Generating Tests from Counterexamples[*]

Dirk Beyer Adam J. Chlipala
Thomas A. Henzinger Ranjit Jhala
Electrical Engineering and Computer Sciences
University of California, Berkeley, USA

Rupak Majumdar
Computer Science Department
University of California, Los Angeles, USA

Abstract

We have extended the software model checker BLAST *to automatically generate test suites that guarantee full coverage with respect to a given predicate. More precisely, given a C program and a target predicate p,* BLAST *determines the set L of program locations which program execution can reach with p true, and automatically generates a set of test vectors that exhibit the truth of p at all locations in L. We have used* BLAST *to generate test suites and to detect dead code in C programs with up to 30 K lines of code. The analysis and test-vector generation is fully automatic (no user intervention) and exact (no false positives).*

1. Introduction

In recent years software model checking has made much progress towards the automatic verification of programs. A key paradigm behind some of the new tools is the principle of counterexample-guided abstraction refinement [2, 18, 5]. The input to the model checker is both the program source and a monitor automaton [31, 15], which observes if a program trace violates a temporal safety specification, such as adherence to a locking or security discipline. The checker attempts to verify a program abstraction, and if the verification fails, it produces a path that violates the specification. If this abstract path does not correspond to a concrete trace of the program —*i.e.,* the path is infeasible— then the abstraction is automatically refined in a way that removes the infeasible path. The entire process is repeated until either an error trace of the program (a so-called "counterexample") is found, or the absence of such traces is guaranteed. In this way, large Windows and Linux device drivers have been checked without user intervention, and without generating false positives [2, 18].

The information provided by traditional model checkers, however, is limited. In particular, the software engineer is often interested not in obtaining a particular program trace that violates a given temporal property, but in the set of *all* program locations where the property may be violated: given a predicate p, the programmer may wish to know the set of all program locations q that can be reached such that p is true at q. For example, when checking the security properties of a program it is useful to find the locations where the program has root privileges. We have extended the model checker BLAST[1] [18] to provide this kind of information. As a special case (take p to be the predicate that is always true), BLAST can be used to find the reachable program locations, and by complementation, it can detect dead code.

Moreover, if BLAST claims that a certain program location q is reachable such that the target predicate p is true at q, then from the program trace that exhibits p at q, the tool automatically produces a test vector that witnesses the truth of p at q. This feature enables the software engineer to pose reachability queries about the behavior of a program, and to automatically generate test vectors that satisfy the queries [27]. Technically, we symbolically execute the counterexample trace produced by the model checker, and extract a satisfying assignment of the symbolic constraints as a test vector. In particular, for a predicate p and its negation, the tool automatically generates for each program location q, if p is always true at q, a test vector that exhibits p at q; if p is always false at q, a test vector that exhibits $\neg p$ at q; and if p may be true or false at q, then two test vectors, one that exhibits the truth of p at q, and another one that exhibits the falsehood of p at q. In this way, BLAST generates more informative test suites than any tool that is purely based on coverage, because the program locations of the third kind are each covered by two test vectors with different outcomes.

Often a single test vector covers the truth of p at many locations, and the falsehood of p at others, and BLAST

∗ This research was supported in part by the NSF grants CCR-0085949, CCR-0234690, and ITR-0326577.

1 Available at `http://www.eecs.berkeley.edu/~blast`.

produces a small set of test vectors that provides the desired information. It is essential that BLAST uses incremental model checking technology [17], which reuses partial proofs and counterexamples as much as possible. We have used our extension of BLAST to query C programs with 30 K lines of code about locking disciplines, security disciplines, and dead code, and to automatically generate corresponding test suites.

There is a rich literature on test-vector generation using symbolic execution [8, 23, 30, 21, 14, 12, 22]. Our main insight is that given a particular target, one can guide the search to the target efficiently by searching only an *abstract* state space, and refining the abstraction to prune away infeasible paths to the target found by the abstract search. This is exactly what the model checker does for us. In contrast, unguided symbolic-execution based methods would have to precisely execute many more paths, resulting in scalability problems. Therefore, most research on symbolic-execution based test generation curtails the search by bounding, *e.g.,* the number of iterations of loops, or the size of the input domain [20, 4, 22]. Unfortunately, this makes the results incomplete: if no trace to the target is found, one cannot conclude that no execution of the program reaches the target. Of course, once a suitable trace to the target is found, all previous methods to generate test vectors still apply.

This is not the first attempt to use model checking technology for automatic test-vector generation. However, the previous work in this area has followed very different directions. For example, the approach of [19] considers fixed boolean abstractions of the input program, and does not automatically refine the abstraction to the degree necessary to generate test vectors that cover all program locations for a given set of observable predicates. Peled [28] proposes three further ways of combining model checking and testing. Black-box checking and adaptive model checking assume that the actual program is not given at all or not given fully. Unit checking [13] is the closest to our approach in that it generates test vectors from traces, however, these traces are not found by automatic abstraction refinement.

2. Overview

We first give an overview of the method using a few small examples. Consider the program of Figure 1(a), which computes the middle value of three integers. The program takes three inputs and invokes the function `middle` on them. A test vector for this program is a triple of input values, one for each of the variables `x,y,z`. The right column of Figure 1(b) shows the control-flow automaton (CFA) for `middle`. The CFA is essentially the control-flow graph of `middle` with the control locations as nodes, and edges labeled by the operations that take the program from one node to the next —either basic blocks of assignments, or predicates that correspond to branch conditions which must be true for control to flow across an edge. For brevity, we omit the CFA for the `main` function. We first consider the problem of location coverage, *i.e.,* we wish to find a set of test vectors such that for each location of the CFA, there is some test vector in the set that drives the program to that location.

