editor

Vim is a popular, open-source text editor [26]. Vim 5.7 contains roughly 150K lines of C code and, after instrumentation, 39166 branches.

We replaced the `safe_vgetc` (and `vgetc`) input functions with ones returning symbolic inputs. 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, and therefore report relative in addition to absolute branch coverage. In our experiments we restricted Vim 5.7 to receive 20 symbolic characters as input. Due to the size of Vim and the cost of symbolic execution, we could run only one test execution every 2-3 seconds.

Figure 3 is a plot of the coverage achieved by the various search strategies over 4000 iterations, averaged over three trials. In 2-3 hours of testing, the most effective search strategy covered 19% of the total branches, or nearly a third of the estimated 23400 reachable branches. This performance is close to that reported in [19] on Vim 5.7, but here this coverage is achieved with inputs of only 20 characters rather than inputs thousands or millions of characters long.

Note that, on this larger benchmark, the control-flow directed and random-branch searches achieved more than

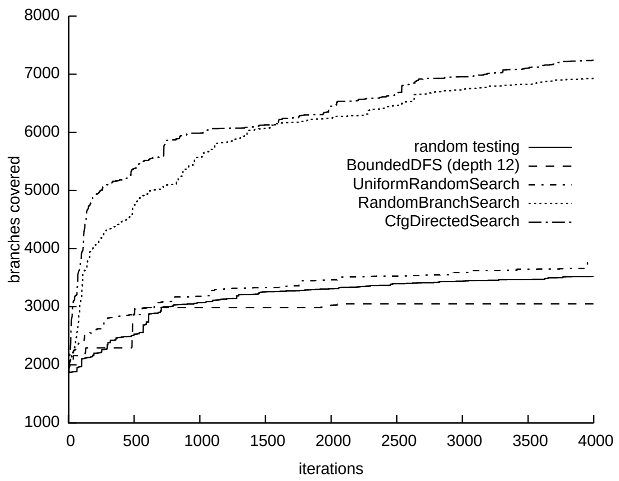

Figure 3. Branch coverage achieved by different search strategies on Vim 5.7, averaged over three runs. Vim 5.7 contains 39166 branches, an estimated 23400 of which are reachable given our instrumentation.

twice the coverage of random testing or concolic testing with a depth-first search. Further, these two strategies obtained coverage very rapidly, achieving at iterations 100 and 150, respectively, greater coverage than the other strategies did in 4000 iterations.

Further, even allowing 200 characters of input, random testing covered only roughly 4000 branches over 4000 iterations, still far short of the performance of the control-flow directed and random-branch searches.

6 Related Work

Groce and Visser [15] present several strategies for exploring the state space in the context of model checking Java programs. Specifically, they propose best-first, A^*, and beam search, combined with code coverage heuristics, and compare these search strategies with traditional DFS and BFS. Further, dynamic test generation tools EXE [3] and SAGE [13] use similar strategies and code coverage heuristics. One key difference between these approaches and our CFG-directed strategy is that these heuristics use dynamic properties (i.e. measured along the already-explored paths), whereas our CFG-directed heuristic exploits the static structure of the program. Therefore, our search strategy bases its decision on both explored and unexplored parts of the program.

Another key difference is that the above approaches use *global* searches, which maintain a pool of many possible executions from which the search through the path space can be continued. The search strategies in this work are essentially *local*, considering only a single execution at a time from which new executions are generated. We believe such global search strategies could be gainfully combined with our static control-flow based heuristic search.

Grammar-based techniques have recently been proposed [20, 11] for generating complex inputs for software systems. While very effective, these techniques require a grammar to be given for the test program's input, which may not always be feasible.

Hybrid Concolic Testing (HCT) [19] interleaves random testing with bounded, depth-first concolic search. The key difference between HCT and our search strategies is that HCT is only suitable for reactive systems that have infinite behavior, whereas CREST is applicable to general programs. Moreover, HCT typically produces extremely long test inputs, which can hinder debugging.

