Path Slicing

Ranjit Jhala
CS Department, UC San Diego
jhala@cs.ucsd.edu

Rupak Majumdar
CS Department, UC Los Angeles
rupak@cs.ucla.edu

Abstract
We present a new technique, path slicing, that takes as input a possibly infeasible path to a target location, and eliminates all the operations that are irrelevant towards the reachability of the target location. A path slice is a subsequence of the original path whose infeasibility guarantees the infeasibility of the original path, and whose feasibility guarantees the existence of some feasible variant of the given path that reaches the target location even though the given path may itself be infeasible. Our method combines the ability of program slicing to look at several program paths, with the precision that dynamic slicing enjoys by focusing on a single path. We have implemented Path Slicing to analyze possible counterexamples returned by the software model checker BLAST. We show its effectiveness in drastically reducing the size of the counterexamples to less than 1% of their original size. This enables the precise verification of application programs (upto 100KLOC), by allowing the analysis to focus on the part of the counterexample that is relevant to the property being checked.

Categories and Subject Descriptors: D.2.4 [Software Engineering]: Software/Program Verification; D.3.4 [Programming Languages]: Processors.

General Terms: Languages, Verification, Reliability.

Keywords: Program slicing, counterexample analysis.

1. Introduction
We introduce path slicing, a technique to determine which subset of the edges along a given control flow path to particular target location are relevant towards demonstrating the (un)reachability of the target location along the given path.

Static analyses used to verify safety properties return control flow paths to an error location as possible counterexamples demonstrating that the path is unsafe. As the static analysis is typically conservative, this path may or may not represent an actual violation of the property. Traditionally, such control flow paths, are examined manually to determine whether they correspond to a feasible program execution, and therefore, a bug [9, 8, 14, 23]. For large programs, the sheer length of the path makes such manual inspection difficult. More recently, counterexample-guided program analyses [3, 18, 7, 12] attempt to automatically find out if the path is feasible, and if not, they exploit the infeasibility of the path to refine the abstraction used for analysis. Once again, long counterexamples complicate this process.

Consider the program Ex2 shown in Figure 1(A). For the moment, let us assume that the shaded grey code is missing, i.e., the program begins from the label 2.: This program is a simplified version of the following verification instance: The variable x corresponds to a file; it is non-zero if the file has been opened. The file may be opened and read depending on some condition, captured by the variable a, that is unconstrained in the example (suppose it is an input). The file should only be read if open; the branch at 6: models the check that the file is indeed open, which is an assert that we put in just before an actual read from the file takes place. Thus, the label ERR is reached if this assertion is violated. Let us suppose that the function f, whose code is not shown, always terminates, and that f does not modify the variables a, x. In this case, as the program eventually breaks out of the loop, a is unconstrained, and x is not set anywhere (recall that we are assuming the shaded lines are missing), the target is indeed reachable. However, any feasible path to the target must contain a thousand unrollings of the “for-loop”, not to mention feasible paths through f. Any path to the target location that does not meet these requirements is infeasible, and if our static analysis returns such a path, we have no option but to dismiss it as a false positive arising from the imprecision of the analysis. In this case, the analysis cannot be used to make any claim about the reachability of the target location. Counterexample-guided refinement based techniques are doomed to either take a long time to find an abstraction precise enough to force them around the loop a thousand times, or worse, not terminate, because of the difficulty of finding a feasible path through f, which may be arbitrarily complex.

The question then is, is it possible to use (overapproximate) static analyses to precisely report that the target location is reach-
Path slicing is distinct from dynamic slicing [20, 27] in two ways. First, the paths are not the result of a dynamic execution and hence are not guaranteed to be feasible. Second, and more importantly, we consider alternative program paths to find if some variant of the given path is in fact feasible, as shown by the path through Ex2.

We show how to efficiently compute a path slice, by performing a backwards dataflow analysis over the given path. Our algorithm, called PathSlice, iterates backwards over the path, tracking, at each point, the set of lvalues that determine if the suffix of the path from that point is feasible, called the live lvalues and the source location of the last edge added to the slice, called the step location. We take an edge corresponding to an assignment if the assignment is to a live variable. We take an edge corresponding to a conditional if either the conditional corresponds to the branch direction that must be taken to reach the step location, or if in the branch not taken, the program may have modified a live variable. If an edge is taken, the live set and step location are appropriately updated. We show how a must-reachability analysis can be used to detect the former case, and a modified-variable analysis can be used to detect the latter case. Both these analyses are intraprocedural and hence give us an efficient path slicing algorithm. For function calls, we only enter the body of the call if the function can modify a live variable, otherwise the entire path through the call is sliced away.

One limitation of path slicing is that it avoids the difficult question of statically reasoning about termination. As a result, the feasibility of a path slice guarantees that either the target location is reachable, or all states that can execute the path slice, cause the program to enter an infinite loop.

We have implemented Algorithm PathSlice in the counterexample analysis phase of the software model checker BLAST [18]. We ran this enhanced algorithm to check for file handling errors in a set of application programs. The largest programs checked had about 100K lines of code. Without the path slicing algorithm, the counterexample analysis phase of BLAST did not scale to any of these examples, because first the counterexamples were too large to analyze for feasibility, and second they contained many irrelevant reasons for infeasibility, causing a blowup in the size of the abstractions as well as the number of iterations taken to find the abstraction relevant to the property.

