Generating Tests from Counterexamples

Adam J. Chlipala Thomas A. Henzinger Ranjit Jhala Rupak Majumdar
UC Berkeley
{adamc,tah,jhala,rupak}@eecs.berkeley.edu

Abstract

We extend the software model checker Blast *to automatically generate test suites that guarantee full coverage with respect to a given predicate, or with respect to a more general temporal safety property. More precisely, given a C program and a predicate p, extended* Blast *determines the set L of program locations in which the predicate can be true, and automatically generates a set of test cases that exhibit the truth of p at all locations in L. We have used extended* Blast *to generate test suites for security disciplines, and o detect dead code, in C programs with up to 30K lines of code. The analysis and test-case 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 [1, 13]. The input to the model checker is both the program source and a monitor automaton, which observes if a program trace violates a temporal safety specification, such as adherence to a locking or security discipline [10, 2, 22]. The checker attempts to verify a program abstraction, and if the verification fails, it produces a counterexample in the form of an abstract trace that violates the specification. If the abstract trace does not correspond to a concrete error trace —that is, if the counterexample is spurious— then the abstraction is automatically refined in a way that removes the spurious counterexample. The entire process is repeated until either a genuine error trace is found, or the absence of such traces is guaranteed. In this way, Windows and Linux device drivers with more than 100K lines of code have been checked without user intervention, and without generating false positives.

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 (counterexample) that violates a given temporal property, but in the set of *all* program locations where the property may be violated. For instance, given a predicate p, the programmer may wish to know the set of all program locations ℓ that can be reached such that p is true at ℓ. For example, when checking the security properties of a program it is useful to find the locations where the program has root privilege. We show that the model checker Blast can be extended to provide this kind of information. As a special case (take p to be the predicate that is always true), extended 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 ℓ is reachable such that the predicate p is true at ℓ, then from the program trace that exhibits p at ℓ the tool automatically produces a test case that witnesses the truth of p at ℓ. This feature enables the software engineer to pose reachability queries about the behavior of the program, and to automatically generate test cases that satisfy the queries [19]. In particular, for a predicate p and its negation, the tool automatically generates for each program location ℓ, if p is always true at ℓ, a test case that exhibits p at ℓ, if p is always false at ℓ, a test case that exhibits $\neg p$ at ℓ, and if p may be true or false at ℓ, then two test cases, one that exhibits the truth of p at ℓ, and another test case that exhibits the falsehood of p at ℓ. 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 case are each covered by two test cases with different outcomes. As a side effect, of course, extended Blast can be used to generate test suites that cover all reachable program locations (take p to be true everywhere).

Often a single test case covers the truth of p at many locations, and the falsehood of p at others, and Blast produces a minimal set of test case that pro-

vides the desired information. For this, it is essential that BLAST uses incremental model-checking technology [13, 12], which reuses partial proofs and counterexamples as much as possible. We have used this extension of BLAST to query C programs with 30K lines of code about locking disciplines, security disciplines, and dead code, and to automatically generate corresponding test suites.

This is not the first attempt to use model-checking technology for automatic test-case generation. However, to our knowledge, the previous work in this area has followed very different directions. For example, the approach advocated by [15] uses symbolic execution to generate test cases that cover all truth value combinations of a given set of predicates, but it does not guarantee coverage in terms of program locations. The approach of [14] considers fixed boolean abstractions of the input program, and does not automatically refine the abstraction to the degree necessary to generate test cases that cover all program locations for a given set of observable predicates. Peled [20] 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 [9] is the closest to our approach in that it generates test cases from traces, however, these traces are not found by automatic abstraction refinement.

2. Example

Figure 1 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 ids, each process has a real id, and an effective 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 [5], but is sufficient for exposition.

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 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,

```
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 1. The program `Example`.

```
global int _E = 0;

event {
  pattern { $? = seteuid($1); }
  action { _E = $1; }
}
```

Figure 2. A simple seteuid specification

then the program does some work (in the function call `work_and_drop_privileges`) and sets the effective user id back to `saved_uid` (the real user id of the process) at the end (line L9). We model the `getuid` function to return a nonzero value (modeling that the real user id of the process is not zero), and the `geteuid` function to return either a zero or a nonzero value nondeterministically. These are omitted from the figure for simplicity.

Figure 2 shows a simple specification in our language that describes an automaton that keeps track of security privileges of the program. The automaton tracks a single global (specification) variable $\_E$ denoting the current effective user id (and hence privilege level) of the system. If $\_E$ equals zero, the process runs with root privileges. Initially, this value is set to 0. The specification describes state changes of the automaton through *events* that cause a transition. For example,