Phase 1. Model checking. To find a test vector that takes the program to location `L5`, we first invoke BLAST to check the property that `L5` is reachable. BLAST proceeds by iterative abstraction refinement to check that `L5` is reachable and, if this is the case, it finds a counterexample, *i.e.,* a trace to `L5` in the CFA. This trace is given by the following sequence of operations: `m=z; assume (y<z); assume (x<y);` where the first operation corresponds to the assignment upon entry, and the second and third (`assume`) operations correspond to the first two branch conditions being taken.

Phase 2. Tests from counterexamples. In the second step, we use the counterexample trace from the model-checking phase to find a test vector, *i.e.,* an initial assignment for `x,y,z` that takes the program to location `L5`. This is done as follows. First, we build a trace formula (TF), which is a conjunction of constraints, one constraint per operation in the trace. In this case the formula is $(m = z) \wedge (y < z) \wedge (x < y)$. Second, the feasibility of the trace implies that the TF is be satisfiable, and we find a satisfying assignment to the formula, *e.g.,* "`x=0,y=1,z=2,m=2`", which after ignoring the value for `m`, gives a test vector that takes the program to `L5`.

We repeat these two phases for each location, noting that one input takes us to several locations —those along the trace— until we have a set of test vectors that covers all locations of the CFA. Along with each test vector, BLAST also produces a trace in the CFA that is exercised by the test. A set of test vectors for node coverage of `middle` is shown in Figure 2. Each row in the table gives an input vector — initial values for `x,y,z`— and the corresponding trace as a sequence of locations. For example, the vector of test values for the target location `L12` is `(1,0,1)`, and BLAST reports the trace $\langle$`L1,L2,L3,L6,L10,L12`$\rangle$, which is easy to understand with the help of the CFA in Figure 1(b). The trace is a prefix of the complete program execution for the corresponding test vector.

The alert reader will have noticed that the tests do not cover all locations; `L13` and `L15` remain uncovered, as denoted by the absence of shading in Figure 1(b). It turns out that BLAST proves that these locations are not reachable — *i.e.,* they are not visited for any initial values of `x,y,z`— and hence there exists dead code in `middle`. A close analysis of the source code reveals that a pair of braces is missing, and that the indentation is misleading for the code with-

```
#include <stdlib.h>
#include <stdio.h>

int readInt(void);

int middle(int x, int y, int z) {
L1:   int m = z;
L2:   if(y < z)
L3:     if(x < y)
L5:       m = y;
L6:     else if(x < z)
L9:       m = x;
      else
L10:   if(x > y)
L12:      m = y;
L13:   else if(x > z)
L15:      m = x;
L7:  return m;
}
int main() {
  int x, y, z;
  printf("Enter the 3 numbers: ");
  x = readInt();
  y = readInt();
  z = readInt();
  printf("Middle number: %d", middle(x,y,z));
}
```

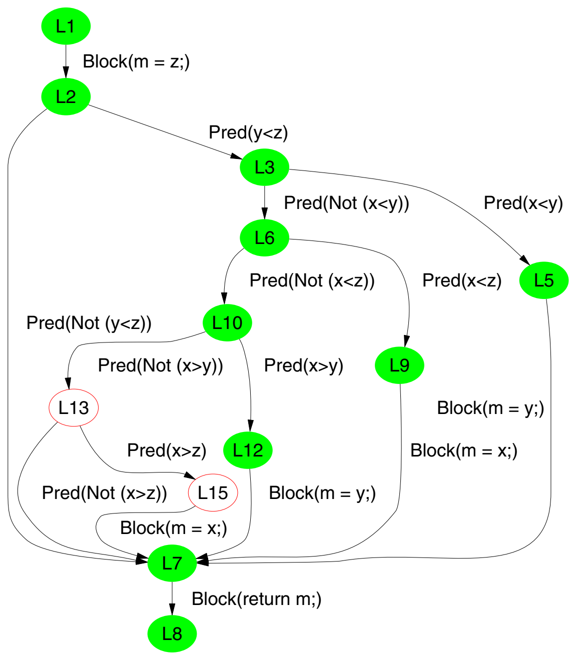

Figure 1. `middle` **(a) Program (b) CFA**

out braces: the `if` on `L6` matches the `else` after `L9`, which is meant for the `if` on `L2`.

Executing tests. To execute the generated tests, we automatically build a test driver from the given program. We feed the program and the name of the initial function from which the program's execution begins, into BLAST's test-driver generator, which results in a C program that is compiled into a test driver. The test driver reads a file containing a set of test vectors we wish to run, and executes the program being tested using the vectors as input values. The user may run the driver in a debugger to study the dynamic behavior of the program under the various test inputs.

A security example. We now show how BLAST can offer help to the programmer to check for security vulnerabilities in programs. Figure 3 shows a simple program that manipulates Unix privileges using `setuid` system calls. Unix processes can execute in several privilege levels; higher privilege levels may be required to access restricted system resources. Privilege levels are based on process user id's. Each process has a real user id, and an effective user id. The `seteuid` system call is used to set the effective id, and hence the privilege level of a process. The user id `0` (or root) allows a process full privileges to access all system resources. We assume for our program that the real user id of the process is not zero, *i.e.,* the real user does not have root privileges. This specification is a simplification of the actual behavior of `setuid` system calls in Unix [7], but is sufficient for exposition.

x	y	z	Counterexample Trace
0	0	0	⟨L1,L2,L7,L8⟩
0	1	2	⟨L1,L2,L3,L5⟩
0	0	1	⟨L1,L2,L3,L6,L9⟩
1	0	1	⟨L1,L2,L3,L6,L10,L12⟩