With the path slicing algorithm we were able to check file handling errors in all the programs, and found violations of the property in some of them. In general, we found that slicing reduced counterexample traces to less than 1% of their original size in most cases. In fact, as the size of counterexample traces got bigger, the slicing was more effective (traces over 5000 basic blocks almost always produced slices between 0.1% and 1% of their original sizes. The end-to-end verification times (including model checking) for these examples were all below one hour. In the cases where tool returns a feasible path slice it is much easier for the user to go over the more succinct slice to ascertain the veracity of the counterexample. Thus, our experiments suggest that path slicing can extend the scope of static analysis by eliminating irrelevant details.

Related Work. Program slicing [25] has been developed as a useful tool program debugging, comprehension and testing [24]. Slicing is studied primarily in two forms: static and dynamic. In static slicing [19], the input is the static text of the program and a slicing criterion (for example, a set of variables at a particular program point), and a static analysis algorithm is used to overapproximate the set of all variables that may affect the slicing criterion [24]. Unfortunately, while the sliced program generated by these algorithms contains a subset of the statements and control
predicates of the original algorithm, it is not guaranteed to be an executable program. Moreover, static analysis algorithms for slicing, especially in the presence of memory aliasing, are usually overly conservative, and manage to retain a large percentage of the original program [22], as we found in experiments using a state-of-the-art slicing tool [15].

The highly conservative nature of static slicing promoted the study of dynamic slicing [20, 27], where the program is executed on a particular test input, and the resulting execution trace (for that single input) is sliced. While this may not produce a slice for the entire program, it can be more precise on the particular execution, and provide a smaller slice for applications in debugging.

As we have discussed in the prequel, path slicing is different from both of the above—it combines the precision of dynamic slicing with the static slicing’s ability to reason about multiple paths. Parametric slicing [11] generalizes static and dynamic slicing by constraining a subset of the inputs to a program. Path slicing is different because the control flow path may not be feasible, moreover, the objective of the analysis is to construct suitable constraints on the inputs that makes some variant of the path feasible.

There has been some work in the model checking community to present to the user the “cause” of an error [2, 16, 4]. Typically these algorithms assume that the path is feasible, and use differencing algorithms to identify commonalities between multiple paths to error. The algorithm of [4] uses a dependence analysis similar to path slicing for this purpose. In the past, automatic refinement based techniques [3, 18] were used to analyze safety properties of low level device drivers. In our experience, the counterexamples for such checks are typically two orders of magnitude smaller than counterexamples arising from application level programs, such as the ones we consider here, and hence could be directly analyzed [17]. It should be noted that path slicing is orthogonal to parsimonious abstractions [17] (finding local predicates that show the infeasibility of a trace). Path slicing finds which operations along a trace can possibly influence reachability of the error location, while the refinement algorithm of [17] analyzes the output of the path slicer to find why a path is infeasible.

2. Motivating Examples

We begin by illustrating path slices and how our algorithm computes them using some small examples. Recall the program Ex2 shown in Figure 1(A). The label ERR: corresponds to a target location, and we are interested in whether the the program can ever reach the target location, as this is equivalent to there being some program execution where the assertion being checked is violated.

Control Flow Automata. We model programs as control flow automata (CFA), which are essentially the CFG of each function, with the operations labeling the edges instead of the vertices. A CFA consists of: (1) integer variables, (2) control locations, including the special locations start and exit, and (3) directed edges that connect the locations. Control starts at the start location and ends at the exit location. Each edge is labeled by either an assignment that is executed when the program moves along the edge, or by an assume predicate which must be true for control to move along the edge, or a function call which corresponds to control jumping to the start location of the called procedure. When the called procedure reaches its exit location, control transfers back to the successor of the call-edge in the caller. A program is a set of CFA, one for each function.

Example 1: [CFA] The CFA for Ex2 is shown on the right in Figure 1(B). We omit the CFA for the function f for brevity. As with the program, let us assume for the moment that the shaded vertices are not present, and the CFA start node is 2.

Paths. A (program) path is a sequence of CFA edges, such that the calls and exits from functions are balanced, and within each CFA, the source of each edge in the sequence is the target of the previous edge of the sequence. A path corresponds to a sequence of operations, namely those labeling the edges. A path is feasible if there is some input on which the program executes the corresponding sequence of operations.

Example 2: [Path] Figure 1(C) shows a path from the start location of the CFA to the target location, again ignoring the shaded vertices. The path segment through e is omitted in the figure for clarity. This path is infeasible. We set $i$ to 1 at the start of the loop (on edge $3' ightarrow 3'$), then go through the loop once, incrementing $i$ once (on edge $4 ightarrow 3'$) and then break out of the loop assuming that $i$ exceeds 1000 (edge $3' ightarrow 5$) even though $i$ is really 2.

Path Slices. A slice of a path $\pi$ is a subsequence of the edges of the $\pi$ such that

1. (complete) whenever the sequence of operations labeling the subsequence is feasible, the target location is reachable, and
2. (sound) whenever the sequence of operations labeling the subsequence is infeasible, the path is infeasible.

Intuitively, a path slice is obtained by dropping some edges along the path, but leaving the edges corresponding to branches that must be taken to reach the target, and assignments that feed the values to the expressions in the branches.

Example 3: [Completeness: Feasible Path Slice] In Figure 1(C), the set of solid edges (ignoring the shaded edge) is a slice of the entire path shown. Notice that the sequence of operations labeling the solid edges is feasible, and hence, by completeness, demonstrates that the target location is reachable. In particular, the inputs that can execute the sequence of solid edges are those that satisfy the formula $a \geq 0$, and upon starting at any state satisfying this formula, the program can reach the target location.