```
L10: saved_uid = getuid();
  /* call to getuid omitted */
L11: saved_euid = geteuid();
  /* call to geteuid omitted */
  /* geteuid returns 0 */
L12: seteuid(saved_uid);
  /* call to seteuid omitted */
  /* _E = saved_uid */
L14: tmp = get_root_privieleges();
  L1: if (saved_euid!=0) /* fails */
  L3: seteuid(saved_euid);
  /* _E = 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: /* _E = 0 */
```

Figure 3. (a) Annotated `main` produced by BLAST. All filled nodes are reachable with $\_E = 0$. (b) A trace generated by BLAST

every time `seteuid` is called, the value of $\_E$ is updated to be equal to the parameter passed to `seteuid` in the program. This happens in the single `event` block of the automaton. The `pattern` keyword denotes a code pattern in the source program that must be matched, and the `action` keyword denotes how the specification state must be modified when the pattern matches. Given a C program and a specification, BLAST automatically instruments the program with the specification.

Good programming practice requires that certain system calls that run untrusted programs should not be made with root privileges [4], since 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 actually run with root privileges.

We use the model checker BLAST in test mode to check which code lines can be executed with root privileges. Precisely, we ask the model checker to output all locations that are reachable in the program such that the automaton state is $\_E = 0$. For each such location, we also demand a program trace that reaches the location with the automaton simultaneously in the state $\_E = 0$, and additionally a test case that will cause this trace to be executed. The resulting annotated control flow automaton is shown in Figure 3. The tree shows, surprisingly, that the `execv` system call can be executed with root privileges. An inspection of the symbolic trace generated by the model checker shows that there is a bug in the `work_and_drop_privileges` function: in case the call to `fopen` fails, the function returns without dropping root privileges.

3. A Language for Specifying Observers

3.1 Overview

The BLAST specification language allows for the description of temporal safety properties of C programs in a natural way, based on syntactic pattern matching of relevant kinds of C code. Each specification file defines an observer automaton that tracks changes to the state of an executing C program and possibly signals an error if an invariant is violated. Rather than being limited to finite states, an observer automaton may have global state variables of any C types, as well as typestate information associated with an observed program's in-scope variables. The author of a specification is not forced to discretize its state space unnaturally; rather, he may rely on the BLAST verification engine to abstract the specification and program as necessary while maintaining soundness.

An observer automaton has a collection of syntactic patterns that, when matched in the current execution point of a C program, trigger transitions in the observer. The precise transitions used may depend on arbitrary C expressions and statements that may refer to pattern variables matched in the triggering patterns. One possible transition is to trigger an error. When BLAST is run with a C program and a specificiation it is to satisfy, it uses lazy abstraction and refinement to attempt to find a path to one of these error transitions in the program-specification system.

3.2 Abstract Syntax

3.2.1 Global specification variables

The global (i.e., object-independent) control state of the specification automaton is defined by valuations to a set V of typed global specification variables. The type of a variable $v \in V$ may be any C type. Variables in V can be read and written by events. For example, in the case of a specification that "at most 3 lock calls may be made at once," a global specification variable `numCalls` of type `int` might be used to track the number of lock calls made.

3.2.2 Typestate

In addition to global variables, the specification language allows maintaining control information on a per-object basis. Each distinct C type may have typestate declared to be associated with any value of that type. This information is maintained through a set S of typed *typestate variables*, each having a C type. The set of

typestate variables for a particular C type is declared with