Figure 2. Generated test vectors for `middle`

The `main` routine first saves the real user id and the effective user id in the variables `saved_uid` and `saved_euid`, respectively, and then sets the effective user id of the program to the real user id. This last operation is performed by the function call `seteuid`. The function `get_root_privileges` changes the effective user id to the id of the root process (id `0`), and returns `0` on success. If the effective user id has been set to root, then the program does some work (in the function `work_and_drop_privileges`) and sets the effective user id back to `saved_uid` (the real user id of the process) at the end (`L9`). To track the state changes induced by the `setuid` system calls, we instrument the code for the relevant system calls as follows. The user id is explicitly kept in a new integer variable `uid`; the `getuid` function is instrumented to return a nonzero value (modeling the fact that the real user id of the process is not zero); and the `geteuid` function is instrumented to nondeterministically return either a zero or a nonzero value. Finally, we change the `seteuid(x)` system call so that the variable `uid` is updated with the argument `x` passed to `seteuid` as a parameter. The instrumented versions are omitted for brevity.

```
int saved_uid, saved_euid;

int get_root_privileges () {
L1: if (saved_euid!=0) {
L2:   return -1;
    }
L3: seteuid(saved_euid);
L4: return 0;
}

work_and_drop_priv() {
L5: FILE *fp = fopen(FILENAME,"w");
L6: if (!fp) {
L7:   return;
    }
L8:   // work
L9: seteuid(saved_uid);
}

int main(int argc, char *argv[]) {
L10:saved_uid = getuid();
L11:saved_euid = geteuid();
L12:seteuid(saved_uid);
L13:  // work under normal mode
L14:if (get_root_privileges ()==0){
L15:  work_and_drop_priv();
    }
L16:execv(argv[1], argv + 1);
}
```

Figure 3. The `setuid` example program

```
L10: saved_uid = getuid();
  /* body of getuid omitted */
L11: saved_euid = geteuid();
  /* body of geteuid omitted */
  /* geteuid returns 0 */
L12: seteuid(saved_uid);
  /* uid = saved_uid */
L14: tmp = get_root_privieleges();
  L1: if (saved_euid!=0) /* fails */
  L3: seteuid(saved_euid);
  /* uid = saved_euid */
  L4: return 0;
L14: if (tmp==0) /* succeeds */
L15: work_and_drop_priv();
  L5: fp = fopen(FILENAME, ``w'');
  L6: if (!fp) /* succeeds */
  L7: return;
L16: /* uid = 0 */
```

Figure 4. A trace generated by BLAST

Secure programming practice requires that certain system calls that run untrusted programs should not be made with root privileges [6], because the privileged process has full permission to the system. For example, calls to `exec` and `system` must never be made with root privileges. Therefore it is useful to check which parts of the code may run with root privileges.

We use the model checker BLAST in test mode to check which code lines can be executed with root privileges. More specifically, we ask the model checker to output all locations that are reachable in the program, with `uid=0` as target predicate (which indicates root privileges). For each such location, BLAST generates a test vector that causes the program to reach that location with the system being in a state where `uid=0`.

The BLAST output shows, surprisingly, that the `execv` system call can be executed with root privileges. An inspection of the symbolic program trace generated by the model checker (Figure 4) shows that there is a bug in the `work_and_drop_privileges` function: if the call to `fopen` fails, the function returns without dropping root privileges.

Organization. Section 3 gives an overview of the model-checking algorithm, which finds program traces ("counterexamples") to specified locations. Section 4 shows how test vectors are generated from counterexamples, how sufficiently many counterexamples are obtained to guarantee coverage for the resulting test suite, and how the corresponding test driver is constructed from the program. In Section 5 we conclude by presenting some applications and experimental results.

3. Verification with BLAST

3.1. Programs

Syntax. Our representation for a C program is a *control-flow automaton* (CFA), which is a tuple $\langle Q, q_0, X, \texttt{Ops}, \rightarrow \rangle$, where Q is a finite set of control locations, q_0 is the initial control location, X is a set of typed variables, `Ops` is a set of operations on X, and $\rightarrow \subseteq (Q \times \texttt{Ops} \times Q)$ is a finite set of edges labeled with operations. An edge $(q, \texttt{op}, q')$ is also denoted $q \xrightarrow{\texttt{op}} q'$. The set `Ops` of operations contains (1) *basic blocks* of instructions, *i.e.,* finite sequences of assignments `lval = exp`, where `lval` is an lvalue from X (a variable, structure field, or pointer dereference), and `exp` is an arithmetic expression over X; and (2) *assume predicates* `assume(p)`, where `p` is a boolean expression over X (arithmetic comparison or pointer equality), representing a condition that must be true for the labeled edge to be taken. Any C program can be converted to this representation [26].

Currently, the test extension of BLAST has been implemented only for integer variables, *i.e.,* all variables in X have the type integer. Moreover, for ease of exposition, we describe our method only for programs (and CFAs) without function calls; it can be extended to handle function calls in a standard way (and function calls are handled by the BLAST implementation).

Semantics. A *data valuation* is a type-preserving function from X to values. A *region* is a quantifier-free first-order formula over some fixed set of relation and function symbols. We use regions to represent sets of data valuations, *i.e.,* a region r represents all data valuations that satisfy r. Let $\mathcal{R}$ be the set of regions. The semantics of operations is given in terms of the strongest-postcondition operator [10]: $sp(r, \mathtt{op})$ of a region r with respect to an operation $\mathtt{op}$ is the strongest formula (in the implication ordering) whose truth holds after $\mathtt{op}$ terminates when executed in a valuation that satisfies r. For a formula $r \in \mathcal{R}$ and operation $\mathtt{op} \in \mathtt{Ops}$, the formula $sp(r, \mathtt{op}) \in \mathcal{R}$ is syntactically computable; in particular, after Skolemization, the strongest postcondition is again a quantifier-free formula.