Example 4: [Soundness: Infeasible Path Slice] Let us now suppose that the program Ex2 in Figure 1(A) contains the shaded instructions as well. In this case, the target location is not reachable, as the programmer has ensured that if $a \geq 0$, then $x$ is set to 1. The corresponding CFA is that shown in Figure 1(B) with the shaded vertices included, and a possible path to the target is shown in Figure 1(C), once again including the shaded vertices. The path through the loop remains infeasible, but as it is irrelevant, the edges corresponding to the loop are omitted from the path slice. Note that the two (inconsistent) branches that pertain to the reachability of the target remain in the slice. Thus any automatic refinement scheme would be able to focus on the real reason for infeasibility, and hence the real reason for the unreachability of the target, without gathering irrelevant facts about the loop iterations.

Computing Path Slices. Our algorithm iterates backwards over the path, tracking at each point (1) the live lvalues: the set of lvalues that determine if the suffix of the path from that point is feasible, and (2) the step location: the source location of the last edge added to the slice.

An assignment $l := e$ gets added to the slice if $l$ is an lvalue in the live set; in this case the lvalue is removed from the live set, and the lvalues of $e$ are added to the set. The more interesting case is for branches. The trace represents only one of the possible choices that the program may have made at a branch point, and an assume must be added to the slice if along any of these directions something may happen that affects the slice suffix. In particular, an assume

---

1 modulo termination, see Section 3 for details.
branch is important if either along one of the other branch edges the program control veers off and does not return to the slice suffix, or if along one of the other branch edges the program writes an lvalue that is live, i.e., read along the slice suffix. The first case is important because only one direction leads to the error along the slice suffix, and the second case because the branch controls whether or not a live variable gets updated.

To determine if the first case occurs, we check if there exists a path from the source of the branch that can “bypass” the step location altogether, if so then only the assume operation corresponds to the branch direction that leads to the step location and hence must be added. To determine if the second case occurs, we check if there exists a path between the source of the branch and the step location along which a live variable is written.

This analysis is different from dynamic slicing algorithms where we know that the trace under consideration is feasible. In our algorithm, although the given path may be infeasible, a variant of it, obtained by dropping irrelevant edges, may be feasible. In path slicing, we keep only those edges, in particular those branches, that must be taken along this path to reach the target location.

**Example 5: [Computing a Path Slice]** We now illustrate our algorithm by showing how the path slice shown in Figure 1 is computed. In Figure 1(C), the value of the step location and the live set is shown at each point along the path. We start at the last location, the target location. At this point, the step location and the live set are (trivially) ERR and the empty set respectively. Next, we process the location 6. Notice that 6 can bypass the current step location ERR, if the operation corresponding to the “else” branch is taken. Hence, this assume is relevant, and we add the edge 6 → ERR to the slice. The step location and live set at 6 are updated to 6 and the set {a}, as a appears in the branch condition. Next, we process the location 5. Notice that 5 can also bypass the current step location 6, again by following the “else” branch. Thus, this assume edge is also added to the slice, and the step location and live set at 5 are updated to 5 and {x, a} as a appears in the branch condition. Next, we process (the second instance) of location 3'. This time, we find that 3' cannot bypass the step location 5 — unless either the loop or 1 does not terminate, it is inevitable that location 5 will be reached. Also, on no path from 3' to 5, are any of the currently live variables {x, a} modified. Hence, the assume edge 3' → 5 is irrelevant and not added to the slice (is dotted in the figure), and the step location and live set remain unchanged. Next, we process the edge 4 → 3'. As none of the live variables are modified by these assignments, we discard the edge from the slice. Then we process the assume edge 3' → 4. Once again, as 3' cannot bypass the current step location 5, and no path from 3' to 5 modifies a live variable, we omit the edge. The assignments along the edges 3 → 3' and 2 → 3 are not to live variables and so these edges are omitted from the slice. This is the slice of the program starting at node 2. If the shaded code is included, we come to the last assume edge 0 → 2. Notice that 0 cannot bypass the current step location 5 (node 5 postdominates node 0). However, there exists a path from 0 to 5 along which the live variable x is modified, and this path is along the branch not taken. Hence, in order that the given path correspond to a path to the target, this particular branch must be taken. As a result, this edge is added to the slice, rendering the path slice infeasible.

**Example 6: [Path Slicing vs. Program Slicing]** In both the previous examples, a program slice would have sufficed to eliminate the for loop. However, consider example Ex1 in Figure 2(A). In Figure 2(B), the solid edges denote the path slice for the path corresponding to all the edges (the subpath through complex is omitted for clarity). The annotations on the right are the values of the step location and live set as we iterate backwards over the path. The reader can use them to verify that the slice shown is indeed the one that the above algorithm would compute. Hence, by focusing the slice on a particular path, we can eliminate the procedure complex entirely. Thus, without actually finding a complete feasible path, we can show that if complex terminates, then upon starting from any state where a ≤ 0, the program can reach the target location.

3. **Programs, Paths, and Slices**

We illustrate our algorithm on a small imperative language with integer variables, references, and functions with call-by-value parameter passing. We begin by completely developing our technique for programs without procedure calls or references. After this, we add pointers. In the next section, we show how the techniques are generalized to programs with procedure calls.

3.1 **Syntax and Semantics**

We first consider a language with integer valued variables, and no procedure calls. Boolean expressions arise via boolean combinations of arithmetic comparisons.

**Operations.** Our programs are built using two kinds of basic operations:

1. An assignment operation is of the form \( l := e \); which corresponds assigning the value of the expression \( e \) to the variable \( l \).
2. An assume operation is of the form \( \text{assume}(p) \); if the boolean expression \( p \) evaluates to true, then the program continues, and otherwise the program halts. Assumptions are used to model branch conditions.