```
shadow ctype { structure  }
```

where `ctype` is a C type not previously given typestate, and `structure` is a C struct definition augmented with initial values for all fields. This list of fields specifies the typestate variables for `ctype`.

3.2.3 Initialization of state

All global variables and shadow information declarations come with initializers. Together, these specify the initial state of the specification automaton. When a new object of type `ctype` is allocated, the typestate variables associated with `ctype` are initialized to the initial values specified by their initializers.

3.2.4 Events

An *event* specifies how the specification state changes due to program actions. An event consists of up to four parts: a pattern, a temporal specifier, an guard, and an action.

Patterns provide a way to identify source actions that are relevant to an event. They are drawn from a simplified subset of C statements consisting only of assignments and function calls involving side-effect free expressions. They may refer to variables named $\$i$, for $i \geq 1$. Each such variable may appear at most once in a pattern, and BLAST uses simple type inference to determine a type for each that appears. There is also a pattern variable `$?`. When a program is being verified to match a specification, at each point in its execution, the piece of a C statement that it is about to execute is checked against all of the specification's patterns. The check is done with syntactic pattern matching, with any of the special pattern variables matching any expressions in the actual C statement.

The *temporal specifier* is either `before` or `after`. It specifies whether the event describes a transition of the observer before or after a source code AST node matching the pattern is executed.

A *guard* is a boolean condition expressed as a C expression that may refer to no variables but global specification variables, numbered pattern variables $\$i$, and typestate information associated with either of these. A guard states a logical invariant that must hold every time the corresponding pattern is matched. If a guard is omitted, it is assumed to be `true`. If at run time, the guard of an event is violated, the safety property expressed by the specification is deemed to be violated.

An *action* is a sequence of C statements that is executed whenever a pattern is matched, with the following restrictions: First, the only variables it may read are global program variables, global spec variables, the numbered $\$i$ pattern variables found in the corresponding pattern, and the typestate associated with any of these. Second, the only variables an action may write are global spec variables, numbered pattern variables, and typestate information associated with these. In particular, an action cannot update program state.

In a given execution of an action, the numbered pattern variables refer to the expressions with which they unified in the pattern matching that triggered the event. This means that it is invalid for an action to attempt to modify a pattern variable that may have matched a C expression that is not an lvalue.

3.3 Semantics

Since syntatic pattern matching on literal C code must deal with code structuring issues that are not necessarily connected with semantics, BLAST performs pattern matching in the simplified statement language mentioned earlier in the discussion of patterns. Recall that this language requires programs to be broken into two kinds of actions: variable assignments and function calls. In our implementation, a sound transformation from C programs to the simplified statement language is performed by Cil [18]. The actual program verification is performed on a transition system with nodes corresponding to program states and edges labelled with a superset of the language of simplified statements. The other kinds of edges are not relevant to this discussion.

The specification observer is run in parallel to the program, which has been translated to the simplified form. At every point in the execution, for every event, the observer checks if the program is about follow a transition labelled with a statement that matches the event's pattern (for `before` events) or the program has just followed such a transition (for `after` events).

When such a check succeeds, the event's guard is evaluated with the standard C semantics, with the numbered pattern variables replaced by the expressions that they matched. If the guard evaluates to `false`, then the event's repair is executed with standard C semantics with the same pattern variable values as above. Since the default repair is a call to `abort()`, an event with no repair specified will at this point make such a call and enter an error state. BLAST will then determine whether the error trace it has found is feasible. If so, it returns it to the user and halts. If not, it refines its abstraction of the program and specification and continues state exploration from the appropriate

```
#include <locking_functions.h>

global int locked = 0;

event {
  pattern { $? = init(); }
  action { locked = 0; }
}

event {
  pattern { $? = lock(); }
  guard { locked == 0 }
  action { locked = 1; }
}

event {
  pattern { $? = unlock(); }
  guard { locked == 1 }
  action { locked = 0; }
}
```

Figure 4. A specification enforcing correct use of a global lock

point. When a pattern matches and its guard evaluates to `true`, the pattern's action is executed in the same manner as a repair would be.

Every pattern in the specification is checked in this way in the same order as they are listed in the specification, and every one that matches transforms the system's state in the way described above. These checks and transitions take place in between the normal execution of the program transition system, and the end result of this interleaved traversal is that either an error state is found (with results described above) or the entire abstract state space is explored without reaching an error state, proving that the program satisfies the specification.

3.4 Example specification files

3.4.1 A global lock

The specification in Figure 4 models correct usage of abstract global locking functions. A global variable is created to track the status of the lock. Simple events match calls to the relevant functions. The event for `init` initializes the global variable. The other two events ensure that the lock is in the right state before making a function call. When these checks succeed, the global variable is updated and execution proceeds. When they fail, an error is signalled.

The `$?`'s above match either a variable to which the result of a function call is assigned or the absence of such an assignment, thus making the patterns cover all possible calls to the functions.