For a predicate p over program variables, a *p-trace* is a path $q_0 \xrightarrow{\mathtt{op}_1} \cdots \xrightarrow{\mathtt{op}_n} q_n$ through the CFA such that the formula $sp(\ldots(sp(sp(\mathit{true}, \mathtt{op}_1), \mathtt{op}_2)\ldots), \mathtt{op}_n) \wedge p$ is satisfiable. A location $q \in Q$ is *p-reachable* in C if there is a p-trace ending at q.

3.2. Reachability Trees

Let $T = (V, E, \mathtt{n}_0)$ be a (finite) rooted tree, where each node $\mathtt{n} \in V$ is labeled by a pair $(q, r) \in Q \times \mathcal{R}$, each edge $e \in E$ is labeled by an operation $\mathtt{op} \in \mathtt{Ops}$, and $\mathtt{n}_0 \in V$ is the root node. We write $\mathtt{n} : (q, r)$ if node $\mathtt{n}$ is labeled by location q and region r; we say that r is the *reachable region* of $\mathtt{n}$. If there is an edge from $\mathtt{n} : (q, r)$ to $\mathtt{n}' : (q', r')$ labeled by $\mathtt{op}$, then node $\mathtt{n}'$ is an $(\mathtt{op}, q')$*-child* of node $\mathtt{n}$. The labeled tree T is a *complete reachability tree* for the CFA C if (1) the root is $\mathtt{n}_0 : (q_0, \mathit{true})$, where q_0 is the initial location of the CFA; (2) each internal node $\mathtt{n} : (q, r)$ has an $(\mathtt{op}, q')$-child $\mathtt{n}' : (q', r')$ iff there is an edge $q \xrightarrow{\mathtt{op}} q'$ of C and $sp(r, \mathtt{op}) \Rightarrow r'$; and (3) for each leaf node $\mathtt{n} : (q, r)$, either q has no successors in C, or there are internal nodes $\mathtt{n}_1 : (q, r_1), \ldots, \mathtt{n}_k : (q, r_k)$ such that $r \Rightarrow (r_1 \vee \ldots \vee r_k)$. Intuitively, a complete reachability tree is a finite unfolding of the CFA whose nodes are annotated with regions, and whose edges are annotated with corresponding operations from the CFA.

For a set $F \subseteq V$ of leaf nodes of T, the pair (T, F) is a *partial reachability tree* for C if conditions (1) and (2) hold, and (3$'$) for each leaf node $\mathtt{n} : (q, r)$, either $\mathtt{n} \in F$, or q has no successors in C, or there are internal nodes $\mathtt{n}_1 : (q, r_1), \ldots, \mathtt{n}_k : (q, r_k)$ such that $r \Rightarrow (r_1 \vee \ldots \vee r_k)$. A partial reachability tree is a prefix of a complete reachability tree, where the nodes in F have not yet been unfolded.

Let q be a CFA location and p be a predicate over program variables. A complete reachability tree T for C is *safe* w.r.t. (q, p) if every tree node $\mathtt{n} : (q, r)$ is such that $p \wedge r \Rightarrow \mathit{false}$. A complete reachability tree that is safe w.r.t. (q, p) demonstrates that q is not p-reachable.

Theorem 1 [18] *Let C be a CFA, p a predicate, and T a complete reachability tree for C. For every location q of C, if T is safe w.r.t. (q, p), then q is not p-reachable in C.*

3.3. The Verification Algorithm

The model-checking algorithm of BLAST takes as input a CFA C, a partial reachability tree (T, F), and a pair of (q, p), where q is a target location and p a target predicate. Provided it terminates, the algorithm returns with one of two outcomes: either O_1, a complete reachability tree T' that is safe w.r.t. (q, p), or O_2, a partial reachability tree (T', F') that has a path from the root node $n_0 : (q_0, \mathit{true}) \xrightarrow{\mathtt{op}_1} \cdots \xrightarrow{\mathtt{op}_n} n : (q, \cdot)$ such that $q_0 \xrightarrow{\mathtt{op}_1} \cdots \xrightarrow{\mathtt{op}_n} q$ is a p-trace. In the former case, from Theorem 1 we conclude that q is not p-reachable in C; in the latter case, we shall extract from the program trace a test vector that drives the program to the location q such that at q, the program variables satisfy the predicate p.

The BLAST algorithm implements an abstract–check–refine loop, where abstraction is done lazily [18]. We give only a brief outline of this algorithm. The abstract reachability tree is built in two phases: forward reachability and backward counterexample-driven refinement. At each point, the algorithm maintains a finite set of *abstraction predicates*. In the forward reachability phase, the algorithm searches the program state space to construct a complete reachability tree. Each path in the tree corresponds to a path in the CFA. Each node of the tree is labeled by a reachable region, which is an overapproximation of the actual reachable states of the program along the path from the root in terms of the abstraction predicates. If a complete reachability tree is constructed and no path to q is found in the tree, then the algorithm stops and returns the complete tree (O_1). This phase is guaranteed to either terminate or find a path to q relative to the current set of abstraction predicates [18]. If we find a path to q in the tree, then we proceed to the second phase, which checks if the path to q corresponds to a program trace, or results from the abstraction being too coarse (*i.e.,* we lost too much information by restricting ourselves to a particular set of abstraction predicates). In the former case, we have found a p-trace to q in the tree and we return the current partial tree (O_2). In the latter case, BLAST asks a theorem prover to suggest additional abstraction predicates, which rule out that particular infeasible path, and repeats the first phase with the extra predicates.