The set of operations is denoted \( \text{Ops} \).

**Control Flow Automata.** Each function \( f \) is represented as a control flow automaton \( \text{CFA}\ f = (PC_\ f, pc_\ 0, pc_\ \text{out}, E_\ f, V_\ f) \).

The CFA \( \text{CFA}\ f \) is a rooted, directed graph with:

1. A set of control locations (or program counters) \( PC_\ f \) which include a special start location \( pc_\ 0 \in PC_\ f \), and a special exit location \( pc_\ \text{out} \in PC_\ f \).
2. A set of edges \( E_\ f \subseteq PC_\ f \times \text{Ops} \times PC_\ f \). We write \( (pc, op, pc') \) to denote the edge from \( pc \) to \( pc' \) labeled \( op \).

A CFA is the control flow graph of a program; its locations correspond to program locations, and its edges correspond to the commands that take the program from one location to the next. A program comprises a single CFA \( \text{CFA}\ f \) corresponding to a procedure \( f \).

**States, Transitions.** For a set of variables \( X \), an \( X \)-state is a valuation for the variables \( X \). The set of all \( X \)-states is written as \( \text{Val}_X \). Each operation \( op \) gives rise to a transition relation \( \omega_{op} \subseteq \text{Val}_X \times \text{Val}_X \) as follows. We say that \( s_{op} s' \) if:

\[
\begin{align*}
  s' &= \begin{cases} 
  s & \text{if op } \equiv \text{assume}(p) \text{ and } s \models p \\
  s[1 \mapsto s.e] & \text{if op } \equiv 1 := e 
  \end{cases}
\end{align*}
\]

We say that a state \( s \) can execute the operation \( op \) if there exists some \( s' \) such that \( s_{op} s' \).

**Weakest Preconditions.** An alternative way to view the semantics of programs is via logical formulas. A formula \( \varphi \) over the variables \( X \) represents all \( X \)-states where the values of the variables satisfy \( \varphi \). The weakest precondition of \( \varphi \) w.r.t. an operation \( op \), written \( \text{WP}_\varphi.\varphi.\varphi \) is the set of states that can reach a state in \( \varphi \) after executing \( op \). Formally: \( \text{WP}_\varphi.\varphi.\varphi \equiv \{ s \mid \exists s' \in \varphi, s_{op} s' \} \). The weakest precondition operator for our language can be computed syntactically as a predicate transformer [10], as shown in the second column of Figure 3.
Traces. A trace \( \tau \) is a sequence of operations. We say that a state \( s \) can execute the trace \( \tau \) if either \( \tau \) is the empty sequence \( \epsilon \), or \( \tau \equiv op \cdot \tau' \) and there exists \( s' \) such that \( s \equiv op \cdot s' \) and \( s' \) can execute the sequence \( \tau' \). We say that a trace \( \tau \) is feasible if there exists a state \( s \) that can execute \( \tau \), and we say the trace is infeasible otherwise. The weakest precondition condition extends easily to traces as:

\[
WP(\varphi, \tau) \equiv \begin{cases} 
\varphi & \text{if } \tau \equiv \epsilon \\
WP((WP(\varphi, op), \tau')) & \text{if } \tau \equiv op \cdot \tau';
\end{cases}
\]

It is easy to check that a trace \( \tau \) is feasible iff \( WP(true, \tau) \) is satisfiable [10].

Program Paths. A CFA edge \((pc', \ldots, pc)\) is a successor of another CFA edge \((\ldots, pc)\), if \( pc = pc' \). A program path, or path in brief, is a sequence of CFA edges \( \pi = \pi_1; \ldots; \pi_n \), such that for each \( 2 \leq i \leq n \), the edge \( \pi_i \) is a successor of the edge \( \pi_i-1 \). We denote by \( |\pi| \) the number of edges on the path \( \pi \). We say that \( \pi \) is a path from \( pc \) to \( pc' \) if \( \pi.1 \equiv \{pc, \ldots, pc\} \) and \( \pi.|\pi| \equiv \{\ldots, pc'\} \).

The trace \( Tr.\pi \) of a path \( \pi \) is the sequence of operations labeling the edges along the path. A state \( s \) can execute the trace \( \pi \) if it can execute the trace \( Tr.\pi \). A path \( \pi \) is feasible if there is some state that can execute \( \pi \), and infeasible otherwise.

Reachability. A state \( s \) can reach a location \( pc \) if there exists a path \( \pi \) from \( pc_0 \) to \( pc \) such that \( s \) can execute \( \pi \), otherwise it cannot reach the location. A location \( pc \) is reachable if there exists some state that can reach it.

3.2 Path Slices

Slices. Let \( \pi \) be a path. Then any subsequence \( \pi' \equiv \pi_{i1}; \ldots; \pi_{ik} \) for \( 1 \leq i_1 < \ldots < i_k \leq |\pi| \), is a path slice of \( \pi \), written \( \pi' \preceq \pi \).

In the sequel assume that \( \pi \) is a program path from \( pc_0 \) to a (special error location) \( pc\).

Soundness. We say that a path slice \( \pi' \) of \( \pi \) is a sound slice if \( WP(true, (Tr.\pi)) \subseteq WP(true, (Tr.\pi')) \). In other words, every state that can execute the trace \( Tr.\pi \) can also execute the trace \( Tr.\pi' \).