```
#include <sys/types.h>
#include <unistd.h>
#include <pwd.h>
#include <stdlib.h>

global int __E__ = 0;

event {
  pattern { $? = seteuid($1); }
  action { __E__ = $1; }
}

event {
  pattern { $? = system($?); }
  guard { __E__ != 0 }
}
```

Figure 5. A specification enforcing a simplified model of UNIX `setuid` semantics

3.4.2 Simplified `seteuid` and `system`

The specification in Figure 5 models the requirement that a setuid program should not call the `system` function until it has changed the effective uid to a nonzero value. The `$1` in the `seteuid` patterns will match any parameter, including the result of a complicated series of function calls. Here `$?` is used as a function parameter to match all remaining actual parameters.

4. The BLAST Algorithm

4.1. Programs and Semantics

We store C programs internally as control flow automata. A *control flow automaton* (CFA) C is a tuple $\langle Q, q_0, X, \mathtt{Ops}, \rightarrow \rangle$, 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 $\rightarrow \subseteq (Q \times \mathtt{Ops} \times Q)$ is a finite set of edges labeled with operations. An edge $(q, \mathtt{op}, q')$ is also denoted $q \xrightarrow{\mathtt{op}} q'$. The set `Ops` of operations contains (1) *basic blocks* of instructions, that is, finite sequences of assignments `lval = exp`, where `lval` is an lvalue from X (i.e., a variable, structure field, or pointer dereference), and `exp` is an arithmetic expression over X; and (2) *assume predicates* [p], where `p` is a boolean expression over X (arithmetic comparison or pointer equality), representing a condition that must be true for the edge to be taken. For ease of exposition we describe our method only for CFAs without function calls; the method can be extended to handle function calls in a standard way (and function calls are handled by the BLAST implementation).

The set $\mathcal{V}_X$ of (*data*) *valuations* over the variables X contains the type-preserving functions from X to values. A *region* is a set of data valuations. Let $\mathcal{R}$ be the set of regions. We use quantifier-free first-order formulas over some fixed set of relation and function symbols to represent regions. The semantics of operations is given in terms of the strongest-postcondition operator [7]: $sp(r, \mathtt{op})$ of a formula r with respect to an operation $\mathtt{op}$ is the strongest formula 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. A location $q \in Q$ is *reachable from a precondition* $Pre \in \mathcal{R}$ if there is a finite path $q_0 \xrightarrow{\mathtt{op}_1} q_1 \xrightarrow{\mathtt{op}_2} \cdots \xrightarrow{\mathtt{op}_n} q_n$ in the CFA and a sequence of formulas $r_i \in \mathcal{R}$, for $0 \leq i \leq n$, such that $q_n = q$, $r_0 = Pre$, $r_n \not\Leftrightarrow false$, and $sp(r_i, \mathtt{op}_{i+1}) = r_{i+1}$ for all $0 \leq i < n$. The witnessing path is called a *feasible* path from (q_0, Pre) to q. We write $sp(r, \mathtt{op}_1\mathtt{op}_2 \ldots \mathtt{op}_n)$ to denote $sp(\ldots sp(sp(r, \mathtt{op}_1), \mathtt{op}_2) \ldots)$.

4.2. 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 n is labeled by control location q and region r; in that case, we say that r is the *reachable region* of n and write $reg(n) = r$. 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 an *abstract reachability tree* for the CFA C if (1) the root $n_0 : (q_0, r_0)$ is labeled by the initial location q_0 of the CFA; (2) each internal node $n : (q, r)$ has an $(\mathtt{op}, q')$-child $n' : (q', r')$ 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 $n : (q, r)$, either q has no successors in C, or there are internal nodes $n_1 : (q, r_1), \ldots, n_k : (q, r_k)$ such that $r \Rightarrow (r_1 \vee \ldots \vee r_k)$. In the latter case, we say that n is *covered* by $n_1, \ldots, n_k$. Intuitively, an abstract 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 $n : (q, r)$, either $n \in F$, or q has no successors in C, or there are internal nodes $n_1 : (q, r_1), \ldots, n_k : (q, r_k)$ such that $r \Rightarrow (r_1 \vee \ldots \vee r_k)$. Intuitively, a partial reachability tree is a prefix of an abstract reachability tree, where the nodes in F have not yet been explored. Of course, if $F = \emptyset$, then (T, F) is an abstract reachability tree. To distinguish abstract reachability trees from partial reachability trees, we refer to the former as *complete* reachability trees.