4. Testing with BLAST

4.1. The Testing Framework

Testing is usually carried out within a framework comprising (1) a suitable representation of the program, (2) a representation for test vectors, and a set of test vectors called a *test suite*, (3) an *adequacy* criterion that determines whether a test suite adequately tests the program, (4) a test generation procedure that generates an adequate test suite, and (5) a *test driver* that executes the program with a given test vector by automatically feeding input values from the vector.

Programs and tests. We use CFAs as our representation of programs. This representation is very similar to the control-flow graphs [1] used in many testing frameworks. A *test vector* is a sequence of input data required for a single run of the program. This sequence contains the initial values for the formal parameters of the program, and the sequence of values supplied by the environment whenever the program asks for input. In other words, in addition to input values, the test vector also contains a sequence of return values for external function calls. For example, when testing device drivers, the test vector would contain a sequence of suitable return values for all calls to kernel functions made by the driver, and a sequence of values for data read off the device.

Target predicate coverage. Ideally, one would like the test suite to exercise all execution paths of the program ("path coverage"), and thus expose any errors that the program may have. As such test suites will be infinitely large for most programs, the notions of location and edge coverage are used to approximate when a program has been tested sufficiently[25, 29].

We use the following notion of *target predicate coverage*: given a C program in the form of a CFA, and a target predicate p, we say a test vector *covers* a location q of the CFA w.r.t. p if the execution resulting from the vector takes the program into a state where it is in location q and the variables satisfy the predicate p. We deem a test suite adequate w.r.t. p if all p-reachable CFA locations are covered w.r.t. p by some vector in the suite.

For example, consider the program in Figure 3 and the target predicate `uid=0`. The algorithm outputs tests vectors for all locations that the program can reach with the value of `uid` being `0`. As another case, suppose that the target predicate p is $true$. Then the test-generation algorithm outputs test vectors for all *reachable* CFA locations. Furthermore, BLAST reports all CFA locations that are (provably) unreachable by any execution, as dead locations (they correspond to dead code). If we run BLAST on a program with both predicates p and $\neg p$, then for all CFA locations q that can be reached with p either true or false, we obtain *two* test vectors —one that causes the program to reach q with p true, and another one that causes the program to reach q with p false.

The notion of target predicate coverage corresponds to *location coverage* ("node coverage") if $p = true$. For *edge coverage*, for an edge e that represents a branch condition p_e, we can find a test that takes the program to the source location of e with the state satisfying the predicate p_e, thus causing the edge e to be traversed in the subseqent execution step. We can similarly adapt our technique to generate tests for other testing criteria [19, 29]; we omit the details.

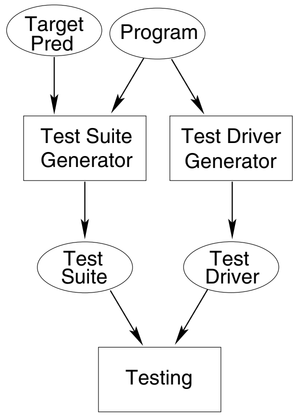

Figure 5. Test flow

Test flow. The overall testing framework as implemented in BLAST is shown in Figure 5. A program and a target predicate are fed as inputs. The *test-suite generation* procedure takes the program and the target predicate as input and produces an adequate test suite, *i.e.,* one such that every p-reachable CFA location is covered w.r.t. p by some test vector in the suite. The *test-driver generation* procedure takes the program as input and produces another program, the test driver, that runs the original program on the test inputs. The test driver has a wrapper function for all external function calls, which returns values from the test vector to the program.

In the following subsections we describe how to generate an adequate test suite, and how to generate a test driver that can execute the program on the test vectors in the suite in order to allow developers to see how the program behaves on the generated tests.

4.2. Test Suite Generation

Recall that the model-checking algorithm described in the previous section takes as input a partial reachability tree and a pair (q, p) of target location and target predicate, and returns either with outcome O_1, a complete reachability tree T that is safe w.r.t. (q, p), or with outcome O_2, a partial reachability tree that contains a node $\mathtt{n} : (q, \cdot)$ such that the path from the root to `n` corresponds to a p-trace ending at q. Given a program and a target predicate p, the test-suite generation now proceeds as follows.

Step 1. The locations of the CFA are numbered in depth-first order, and put into a *worklist* in decreasing order of the numbering (*i.e.,* the location numbered last in DFS order is first on the worklist). We create an initial partial reachability tree (T, F), where T is a tree with a single node, namely the root $\mathtt{n}_0 : (q_0, true)$, and F is the singleton set containing $\mathtt{n}_0$. The initial test suite is the empty set.

Step 2. If the worklist is empty, then we return the current test suite; otherwise let q be the first CFA location in the worklist. We invoke the model checker with the partial reachability tree (T, F) and (q, p), and we update the current reachability tree with the result of the model checking.

Step 3. If the model checker returns with outcome O_1, then we conclude that for all locations q' such that the new reachability tree (T', F') is safe w.r.t. (q', p), no test vector exists, and so we delete all such locations from the worklist. Otherwise, if the model checker returns with outcome O_2, then we have a p-trace to the location q. We use this trace to compute a test vector that covers the location q w.r.t. p using a procedure described below. We add this vector to the test suite, and remove q from the worklist. In both cases, we go back to **step 2**.

It can be shown that upon termination, the above procedure returns a test suite that is adequate w.r.t. p according to our criterion of target predicate coverage.

We incorporate several optimizations to the above loop. First, when a test vector is found, we can additionaly find (by symbolically executing the program on the vector) which other locations it covers, and we remove those locations from the worklist. Second, the model-checking algorithm uses heuristics to choose the next node to unfold in the partial reachability tree. The nodes that need to be unfolded are partitioned into those that have been covered by a vector in the current test suite, and those that are still uncovered. The model checker unfolds uncovered nodes first, and it unfolds covered nodes only if there remain no uncovered nodes. A node that has been covered by a previous test may still need to be unfolded, because a path to an (as yet) uncovered location may go through it. Third, the user has the option to give a time-out for the model checking. Thus in **step 3**, if instead of O_1 or O_2, the model checker times out, then we give up on the location q, by deleting it from the worklist and going back to **step 2**.