In particular, if \( \pi' \) is a sound slice of \( \pi \) and \( WP(true, (Tr.\pi')) \) is not satisfiable, then \( WP.true((Tr.\pi)) \) is not satisfiable, i.e., \( \pi \) is infeasible. We can think of \( \pi' \) as overapproximating \( \pi \). While a sound slice may be very coarse — the empty slice is sound — not all slices are sound. For example, the result of dropping the second edge in the path \((pc_0, assume(1 = 0), pc_1); (pc_1, 1 := 1, pc_2); (pc_2, assume(1 = 1), pc_3)\) is an unsound slice.

Completeness. We say that \( \pi' \) is a complete slice of \( \pi \) if for every \( s \in WP.true(Tr.\pi') \) either:

1. there exists a program path \( \pi'' \) from \( pc_0 \) to \( pc \) such that \( s \) can execute \( \pi'' \), or,
2. \( s \) cannot reach \( pc \).

Hence, if \( \pi' \) is a complete slice of a path to a particular (error) location, and \( Tr.\pi' \) is feasible, then every state that can execute \( Tr.\pi' \) either can reach the error location, or causes the program to loop forever.

Example 7: We can check that the path slice shown in Figure 1(C) is sound and complete. Let us consider the program Ex2, without the shaded parts. In this case we find that the slice computed is sound because the WP over the operations in the slice is \( \varphi \equiv x = 0 \land a > 0 \), while the weakest precondition over the entire trace is \( \varphi \) conjointed with constraints arising from the path through \( \ell \). The slice is complete because every starting state satisfying \( \varphi \), either gets stuck inside \( \ell \) during some iteration of the loop, and hence never reaches the exit, or, after spinning around the loop a thousand times, takes both the "if" branches and reaches the target location.

3.3 Computing Path Slices

We now describe our algorithm PathSlice that takes a program path and computes a sound and complete slice from it. Informally, PathSlice performs a dataflow analysis on the program path. We iterate backwards, tracking at each point along the path:

**Live set.** A set of relevant *live* lvalues whose valuations at that point determine whether or not the error location is reachable along the suffix of the trace, and.

**Step location.** The source location of the last edge along the path that was added to the slice.

We analyze each operation along the path in turn, and use the live set and the step location to decide whether or not to add the current operation to the slice (Procedure Take). At each point, we call the edges that have already been added to the slice the *slice suffix*. The live set and step location encode the outstanding data and control dependencies at each point along the trace.

We now give a formal description of the algorithm. We first describe a few sets and relations that are used to determine whether a given operation should be in the slice, then describe how these are combined inside Procedure Take, and finally, how the latter is used to obtain a sound and complete path slice in the Algorithm PathSlice.

**Reads, Writes.** We denote by \( Lvs.e \) the set of lvalues occurring in an expression \( e \). This is extended to predicates in the natural way. We say \( Rd.op.1 \) to denote the fact that lvalue \( 1 \) is read by the operation \( op \). We write \( Rd.op.2 \) to denote the set \( \{1 \mid Rd.op.1 \} \), i.e., the set of all lvalues read by \( op \). We say \( Wt.op.1 \) to denote the fact that \( 1 \) is written (assigned to) by the operation \( op \). We write \( Wt.op.2 \) to denote the set \( \{1 \mid Wt.op.1 \} \). A formal description of \( Rd \) and \( Wt \) is in the third and fourth column of Figure 3.

**Bypassing Locations.** We say that a location \( pc \) can bypass location \( pc' \), if there exists a path from \( pc \) to the exit location \( pc_{\text{out}} \) for the function that does not visit \( pc' \). The set of locations that can bypass \( pc' \) is written as \( By.pc' \). This is the set of all locations that \( pc' \) does not postdominate.

**Written Between.** We say that an lvalue \( 1 \) is written between locations \( (pc, pc') \), denoted by \( WrBt.(pc, pc').1 \), if there exists a path \( \pi \) from \( pc \) to \( pc' \) such that some operation along the path is an assignment to \( 1 \), i.e., there exists some \( i \) such that \( \pi.i \equiv (, op,) \) and \( Wt.op.1 \). We generalize this to sets of lvalues \( L \) as \( WrBt.(pc, pc').L \equiv \exists l \in L : WrBt.(pc, pc').l \).

**Procedure Take.** Armed with the above relations, we define the procedure Take which takes as input (1) the set of live lvalues \( L \), (2) the step location \( pc_s \), and (3) an edge \( (pc, op, pc') \), and returns a boolean indicating whether this edge should be added to the slice. The formal definition of Take \((L, pc_s, (pc, op, pc')) \) is given in the fifth column of Figure 3. Assignments get taken if the lvalue written to is in the live set Live. Assumes, corresponding to branches, are somewhat trickier. An assume gets taken if there is a path from the current edge that can "bypass" the step location, or if there exists a path from the current edge to the step edge along which a live variable gets modified. If the step location can be bypassed, then it means that if the program had taken the other branch direction, then it may have bypassed the step location and hence not executed along the slice suffix. If there is a path between the source location and the step location along which a live lvalue gets written, it means that if the program had taken the other branch direction then it may have written that live lvalue (which it didn’t along the path being sliced, as the lvalue is currently live), and thus not been able to execute the slice suffix.
and time linear in the size of slice, call this the slice suffix algorithm maintains the following inductive invariant. At any step complete path slice for the path suffix at that point. Precisely, the

The algorithm \( \pi \) complete path slice of \( \pi \) at the end of the path still satisfy \( \phi \) and this valuation to the lvalues all the way to the final location.

### 3.4 Pointers

We now describe how the above generalizes to the setting where in addition to integer variables, the programs contain dereferences. Lvalues now generalize to memory locations, and correspond to either declared variables or dereferences of pointer-typed expressions. Boolean expressions are extended to contain checks of pointer equality.