Several randomized algorithms for model checking have also been proposed. For example Monte Carlo Model Checking [16] uses random walks on the state space to give probabilistic guarantees on the validity of properties expressed in linear temporal logic. Statistical model checking techniques [30, 25] verify probabilistic models against probabilistic properties approximately within probabilistic error bounds. Randomized depth-first search and its parallel extension [8] have been developed to dramatically improve the cost-effectiveness of state-space search techniques using parallelism. Evolutionary algorithms have been used in the Verisoft model-checker [10] to find bugs quickly. However, most of these techniques are applicable to concurrent programs that have data inputs from a small domain.

7 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. We have presented several strategies for dynamically searching the path space of a test program to generate test inputs, including one strategy that uses the static control flow graph of a program to drive dynamic test generation. Our experiments show that two of these approaches, one which randomly searches the path space and one whose search is guided by the static structure of the program under test, can obtain greater coverage on real-world software systems than either random testing or concolic testing with a depth-first search which attempts to exhaustively search the path space. In particular, these strategies achieve more than twice the coverage on our largest benchmark.

8 Acknowledgments

Thanks to Caltech's UGCS, of the Student Computing Consortium, for providing the computing resources used in this work. This work is supported in part by the NSF Grant CNS-0720906 and a gift from Toyota.

References

[1] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. Generating Test from Counterexamples. In *Proc. of the 26th ICSE*, pages 326–335, 2004.

[2] D. Bird and C. Munoz. Automatic Generation of Random Self-Checking Test Cases. *IBM Systems Journal*, 22(3):229–245, 1983.

[3] 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.

[4] K. Claessen and J. Hughes. Quickcheck: A lightweight tool for random testing of Haskell programs. In *Proc. of 5th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 268–279, 2000.

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

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

[7] B. Dutertre and L. M. de Moura. A fast linear-arithmetic solver for DPLL(T). In *Computer Aided Verification*, volume 4144 of *LNCS*, pages 81–94, 2006.

[8] M. B. Dwyer, S. Elbaum, S. Person, and R. Purandare. Parallel randomized state-space search. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 3–12. IEEE, 2007.

[9] 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.

[10] P. Godefroid and S. Khurshid. Exploring very large state spaces using genetic algorithms. In *8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 02)*, pages 266–280, 2002.

[11] P. Godefroid, A. Kiezun, and M. Levin. Grammar-based Whitebox Fuzzing. *PLDI*, 2008. (to appear).

[12] 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.

[13] P. Godefroid, M. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. Technical report, Technical Report MSR-TR-2007-58, Microsoft, May 2007.

[14] GNU grep. `http://www.gnu.org/software/grep/grep.html`.

[15] A. Groce and W. Visser. Heuristics for model checking java programs. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(4):260–276, 2004.

[16] R. Grosu and S. A. Smolka. Monte carlo model checking. In *11th International Conference Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, volume 3440 of *LNCS*, pages 271–286, 2005.

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

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

[19] R. Majumdar and K. Sen. Hybrid concolic testing. In *29th International Conference on Software Engineering (ICSE'07)*, pages 416–426. IEEE, 2007.

[20] R. Majumdar and R. Xu. Directed test generation using symbolic grammars. *Foundations of Software Engineering*, pages 553–556, 2007.

[21] 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.

[22] J. Offut and J. Hayes. A Semantic Model of Program Faults. In *Proc. of ISSTA'96*, pages 195–200, 1996.

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

[24] 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.

[25] K. Sen, M. Viswanathan, and G. Agha. Statistical model checking of black-box probabilistic systems. In *16th International Conference on Computer Aided Verification (CAV'04)*, volume 3114 of *LNCS*, pages 202–215. Springer, 2004.

[26] VIM. `http://www.vim.org/`.

[27] W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. In *Proc. 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 97–107, 2004.

[28] S. Visvanathan and N. Gupta. Generating test data for functions with pointer inputs. In *17th IEEE International Conference on Automated Software Engineering*, 2002.

[29] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Procs. of TACAS*, 2005.

[30] H. L. S. Younes and R. G. Simmons. Probabilistic verification of discrete event systems using acceptance sampling. In *14th International Conference on Computer Aided Verification (CAV'02)*, volume 2404 of *LNCS*, pages 223–235. Springer, 2002.