We have found these optimizations to be essential for the algorithm to work on large programs.

Generating tests from traces

When model checking in **step 2** ends with outcome O_2, the resulting tree contains a path to a node $\mathtt{n} : (q, r)$ such that the path corresponds to a p-trace ending at q. We now describe how to extract from this trace a test vector that, when fed to the program, takes it to location q satisfying the target predicate p.

To decide whether a path in the tree represents a program trace (*i.e.,* the path is feasible), the model checker encodes the path symbolically as a set of constraints on the program variables such that the path corresponds to a trace iff the set of constraints is satisfiable [23, 9, 16]. The symbolic evaluator in BLAST handles arithmetic operators as well as aliasing relationships between program variables. Once path feasibility has been established, we call the satisfiable conjunction of constraints that arise from the path, the *trace formula* (TF). We use a decision procedure to produce a satisfying assignment for the variables of the TF. From the satisfying assignment we build a test vector that drives the program to the target location and target predicate. Instead of describing how the procedure works for all C programs, we restrict ourselves here to programs without function calls and without pointers, and we assume that all variables are integers; see [16] for the complete procedure.

Constraint generation. To denote the values of program variables at various point in a trace, we introduce special constants, each of which is a pair consisting of a variable name and a natural number. For example, the pair $\langle \mathtt{x}, 0\rangle$ is a constant denoting the value of the variable x at the beginning of the trace. We use lvalue maps to generate these special constants as needed. An *lvalue map* is a function θ from the set $Lvals$ of lvalues to $\mathbb{N}$. At every point in the trace, the pair $\langle \mathtt{l}, \theta(\mathtt{l})\rangle$ is the special constant that denotes the value of the lvalue l (variable, structure field, or pointer dereference) at that point of the trace. Whenever an lvalue l is updated, we update the lvalue map so that subsequently a fresh constant is used to denote the value of l. More precisely, the operator Upd: $(Lvals \to \mathbb{N}) \to 2^{Lvals} \to (Lvals \to \mathbb{N})$ takes an lvalue map θ and a set L of lvalues, and returns a new map θ' such that $\theta'.\mathtt{l} = \theta.\mathtt{l}$ if $\mathtt{l} \notin L$, and otherwise $\theta'.\mathtt{l} = i_l$ for a fresh number i_l. The function Sub takes an lvalue map θ and an lvalue l, and returns the pair $\langle \mathtt{l}, \theta(\mathtt{l})\rangle$. The function $Sub.\theta$ is extended to expressions and predicates in the natural way. A *new* lvalue map is one whose range is disjoint from all other lvalue maps.

Given a trace, the corresponding TF is generated inductively by the function Con, as a conjunction of constraints. An *atomic operation* is either an assignment to an lvalue, or an assertion about lvalues in the form of an assume predicate. The function Con takes a pair consisting of an lvalue map θ and a formula φ (representing a partial TF), as well as a finite sequence t of atomic operations (representing the remainder of the trace), and returns a pair consisting of a new lvalue map and a new formula. Let θ_0 be the lvalue map that maps all variables to 0. Given a sequence t of atomic operations, suppose that $Con.(\theta_0, true).t = (\theta, \varphi)$. It can be shown, by induction on the length of t, that the formula φ is satisfiable iff t can be executed, *i.e.,* all assertions in t evaluate to true. In other words, if φ is satisfiable, then t corresponds to a trace of the program, and φ is its TF.

The function Con is defined in the first three rows of Figure 7. For an assignment, we first update the lvalue map to introduce a new constant denoting the new value of x, and the constraint for the operation states that the new constant

`Example() {`				
`  if (y == x)`	`assume (y=x)`	$\langle y,0\rangle = \langle x,0\rangle$	$\langle x,0\rangle \mapsto 0$	$x \mapsto 0$
`    y ++ ;`	`y = y+1`	$\langle y,1\rangle = \langle y,0\rangle + 1$	$\langle y,0\rangle \mapsto 0$	$y \mapsto 0$
`  if (z <= x)`	`assume !(z<=x)`	$\neg\langle z,0\rangle \leq \langle x,0\rangle$	$\langle y,1\rangle \mapsto 1$	$z \mapsto 2$
`    y ++ ;`			$\langle z,0\rangle \mapsto 2$	
`  a = y - z;`	`a = y-z`	$\langle a,2\rangle = \langle y,1\rangle - \langle z,0\rangle$	$\langle a,2\rangle \mapsto -1$	
`  if (a < x)`	`assume (a<x)`	$\langle a,2\rangle < \langle x,0\rangle$		
`    LOC:`				
`}`				
(a) Program	(b) Trace	(c) Trace formula	(d) Assignment	(e) Test vector

Figure 6. Generating a test vector

t	$Con.(\theta, \varphi).t$
$\mathtt{l} = \mathtt{e}$	$(\theta', \varphi \wedge (Sub.\theta'.\mathtt{l} = Sub.\theta.\mathtt{e}))$ where $\theta' = Up\ d\theta.\{\mathtt{l}\}$
$\mathtt{assume}(\mathtt{p})$	$(\theta, \varphi \wedge Sub.\theta.\mathtt{p})$
$t_1;t_2$	$Con.(Con.(\theta, \varphi).t_1).t_2$

Figure 7. Building a trace formula

for x has the same value as the expression e (with appropriate constants plugged in). For an assertion, the generated constraint stipulates that the constants at that point satisfy the assume predicate. The third row shows how the function Con works on a sequence of operations.