An *error function* $\mathcal{E}$: $Q \to \{0, 1\}$ identifies a set $\mathcal{E}^{-1}(1) \subseteq Q$ of error locations of the CFA C. Every safety property φ of a program P can be compiled into an error function on the CFA C that results from composing the CFA of P with a monitor automaton for φ. The automaton C is *safe* with respect to the precondition $Pre \in \mathcal{R}$ and error function $\mathcal{E}$ if no location q with $\mathcal{E}(q) = 1$ is reachable from Pre. An abstract reachability tree T for C is *safe* with respect to the precondition $Pre \in \mathcal{R}$ and error function $\mathcal{E}$ if (1) the root has the form $n : (q_0, Pre)$ and (2) for all nodes of the form $n : (q, r)$ with $\mathcal{E}(q) = 1$, we have $r \Leftrightarrow false$. A partial reachability tree (T, F) is *safe* with respect to the precondition Pre and error function $\mathcal{E}$ if condition (1) holds, and (2′) for all nodes of the form $n : (q, r)$ with $\mathcal{E}(q) = 1$, either $n \in F$ or $r \Leftrightarrow false$. A safe abstract reachability tree witnesses the correctness of the CFA for the precondition Pre and error function $\mathcal{E}$, and the reachable regions that label its nodes provide program invariants. Theorem 1 makes this precise.

Theorem 1 [13] *Let C be a CFA, Pre a precondition, and $\mathcal{E}$ an error function for C. If there exists an abstract reachability tree for C which is safe with respect to Pre and $\mathcal{E}$, then C is safe with respect to the precondition Pre and error function $\mathcal{E}$.*

We shall also use the following version of the above theorem. This states that any location that does not appear in a complete reachability tree (or appears in nodes with empty data region) is not reachable from the precondition Pre.

Theorem 2 *Let C be a CFA with initial location q_0, Pre a precondition, and T a complete reachability tree. Let q be any location of C, and $\mathcal{E}_q$ be the error function that maps q to 1 and q' to 0 for $q \neq q'$. If T is safe with respect to Pre and error function $\mathcal{E}_q$ then the location q is not reachable in C from the precondition Pre.*

4.3. BLAST Verification Algorithm

The model checking algorithm for BLAST takes as input a CFA, an error function $\mathcal{E}$ on the CFA, and a partial abstract reachability tree T and returns either an abstract reachability tree $\hat{T}$ that is safe with respect to Pre and $\mathcal{E}$ (the system is safe); or a concrete path from the precondition to the error state (the system is unsafe). The BLAST algorithm implements an

abstract–model check–refine loop, where abstraction is done lazily [13].

The lazyModelChecker algorithm tries to construct an abstract reachability tree for a CFA which is safe with respect to a precondition and an error function. More precisely, lazyModelChecker takes as input a CFA C with initial location q_0, a precondition Pre and an error function $\mathcal{E}$ for C, and a partial reachability tree (T, F) for C which is safe with respect to Pre and $\mathcal{E}$. If the algorithm terminates, it returns either the pair ("safe", T'), where T' is an abstract reachability tree for C which is safe with respect to Pre and $\mathcal{E}$, or the pair ("unsafe", σ), where σ is an error path of C, that is, a path from (q_0, Pre) to some node q with $\mathcal{E}(q) = 1$. In the former case, C is safe with respect to Pre and $\mathcal{E}$. In the latter case, the model checker provides an actual error trace.

We give a brief outline of the algorithm; for details, see [13]. The abstract reachability tree is built in two phases: forward reachability and backwards counterexample driven refinement. At each point, the algorithm maintains a finite list of *abstraction predicates*. In the forward reachability phase, the algorithm searches the program state space to construct an abstract reachability tree. Each path in the tree corresponds to a path in the CFA. Each node of the tree contains 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 no error node is reachable in the tree, then the algorithm stops and returns safe. If we find that an error node is reachable in the tree, then we proceed to the second phase, which checks if the error is real or results from the abstraction being too coarse (i.e., if we lost too much information by restricting ourselves to a particular set of abstraction predicates). In the latter case, BLAST asks a theorem prover to suggest additional abstraction predicates, which rule out that particular spurious counterexample. By iterating the two phases of forward search and backward counterexample analysis, different portions of the abstract reachability tree will use different sets of abstraction predicates.

5. Generating Tests With Blast

5.1. Testing Algorithm

The input to the testing algorithm is a C program P, a specification automaton A, and a subset R of states of A. The testing algorithm outputs a partition of all locations of the control flow automaton consisting of (1) all locations ℓ such that there exists an execution path such that the CFA is in the location ℓ and the specification automaton is simultaneously in a state in R; and (2) all other locations. Moreover, for each location ℓ for which there is some execution, the algorithm generates a trace through the program that witnesses one such execution, as well as a test case of inputs that causes the program to follow this trace.