The only change that must be made in the algorithm described above, is in the definition of \( Wt \), which must be generalized to deal with updates due to aliasing. Suppose that we have precomputed the aliasing relations \( \text{MayAlias} \) and \( \text{MustAlias} \) on the values of the program. We say \( \text{MayAlias}(1,1)' \) (respectively, \( \text{MustAlias}(1,1)' \)) if \( 1 \) may (respectively, must) be aliased to \( 1' \). We require that the computed \( \text{MayAlias} \) be an over-approximation of the actual points to relation, and that the computed \( \text{MustAlias} \) be an under-approximation of the actual points to relation.

The definition of \( Wt.op.' \) (used to define \( \text{WrBt} \)) is generalized to \( Wt.op.' \) if \( op = 1 \) aliased and \( \text{MustAlias}(1,1)' \). The definition of \( Wt.op \), used to update the \( Live \) set in line 10 of Algorithm PathSlice, changes to the set \( \{1' \mid \text{MustAlias}(1,1)' \} \) if \( op = 1 \) aliased, and remains the empty set otherwise. The algorithm PathSlice is identical to the one previously described, once the generalized versions of \( Wt \) are used. With these generalizations, Theorem 1 holds for programs with pointers.

### 4. Programs with Procedures

We now describe how to generalize path slicing to programs with procedure calls. First, we generalize the definition of the syntax and semantics of programs, and describe what program paths look like in this setting. For clarity of exposition, we make the following assumptions: (1) all variables are integer valued, \( i.e. \), there are no pointers, (2) parameters and return values are passed to and from procedures using global variables \( i.e. \), there are no formal parameters or return values), (3) local variables for different procedures have disjoint names, and there is no recursion. Our method generalizes to and our implementation deals with programs without any of the above restrictions.

#### Syntax

We generalize the definition of programs from Section 3 to contain procedures and calls. A program contains a set of global variables \( V \), and each procedure \( f \) contains in addition a set of local variables \( V_f \). Parameters are passed to procedures via global variables \( i.e. \), parameters are written into some globals, and the called procedure copies the values from the globals into its own local variables). Similarly, procedures return values via global variables.

We extend the set of operations labeling CFA edges to include call operations of the form \( \ell() \) and return operations \( \text{return} \). We assume that in any CFA \( C_f = \{ PC_f, pc_0, pc_{out}, E_f, V_f \} \), every edge \( (pc, pc') \) is such that \( pc' = pc_{out} \) \( i.e. \), all "return" statements lead to the exit location of the CFA. A program is a set of CFAs \( P = \{C_{f_0}, \ldots, C_{f_k}\} \), where each \( C_{f_i} \) is the CFA for
a function $f_i$. There is a special function $\text{main}$ with CFA $C_{\text{main}}$ corresponding to the procedure where execution begins.

**Semantics.** With our various assumptions about the program, the definitions of states and transitions remain essentially unchanged. The relation $\rightarrow$ for the new operations call and return is simply the identity relation. Similarly, the weakest precondition for a call or return statement is the identity map.

**Program Paths.** The only change brought about by the addition of CFA edges is in the definition of program paths, which must be extended to reason about calls and returns. Let $\pi$ be a sequence of CFA edges $\pi.1, \ldots, \pi.n$. Define $\text{Call}.1 = 1$ and for each $2 \leq i \leq n$ define $\text{Call}.i$ as:

$$\text{Call}.i = \begin{cases} 
  i - 1 & \text{if } \pi.(i - 1) = (., f \{ \cdot \}, \cdot) \\
  \text{Call}.(\text{Call}.(i - 1)) & \text{if } \pi.(i - 1) = (., \text{return}, \cdot) \\
  \text{Call}.(i - 1) & \text{o.w.}
\end{cases}$$

We say that $\pi$ is a program path if for each $2 \leq i \leq n$: (1) if $\pi.(i - 1) = (., f \{ \cdot \}, \cdot)$ then $\pi.i = (C.f.p_c, \cdot, \cdot)$, (2) if $\pi.(i - 1) = (., \text{return}, \cdot)$ then $\pi.i$ is a successor of $\pi.(i - 1)$, and, (3) $\pi.i$ is a successor of $\pi.(i - 1)$ otherwise. Intuitively, $\text{Call}.i$ is the operation that "begins" the call frame to which the $i$th operation belongs. It is easy to see that for any program path $\pi$, for all $i > 1$, the edge $\text{Call}.i$ is of the form $(., f \{ \cdot \}, \cdot)$ for some $f$. As before, the trace $\text{Tr}.\pi$ is the sequence of operations on the path $\pi$. A path $\pi$ is feasible if the trace $\text{Tr}.\pi$ is feasible, and we say the path is infeasible otherwise.

**Path Slicing.** We now describe how to generalize the algorithm from the previous section to work in the presence of procedure calls. To do so, we will require another (precomputable) relation $\text{Mods}$. Intuitively, for a function $f$ and lvalue $l$, we say $\text{Mods}.f.l$ if the lvalue $l$ can be modified directly inside $f$, or within any function that may (transitively) be called by $f$. Formally

$$\text{Mods}.f.L \equiv \text{WrBt}.(C.f.p_c, 0, C.f.p_{out}).1$$

where the definition of $\text{WrBt}$ is the same as before. We extend this to sets of lvalues $L$ as:

$$\text{Mods}.f.L \equiv \exists l \in L : \text{Mods}.f.l$$

We write by $\text{Mods}.f$ the set $\{ \text{Mods}.f.l \}$. Notice that this set can be computed by using a standard mod-ref analysis [1].