Figure 6(a) shows a program, and Figures 6(b) and 6(c) show, respectively, a trace to the program location LOC, and the TF for that trace. The constraint for each atomic operation of the trace is shown to the right of the operation; the TF is the conjunction of all constraints.

Tests from constraints. The TF is a conjunction of constraints about special constants of the form $\langle l, i\rangle$, each of which is an arithmetic fact that relates the values of program variables at various points in the trace. In our experience, many programs generate *linear* arithmetic constraints. Thus, we can find a satisfying assignment for the TF using an integer linear programming (ILP) solver. For a satisfiable formula φ, let $S.\varphi$ be a satisfying interpretation of all special constants that occur in the formula. A test vector that exercises the trace t is obtained by setting every input variable x of the program to the initial value $S.\varphi.\langle x, 0\rangle$.

Figure 6(d) shows a satisfying interpretation for the special constants of the TF of Figure 6(c). It is easy to check that if we set the inputs initially to "`x=0, y=0, z=2`," then the program follows the trace of Figure 6(b). The generated test vector is shown in Figure 6(e).

Pointers. The above method can be extended to programs with pointers. We first generate the TF from whose satisfying assignment we obtain a test vector as described above; the details of the TF generation are given in in [16]. The resulting TF contains disjuncts due to possible aliasing. There are two ways to deal with this. First, we can convert the formula to DNF and check each disjunct separately, and on finding a satisfiable disjunct, we can extract a test vector from a satisfying assignment of the disjunct. Second, we can use efficient decision procedures for propositional satisfiability [24] to find a possibly satisfiable disjunct, and then use the ILP solver to find a satisfying assignment for that disjunct, from which again the tests are computed as discussed above. Many off-the-shelf decision procedures already incorporate this style of propositional reasoning [11, 32, 16]. Of course, there are programs for which our constraint-based test-generation strategy fails because the given constraint language is not expressive enough.

Library calls. If a trace contains library calls whose source code is not available for analysis, or asks for user input, the constraint generation assumes that the library call or the user can return any value. Thus, some of our tests may not be executable if the library calls always return values from some subset of possible values. In this case, the user can model postconditions on library calls by writing stub functions that restrict the possible return values.

4.3. Test Driver Generation

Recall that a test vector generated by BLAST is a sequence of integer values (our test-vector generation is currently restricted to integer inputs): these are the values that are fed to the program by the test driver during the actual test; they include the initial values for all formal parameters and the return values for all external function calls.

The test-driver generator takes as input the original program and instruments it at the source-code level to create a test driver containing the following components: (1) a wrapping function, (2) a test-feeding function, and (3) a modified version of the original program. The test driver can then be compiled and run to examine the behavior of the original program on the test suite. It can be run on each individual test vector and the user can study the resulting dynamic behavior as she pleases, by using a debugger for example.

The wrapper is the main procedure of the driver: it reads a test vector and then calls the main function of the original program, passing it initial values for the parameters from the vector. The driver generator modifies the code of the program being tested by replacing every call to an external function with a call to the special test-feeding function. The test-feeding function reads the next value from the test vector and returns it. We are guaranteed that the vector will have taken the program to the target when the test-feeding function has consumed the entire vector. Hence, once the test vector is consumed, the feeder returns arbitrary values.

5. Applications and Experiments

We ran BLAST to generate test suites for several programs. We used two sets of benchmark programs: a set of Microsoft Windows device drivers, and two security-critical programs. The results are summarized in Table 1. The programs `kbfiltr`, `floppy`, `cdaudio`, `parport`, and `parclass` are Microsoft Windows device drivers. The program `ping` is an implementation of the ping utility, and `ftpd` is a Linux port of the BSD implementation of the ftp daemon. The experiments were run on a 3.06 GHz Dell Precision 650 with 4 GB of memory.

We present results for checking the reachability of code. In these experiments, the specification was trivial (*i.e.,* the target predicate was *true*): we checked which program locations are *live* (reachable by some execution) and *dead* (not reachable by any execution), and we generated test vectors that cover all live locations. Syntactically plausible executions (for example, control-flow paths, or data flows) may not be semantically possible, for example, due to correlated branching [3]. This is called the *infeasibility problem* in testing [29, 21]. The usual approach to deal with infeasible paths is to argue manually on a case-by-case basis, or to resort to adequacy scores (the percentage of all static paths covered by tests). By using BLAST we can automatically detect dead code, and generate tests for live code.

In the table, *LOC* refers to lines of code. CFAs represent programs compactly; each basic block is a single edge. In the table, the column *CFA locations* shows the number of locations of the CFA which are syntactically reachable by exploring the corresponding call graph of the program. *Live* is the number of reachable locations, *Dead* is the number of unreachable locations, and *Fail* is the number of locations on which our tool failed. Ideally, the total number of CFA locations is equal to the sum of the live and dead locations. However, in our tool we set a time-out for each location. So in practice, the tool fails on a small percentage of locations. The failure is due both to time-outs, and to not finding suitable predicates for abstraction. In our experiments, we set the time-out to 10 minutes per location.

The column *Tests* gives the number of tests generated. The implementation does not run the model checker for a location that is already covered by a previous test. Thus, the number of tests is usually much smaller than the number of reachable locations. This is especially apparent for the larger programs. *Total* is the total number of predicates, over all locations, generated by the model-checking process. *Average* is the average number of predicates active at any one program point. The average number of predicates at any location is much smaller than the total number of predicates, thus confirming our belief that local and precise abstractions can scale to large programs [18, 16]. *Time* gives the running time rounded to minutes (except for `ftpd`, where the tool ran for two overnight runs).