For example, consider the `setuid` specification from Figure 5. The set of automaton states for which $\_\_E\_\_ = 0$ are exactly those where the effective user id is 0 (i.e., the program has root privileges). Given a C program P that makes setuid system calls, the setuid automaton, and the set of states $\_\_E\_\_ = 0$, the testing algorithm outputs all locations of P that may execute with root privileges. As a special case, suppose that the automaton A is a trivial specification (*true*), and the set of states R is *true*. Then the testing algorithm simply outputs all locations that are reachable, and all locations that are (provably) not reachable along any execution.

At a high level, the testing algorithm essentially runs the model checking algorithm for each location. We build the control flow automaton of the program, and annotate the call graph with the specification automaton. The locations of the CFA are numbered in depth first order. The algorithm then generates tests (or proves unreachability) by model checking each location starting from the maximum number and going down. Model checking is performed only for locations that have not been covered by previous tests, or proved unreachable by a previously generated complete reachability tree. In addition, the model checking algorithm uses heuristics to choose the next location to expand. The nodes are sorted according to priorities in the worklist. We implement a two-level priority scheme where covered locations have low priority, uncovered locations have high priority. The model checker only explores a location of low priority if there are no locations of high priority to be explored. In this way the search strategy of the model checker searches locations that have not already been covered in previous tests first. Each model checking of a location is not started from scratch. Instead, the abstract reachability tree returned by BLAST in the previous phase is used to start the next model checking. Moreover, the user has the option to fix a timeout for the model checking. If model checking has not succeeded in generating a test or a complete reachability tree showing unreachability by the timeout, the algorithm fails on the current location and moves to the next location. We have found these engineering optimizations to be essential for the algorithm to work on large programs.

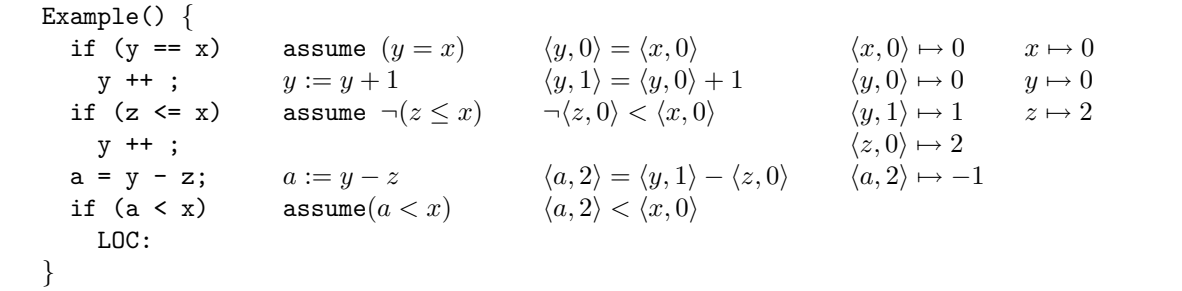

Figure 6. Generating a test case

t	$Con.(\theta, \varphi).t$
$(x := e : pc_i)$	$(\theta', \varphi \wedge (Sub.\theta'.x = Sub.\theta.e))$ **where** $\theta' = Upd.\theta.\{x\}$
$(\texttt{assume}(p) : pc_i)$	$(\theta, \varphi \wedge Sub.\theta.p)$
$t_1; t_2$	$Con.(Con.(\theta, \varphi).t_1).t_2$

Figure 7. Building the TF with Con

5.2. Generating Tests from Traces

When model checking fails, the algorithm produces a trace to the error state. This trace is symbolically represented as constraints on the program variables [16, 6, 11]. The symbolic evaluator in BLAST handles arithmetic operators as well as aliasing relationships between program variables. The method used to decide whether the trace was genuine or not is to check whether the system of constraints is satisfiable —and the path is genuine iff the system of constraints is satisfiable. We shall call the constraints arising from a trace the *Trace Formula* (TF) of the trace.

The basic idea is to use a decision procedure not just to check for satisfiability of the TF, but to also produce a satisfying assignment to the variables when the TF is satisfiable. From the satisfying assignment we can build a test input that will drive the program to the particular location.

Instead of going into how the procedure works for all C programs, we shall restrict ourselves to programs without procedure calls or pointers, *i.e.* we assume all the variables are integers.

Constraint Generation. To denote the value of a variable at some point in the trace, we shall use the pair $\langle \cdot, \cdot \rangle$. For example, in the trace of Figure 6, $\langle x, 0\rangle$ is a constant denoting the value of `x` at the beginning of the trace.

An *lvalue map* is a function θ from *Lvals* to $\mathbb{N}$. We use lvalue maps to generate trace formulas; at a point in the trace, if the map is θ, then the the pair $\langle l, \theta(l)\rangle$ is a special constant which equals the value contained in l at that point in the trace. Whenever some Lvalue x is updated, we update the map so that subsequently, a fresh constant is used to denote the value of x. We do this with the operator $Upd : (Lvals \to \mathbb{N}) \to 2^{Lvals} \to (Lvals \to \mathbb{N})$ takes a map θ and a set of Lvalues L and returns a map θ' where $\theta'.l = \theta.l$ if $l \notin L$ and $\theta'.l = i_l$ where i_l is a fresh integer if $l \in L$. The function Sub takes an Lvalue map θ and an lvalue l and returns $\langle l, \theta(l)\rangle$. The function $Sub.\theta$ is extended naturally to expressions and formulas. A *new* lvalue map is one whose range is disjoint from any other's.