Next, we generalize $\text{Take}$ to deal with calls and returns. Given a return statement, we should only take it and analyze the path between the call corresponding to the return, if a variable in the live set can be modified by the call. A call statement is always taken, as we shall see below, this is done so as to keep the queries to $\text{WrBt}$. By intraprocedural. The formal definition of $\text{Take}$ for calls and returns is shown in the third and fourth rows of Figure 3.

Finally, to generalize PathSlice to handle calls and returns, we replace line 12 (in Figure 1) ($i := i - 1$) with:

$$i := \text{if } \neg l.d_{op} = \text{return} \text{ then } \text{Call}.i - 1 \text{ else } i - 1$$

That is, if we find that the return was not taken, meaning the function being returned from is not relevant, then instead of processing the path through the function, we set $i$ to the first edge before the call. Notice that this implies that whenever a call operation is processed, the corresponding return must have been taken (or there was no corresponding return). With these generalizations, Theorem 1 holds for programs with procedures as well.

**4.1 Intraprocedural Relations**

As we always "take" call operations, the algorithm PathSlice described above has the property that whenever a query is made to $\text{WrBt}$ or $\text{By}$, the pair of program locations used in the query belong to the same CFA. Hence, efficient intraprocedural algorithms can be used to compute those relations. We next describe how to obtain these relations via a fixpoint computation which can either be done explicitly or using symbolic techniques based on BDDs [26, 5]. We assume that $\text{Mods}$ has been computed by a standard mod-ref analysis, and, hence, we can compute $Wt$ for every operation (Figure 3).

**Computing WrBt.** Recall that $\text{WrBt}.(pc, \cdot)$.1 if 1 is written between $pc$ and $pe'$, equivalently, if 1 is written on some edge that is both reachable from $pc$ and can reach $pe'$. Thus, we perform the following intraprocedural fixpoint computation. Let $\text{In}$ and $\text{Out}$ be the least fixed points of the equations below:

$$\text{In}.pc = \bigcup_{e \in (pc, \cdot)} \text{In}.pc'$$

and

$$\text{Out}.pc = \bigcup_{e \in (pc, \cdot)} \text{Out}.pc'.$$

The set $\text{In}.pc$ (respectively, $\text{Out}.pc$) contains the set of edges that can reach $pc$ (respectively, can be reached from $pc$). Then $\text{WrBt}$ is defined as:

$$\text{WrBt}.(pc, \cdot).1 \equiv \exists (\cdot, \cdot, \cdot) \in \text{Out}.pc \cap \text{In}.pc' : Wt.\text{op}.1.$$
the calls get added to the path slice. However, each of these guards may be irrelevant to the reachability of the target: for our purposes, it is sufficient that there is some feasible path from the start of each function on the stack to the location where the function calls the next function on the stack. On the other hand, if some live value can be written between the start and the location where the next function is called, then we should not slice away those edges. This is done by changing the "else" condition on the line 12 of PathSlice (Figure 1) to be:

    if ¬ℓk & ¬WrBt.(pc₀, pc₀).Live then Call.i else i − 1

We found that in some cases this enabled the slice to contain only those branches that guarded modifications to variables relevant to the property. This led to the discovery of the appropriate dataflow facts. Without this modification, whenever the path to the error location contained a deep call stack with several branches guarding the path from the entry of a function to the point where the next function on the stack was called, the resulting path slice was infeasible because of these irrelevant branches, which caused irrelevant data flow facts to be added to the system. However after this modification the resulting slice is not guaranteed to be complete.

5. Experiments

We have implemented the algorithm to generate path slices in the BLAST software model checker [18].

**Benchmark Programs and Property.** Table 1 shows our benchmark programs. The column LOC shows lines of code before and after preprocessing, the number of lines before preprocessing is obtained by simply running wc -l in the source directories, the preprocessed lines of code do not include comments or blank spaces. The procedures column show the number of modeled procedures. This does not include external C system libraries to which calls may have been made. The benchmark ijjpeg is taken from the Spec95 benchmarks suite.

We checked the correct behavior of file pointers. We instrumented the code to track the system calls fopen and fdopen to mark the return value as an open file pointer (in case it is non-null). For every fprintf, fgets, or fputs, we check that the file argument is an open file. Finally, we instrument fclose to expect an open file, and change the file state to closed.

Our methodology to check the file property was the following. For each instrumented program, we check that a certain function call _error_, introduced by the instrumentation cannot be called. Each call to _error_ in the program can be independently checked, and the program satisfies the property if _error_ cannot be called on any program execution path. Instead of checking each _error_ separately, we cluster calls to _error_ according to their calling functions, and then check each function that can potentially call _error_ independently. The column Number of checks in Table 1 shows the number of functions that can potentially call error directly (the first number), as well as the total number of points instrumented with calls to _error_. The column Results shows the number of checks on which the tool proved that the file access was safe, the tool claimed an error trace, and the tool timed out, respectively. We set a timeout limit of 1000s per check. The total time shows the total time of all checks that finished (i.e., excluding the 1000s for each check that timed out), and the Maximum time is the maximum time taken by any one check that finished. Finally, the number of refinements shows the number of abstract counterexample traces produced. These are the traces that were reduced using Algorithm PathSlice.

We found three violations of the specification in wuftpd. The part of the program relevant to the error trace is shown in Figure 4. The error trace says that the fgets in statfilecmd can fail. Since we do not model the library function getrlimit, it is possible for it to return a nonzero value, and hence it is possible for ftpd_popen to return a NULL file pointer. Since at other places of the program, the returned file pointer from ftpd_popen is checked for NULL, we believe that this is a bug. The other violations were similar. In privoxy, the error trace read data off a configuration file and accessed an (unchecked) file pointer based on the data read. Again, since we do not model configuration files, we flag this as a possible error.