We found many locations that were not reachable because of correlated branches. For example, in `floppy`, we found the following code:

```
driveLetterName.Length = 0;
// cut 15 lines
...
if (driveLetterName.Length != 4 ||
  driveLetterName.Buffer[0] < 'A' ||
  driveLetterName.Buffer[0] > 'Z') {
    ...
}
```

Here, the predicate `driveLetterName.Length != 4` is true; so the other tests are never executed. Another reason we get dead code is that certain library functions (like `memset`) make many comparisons of the size of a structure with different built-in constants. At run time, most of these comparisons fail, giving rise to many dead locations.

While the table reports only experiments that check for unreachable code, we ran BLAST also on several small examples with security specifications in order to find which parts of a program can be run with root privileges. Unfortunately, most security programs make recursive calls, and our previous implementation of BLAST did not support recursive function calls. We are currently implementing a new version that does handle recursive calls. We are also optimizing our test-generation procedure to generate tests directly from the internal data structures of the model checker.

Table 1. Experimental results

Program	LOC	CFA locations	Locations			Tests	Predicates		Time
			Live	Dead	Fail		Total	Average	
kbfiltr	5933	381	298	83	0	39	112	10	5 min
floppy	8570	1039	780	259	0	111	239	10	25 min
cdaudio	8921	968	600	368	0	85	246	10	25 min
parport	12288	2518	1895	442	181	213	509	8	91 min
parclass	30380	1663	1326	337	0	219	343	8	42 min
ping	1487	814	754	60	0	134	41	3	7 min
ftpd	8506	6229	4998	566	665	231	380	5	1 d

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] T. Ball and S.K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proc. POPL*, pages 1–3. ACM, 2002.

[3] R. Bodik, R. Gupta, and M.L. Soffa. Interprocedural conditional branch elimination. In *Proc. PLDI*, pages 146–158. ACM, 1997.

[4] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *Proc. ISSTA*, pages 123–133. ACM, 2002.

[5] S. Chaki, E.M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *Proc. ICSE*, pages 385–395. IEEE, 2003.

[6] H. Chen and D. Wagner. MOPS: An infrastructure for examining security properties of software. In *Proc. CCS*, pages 235–244. ACM, 2002.

[7] H. Chen, D. Wagner, and D. Dean. Setuid demystified. In *Proc. Security Symp.*, pages 171–190. Usenix, 2002.

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

[9] L. Clarke and D. Richardson. Symbolic evaluation methods for program analysis. In *Program Flow Analysis: Theory and Applications*, pages 264–300. Prentice-Hall, 1981.

[10] E. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[11] J.-C. Filliâtre, S. Owre, H. Ruess, and N. Shankar. ICS: Integrated canonizer and solver. In *Proc. CAV*, LNCS 2102, pages 246–249. Springer, 2001.

[12] A. Gotlieb, B. Botella, and M. Rueher. Automatic test data generation using constraint solving techniques. In *Proc. ISSTA*, pages 53–62. ACM, 1998.

[13] E. Gunter and D. Peled. Temporal debugging for concurrent systems. In *Proc. TACAS*, LNCS 2280, pages 431–444. Springer, 2002.

[14] N. Gupta, A. Mathur, and M.L. Soffa. Generating test data for branch coverage. In *Proc. ASE*, pages 219–228. IEEE, 2000.

[15] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific static analyses. In *Proc. PLDI*, pages 69–82. ACM, 2002.

[16] T.A. Henzinger, R. Jhala, R. Majumdar, and K. McMillan. Abstractions from proofs. In *Proc. POPL*, pages 232–244. ACM, 2004.

[17] T.A. Henzinger, R. Jhala, R. Majumdar, and M. Sanvido. Extreme model checking. In *International Symposium on Verification: Theory and Practice*, LNCS 2772. Springer, 2003.

[18] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. POPL*, pages 58–70. ACM, 2002.

[19] H.S. Hong, S.D. Cha, I. Lee, O. Sokolsky, and H. Ural. Data flow testing as model checking. In *Proc. ICSE*, pages 232–243. IEEE, 2003.

[20] D. Jackson and M. Vaziri. Finding bugs with a constraint solver. In *Proc. ISSTA*, pages 14–25. ACM, 2000.

[21] R. Jasper, M. Brennan, K. Williamson, B. Currier, and D. Zimmerman. Test data generation and infeasible path analysis. In *Proc. ISSTA*, pages 95–107. ACM, 1994.

[22] S. Khurshid, C. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proc. TACAS*, LNCS 2619, pages 553–568. Springer, 2003.

[23] J. King. Symbolic execution and program testing. *Comm. ACM*, 19:385–394, 1976.

[24] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc. DAC*, pages 530–535. ACM, 2001.

[25] G.J. Myer. *The Art of Software Testing*. Wiley, 1979.

[26] G.C. Necula, S. McPeak, S.P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proc. CC*, LNCS 2304, pages 213–228. Springer, 2002.

[27] D. Peled. *Software Reliability Methods*. Springer, 2001.

[28] D. Peled. Model checking and testing combined. In *Proc. ICALP*, LNCS 2719, pages 47–63. Springer, 2003.

[29] M. Pezze and M. Young. *Software Test and Analysis: Process, Principles, and Techniques*. Manuscript, 2003.

[30] C. Ramamoorthy, S.B. Ho, and W. Chen. On the automated generation of program test data. *IEEE Trans. Software Eng.*, 2:293–300, 1976.

[31] F.B. Schneider. *Enforceable Security Policies*. Tech. Rep. TR98-1664, Cornell, 1999.

[32] A. Stump, C. Barrett, and D. Dill. CVC: A cooperating validity checker. In *Proc. CAV*, LNCS 2404, pages 500–504. Springer, 2002.