The constraints are generated by the function Con which takes a pair of an lvalue map θ and constraint formula φ and a sequence of commands $t \in Cmd^*$ and returns a pair of a new lvalue map and constraint. Let θ_0 map every lval to 0, and for a trace t, let $Con.(\theta_0, true).t = (\theta, \varphi)$. Then the TF of t is φ, and it can be shown by induction on the length of the trace t that the TF is satisfiable iff the trace is feasible.

The function Con is defined in the first three rows of Figure 7. For an assignment operation, we first update the map so that a new constant denotes the value of x, and the constraint for the operation states that the new constant for x has the same value as the expression e (with appropriate constants plugged in). For an assume operation, the constraint generated stipulates that the constraints at that point satisfy the formula p. The third row shows how the function works for a sequence of operations.

In Figure 6(a) is shown a program and in Figure 6(b) and (c) are shown respectively a feasible trace to a point LOC in the program and the trace formula for the trace. The constraint for each operation in the trace is shown to the right of the operation.

Tests from Constraints. Let θ_0 be the lvalue

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

Table 1. Experimental Results.

map that maps all variables to 0. For a trace t, let $Con.(\theta_0, true).t = (\theta, \varphi)$. If t is a feasible trace to a point in the program, then φ has a satisying assignment.

Notice that the constraints are just a conjunction of arithmetic facts. Moreover, in our experience, many programs generate constraints with *linear* arithmetic constraints. Thus, we can check if the constraints are feasible using an ILP solver which would, were the constraints feasible, return a satisfying assignment as well. For a satisfiable formula φ let $S.\varphi$ be a satisfying assignment to all the variables in the formula (of the form $\langle x, i\rangle$). A test input that will exercise the trace t is the vector that sets every input variable x to the $S.\varphi.\langle x, 0\rangle$.

In Figure 6(d) is shown a satisfying assignment for the TF of Figure 6(c). It is easy to check that if we set the inputs as: $x = 0, y = 0, z = 2$ as the algorithm does, then the program follows the trace.

Pointers. The above method can be extended to programs with pointers; we first generate the TF from whose satisfying assignment we can get a test vector in exactly the same way as described above. The details of the TF generation are in [11]. The resulting TF contains disjuncts as well (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 one, extract our test vector from a satisfying assignment of the disjunct. Second, we can use efficient decision procedures for propositional satisfiability [17] 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. In fact, many off-the-shelf decision procedures already incorporate this style of propositional reasoning [23, 8, 11].

6. Applications and Experiments

We ran BLAST to generate tests and prove unreachability 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, `ftpd` is a Linux port of the BSD implementation of the ftp daemon. The experiments were run on a 3.06GHz Dell Precision 650 with 4Gb of memory. We present results for checking reachability of code.

The column CFA denotes number of CFA nodes. The CFA represents programs compactly, each basic block is on a single edge. The number of CFA nodes is the number of nodes reachable in the call graph of the program. To compare, column *LOC* gives the number of source lines of code.

In our experiment, the specification was trivial (i.e., *true*), so that live states corresponded to states that were reachable by some execution, and dead states to code that was not reachable in any execution. Syntactically indicated behaviors (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 [21]. The usual approach to deal with the infeasible traces 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.

Live is the number of reachable nodes, *Dead* is number of unreachable nodes, *Tests* is number of tests generated. Ideally, we should have the total number of nodes (in column CFA) equal to the sum of the live and dead columns. However, in our tool we set a timeout for each location, so in practice, the tool fails on a small percentage of locations. In our experiments, we set the timeout to 10 minutes per node. The implementation does not run the model checker for a node 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 preds is the total number of predicates generated in the model checking process over all nodes, *Average* is the average number of predicates active at one program point. The average number of predicates at any node is much smaller than the total number of predicates, thus confirming our belief that local and precise abstractions can scale to large programs [13]. *Time* gives running time rounded to minutes (except for ftpd where the tool ran for two overnight runs to produce the result).

We found many locations that were not reachable because of correlated branches. 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' ||
  driveLetterName.Buffer[1] != ':') {
    ...
}
```