Figure 5 shows the effect of the running the path slicing algorithm on the abstract counterexample traces produced in the check. The x-axis shows the size of the original traces in number of basic blocks. The y-axis shows the size of the reduced trace as a percentage of the original trace, computed as $\frac{\text{PathSlice \pi}}{\pi} \cdot 100\%$, where $\pi$
is the original trace. The y-axis is in logarithmic scale. On the average, the projected traces are less than 5% of the original traces. The largest projected trace is 57% of the original trace, this happens in openssh where the input trace has 47 operations, and the output has 27. However, as the trace sizes go up, the ratio of projected trace to the original trace goes down. This is true across all our benchmarks. In particular, for traces over 1000 basic blocks in size, the projected traces are less than 1% of the original trace. This confirms our intuition that the size of the counterexample traces grow with the size of the program, for simple properties, there is usually a small abstraction that is sufficient to prove the unsatisfiability of a trace, and conversely that there is a simple explanation of a counterexample.

Limitations. We now describe certain limitations that we found on further experiments. The first limitation is the use of depth first search in the context-free reachability algorithm which results in very long counterexamples. We are investigating breadth first search algorithms that find the shortest counterexamples. The second limitation is in BLAST’s imprecise modeling of the heap. The third is due to inefficiencies in the current implementation.

We tried to check the file property on the program mProp (version 2.1rc1), an IRC proxy. The program was 15K lines of code after preprocessing, had 152 functions, of which 14 were instrumented with a total of 25 possible error points. We were surprised to find that 9 checks failed with error found or BLAST not finding new predicates. On closer examination of the source code, we found that mProp was handling file pointers in the following way. There is a hash table (built as an array of linked lists) that keeps a map from channel names (strings) to file pointers. There are operations to add channels to this table, open or close a file pointer entry in the table, and to remove channels from this table. Since we do not model the heap precisely, BLAST was unable to reason about file pointers being put inside these linked lists. We believe that techniques from shape analysis [23] may help in this example.

Second, we tried to check the file property on gcc. We took gcc code from the Spec95 benchmark suite. The total number of modeled procedures was 2026. We instrumented 703 sites for checks of correct file usage, these instrumentations were scattered in 132 different functions. Again, we tried to check correctness for all instrumentations in the same function together. We met with limited success on gcc. Of the 132 checks we ran on, only 76 finished in the allotted time of 1200s per query. This time includes all the model checking, projection, and refinement time, but does not include the cost of building the alias and the mod/ref information. We looked at the breakdown of the time taken, and in all cases, the time was dominated by the computation of By and WrBt. We believe that efficient implementations of these analyses using state-of-the-art techniques like BDDs [6, 26, 21] to represent the information succinctly can ensure that the techniques scale to large programs. We are currently investigating such algorithms.

However, refinement steps that did finish in the allotted time confirmed our expectations that the projected trace will be much smaller than the original counterexample. Figure 6 shows the results for Algorithm PathSlice on 313 gcc counterexamples. The largest counterexample we encountered had 82,695 basic blocks, however, after projection, the number of operations was a 43. For larger counterexamples, the reduced trace was less than 0.1% of the original size. The reduced traces were amenable to usual counterexample analysis, but the original traces were not. The sizes of counterexamples also show that some form of slicing must be performed in order to get efficient counterexample analysis in counterexample-guided abstraction refinement, since the size of trace formulas generated is usually beyond the limit of current decision procedures.

Table 1. Benchmarks and analysis times.

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
<th>LOC</th>
<th>Procedures</th>
<th>Number of checks</th>
<th>Results</th>
<th>Total time</th>
<th>Max time</th>
<th>Number of refinements</th>
</tr>
</thead>
<tbody>
<tr>
<td>icront 2.9.5</td>
<td>cron daemon</td>
<td>12K/43K</td>
<td>124</td>
<td>100/25</td>
<td>100/60</td>
<td>22.25/5</td>
<td>9.56</td>
<td>13</td>
</tr>
<tr>
<td>wuftp 2.6.2</td>
<td>ftp server</td>
<td>24K/55K</td>
<td>205</td>
<td>33/59</td>
<td>30/50</td>
<td>2471.41/1</td>
<td>412.08</td>
<td>74</td>
</tr>
<tr>
<td>make 3.80</td>
<td>make</td>
<td>30K/39K</td>
<td>296</td>
<td>19/44</td>
<td>18/50</td>
<td>89.8/3.5</td>
<td>32.7</td>
<td>35</td>
</tr>
<tr>
<td>privoxy 3.03</td>
<td>web proxy</td>
<td>38K/51K</td>
<td>291</td>
<td>15/54</td>
<td>13/20</td>
<td>107.5/1</td>
<td>69.1s</td>
<td>13</td>
</tr>
<tr>
<td>jpeg</td>
<td>jpeg compression</td>
<td>31K/37K</td>
<td>403</td>
<td>21/43</td>
<td>21/00</td>
<td>128.6s/1</td>
<td>121.6</td>
<td>23</td>
</tr>
<tr>
<td>openssh 3.5.1</td>
<td>ssh server</td>
<td>50K/114K</td>
<td>745</td>
<td>24/84</td>
<td>230/1</td>
<td>2211.5/1</td>
<td>554.1</td>
<td>135</td>
</tr>
</tbody>
</table>

Figure 6. Trace projection results for gcc

References