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 builtin constants. At run time, most of the comparisons fail, giving rise to many dead locations. Further, the Windows device drivers were already annotated with some safety specifications dealing with IRP completion. Hence, the percentage of unreachable nodes in these drivers are high, since incorrect locations of the automaton are not reached. For example, the driver checked in certain situations that the observer state machine was in the proper state, and called an error function otherwise. In all executions, the state machine would be in the proper state, and therefore the error functions would be unreachable.

While we report only on unreachable code, we have also run Blast on several small examples with the security specification (including the example from Section 2). We are currently applying the testing algorithm to security properties involving setuid system calls. One problem is that most security programs make recursive calls, and our previous implementation of Blast did not handle recursive calls. We are currently implementing a new version that handles recursive calls. We are also applying Blast to see locking behavior in operating system kernel code, using the specification from Figure 4. Moreover, while we have only discussed reachability of nodes in a program, we can use the model checking algorithm to generate tests for more complex testing criteria [14, 21].

References

[1] T. Ball and S. Rajamani. The SLAM project: Debugging system software via static analysis. In *POPL 02: Principles of Programming Languages*, pages 1–3. ACM, 2002.

[2] T. Ball and S. K. Rajamani. SLIC: A specification language for interface checking (of C). Technical Report MSR-TR-2001-21, Microsoft Research, 2001.

[3] R. Bodik, R. Gupta, and M. L. Soffa. Interprocedural conditional branch elimination. In *PLDI 97: Programming Language Design and Implementation*, pages 146–158. ACM, 1997.

[4] H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In *ACM CCS 02: Conference on Computer and Communications Security*, pages 235–244. ACM, 2002.

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

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

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

[8] J.-C. Filliâtre, S. Owre, H. Ruess, and N. Shankar. ICS: Integrated canonizer and solver. In *CAV 01: Computer-aided verification*, Lecture Notes in Computer Science, pages ???–??? Springer-Verlag, 2001.

[9] E. Gunter and D. Peled. Temporal debugging for concurrent systems. In *TACAS 02: Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *LNCS*, pages 431–444. Springer, 2002.

[10] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *PLDI 02: Programming Language Design and Implementation*, pages 69–82. ACM, 2002.

[11] T. Henzinger, R. Jhala, R. Majumdar, and K. McMillan. Abstractions from proofs. In *POPL 04: Principles of Programming Languages*, pages ??–?? ACM, 2004.

[12] T. Henzinger, R. Jhala, R. Majumdar, and M. Sanvido. Extreme model checking. In *International Symposium on Verification*, LNCS. Springer, 2003.

[13] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL 02: Principles of Programming Languages*, pages 58–70. ACM, 2002.

[14] H. Hong, S. Cha, I. Lee, O. Sokolsky, and H. Ural. Data flow testing as model checking. In *ICSE 2003: Software Engineering*, pages 232–243. ACM, 2003.

[15] S. Khurshid, C. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *TACAS 03: Tools and Algorithms for the Construction and Analysis of Systems*, LNCS, pages 553–568. Springer, 2003.

[16] J. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[17] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *DAC 01: Design Automation Conference*, pages 530–535, 2001.

[18] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC 02: Compiler Construction*, Lecture Notes in Computer Science 2304, pages 213–228. Springer-Verlag, 2002.

[19] D. Peled. *Software reliability methods*. Springer, 2001.

[20] D. Peled. Model checking and testing combined. In *ICALP 2003: Automata, Languages and Programming*, volume 2719 of *LNCS*, pages 47–63. Springer, 2003.

[21] M. Pezze‘ and M. Young. Software test and analysis: Process, principles, and techniques. Manuscript, 2003.
[22] F. B. Schneider. Enforceable security policies. Technical Report TR98-1664, Cornell University, 1999.
[23] A. Stump, C. Barrett, and D. Dill. CVC: A cooperating validity checker. In *CAV 02: Computer-Aided Verification*, LNCS 2404, pages 500–504. Springer, 2002.