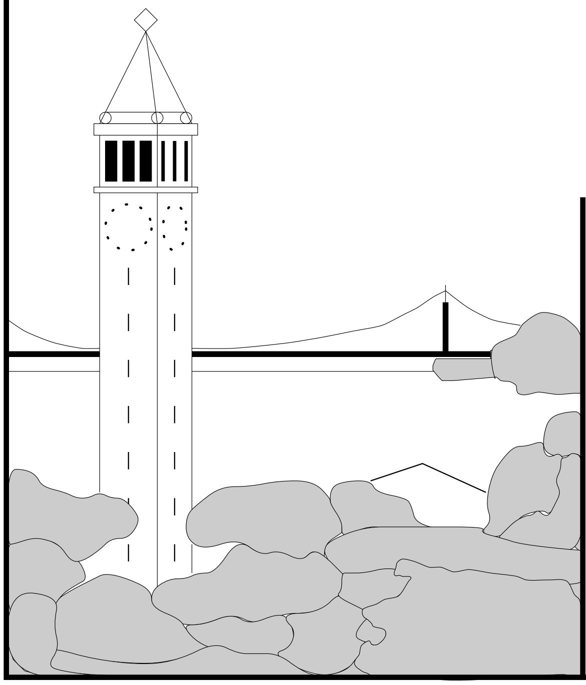
Building a Better Backtrace: Techniques for Postmortem Program Analysis

Ben Liblit
`liblit@cs.berkeley.edu`

Alex Aiken
`aiken@cs.berkeley.edu`



Report No. UCB//CSD-02-1203

October 2002

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Building a Better Backtrace: Techniques for Postmortem Program Analysis *

Ben Liblit
liblit@cs.berkeley.edu

Alex Aiken
aiken@cs.berkeley.edu

October 2002

Abstract

After a program has crashed, it can be difficult to reconstruct why the failure occurred, or what actions led to the error. We propose a family of analysis techniques that use the evidence left behind by a failed program to build a time line of its possible actions from launch through termination. Our design can operate with zero run time instrumentation, or can flexibly incorporate a wide variety of artifacts such as stack traces and event logs for increased precision. Efficient demand-driven algorithms are provided, and the approach is well suited for incorporation into interactive debugging support tools.

1 Introduction

Programs crash. In spite of the best efforts of software engineers, the grim reality is that zero-defect code is rare. Programs misbehave by violating fundamental rules of their implementation language (dereferencing a null pointer, failing to catch an exception), or by violating higher-level, domain specific invariants (failing an `assert()`, forgetting to acquire a lock).

Often, the flaw is not at the crash site (such as a pointer dereference), but rather at some earlier point (such as the forgotten initialization which allowed the null pointer to be seen). When correcting such flaws, the software engineer's main tool is the symbolic debugger. Coupled with a snapshot of program state at the point of failure, such as a Unix `core` file, the debugger helps the engineer reconstruct program activity leading up to the failure. A stack backtrace, for example, provides a partial chronology of how the program reached a crash site: `main()` called `compile()`, `compile()` called `parse()`, and so on. Other techniques such as event logging, single-step execution, and so-called "`printf` debugging" may also be used. Each provides a slightly more detailed chronology, giving the programmer more information about the execution path of the program in the lead-up to the crash. The process is tedious, though, and important actions may be hidden from view, such as functions which were called but silently returned before the crash.

Our goal is to improve this process. We adapt techniques from static program analysis to analyze programs which have already crashed. The intent is not to prove the program correct (for clearly it is not), but rather to create more detailed chronologies of program execution leading up to the error. We marshal information from various artifacts to reconstruct the set of paths that the program may have taken. The paths represent the possible execution histories given what is known from available postmortem evidence.

This paper makes three principal contributions:

- We describe a compact representation for realizable program paths, and present an efficient algorithm for computing the set of paths given a crash site and a global control flow graph [Section 2]. Our approach is a novel adaptation of earlier techniques for computing context-free language reachability in directed graphs.
- We show how this generic path discovery algorithm can be applied to a range of postmortem analysis situations, using a wide variety of post-crash artifacts to narrow the set of possible execution paths. Analyses of asymptotic complexity quantify the trade offs between precision and speed [Section 3].
- We propose strategies for condensing and summarizing detailed path information for use in a debugging system. We can present only those steps which all chronologies must have taken, or the user may selectively explore the various alternatives interactively [Section 4].

*This research was supported in part by NASA Grant No. NAG2-1210, NSF Infrastructure Grant No. EIA-9802069, and NSF Grant No. CCR-0085949. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

2 Path Representation

We begin by reviewing how possible program executions correspond to strings from a context free language of matched parentheses.

A single function can be represented as a single *control flow graph*. Nodes are program statements and edges represent possible transfers of control from one statement to another. Each path through the graph corresponds to one possible execution chronology. A *global control flow graph* generalizes this representation to multi-function programs. Each function contributes its own control flow graph. We split each function invocation into a pair of nodes: one representing the call, and one representing the subsequent return. We add an edge from the call node to the entry node of the called function, representing the transfer of control from caller to callee. A second edge, from the function's exit node back to the return node, represents return of control from callee back to caller. Because the only way from call to return is by going through the called function, we do not place an edge directly from the call node to the return node.

We assign a unique name to each call/return node pair, and use that to label the call-to-entry and exit-to-return edges. If the invocation site is named "l", then the call-to-entry edge is labeled "$(_l$" and the exit-to-return edge is labeled "$)_l$". Figure 1 shows an example. One function may have many callers; labels allow us to enforce proper call/return matching. If a function is called by crossing edge $(_l$, it must return across $)_l$, not some other mismatched edge $)_k$. Paths corresponding to valid executions are those in which all parentheses are properly matched; such paths are *realizable*. The sequence of labels traversed by a realizable path forms a string in a context free language of matched parentheses, where each invocation site defines a unique parenthesis pair. (Realizable paths corresponding to partial executions may contain zero or more unmatched call edges, corresponding to functions which have been called but have not yet returned.) The task of finding well matched paths, then, is a specific instance of a *context free language reachability* (CFL reachability) problem. We will not work with context free languages or grammars directly, but rather with the data structures of an efficient implementation: labeled graphs.

2.1 Segments, Paths, and the Route Map

A *segment* is a single hop from one graph node to another, and a *path* is a sequence of segments. We represent segments and paths using the following datatypes:

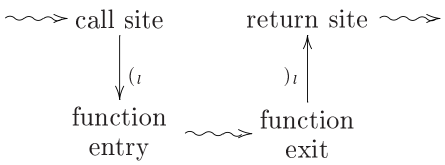


Figure 1: Context free reachability in global control flow graphs. Straight, solid arrows represent single edges. Wavy arrows represent known paths crossing zero or more edges.

$$\begin{aligned} N &::= \text{a graph node} \\ E &::= \text{an intraprocedural flow edge} \\ C &::= \text{an unmatched call edge} \\ L &::= \text{a call/return site label} \\ \textit{Segment} &::= \mathsf{flow}(E) \mid \mathsf{call}(C) \mid \mathsf{match}(L) \\ \textit{Path} &::= \mathsf{seed}(N) \mid \mathsf{compound}(N, \textit{Segment}, N, N) \end{aligned}$$

The argument to each *Segment* constructor justifies the use of that segment in a path: $\mathsf{flow}(e)$ represents crossing intraprocedural flow edge e; $\mathsf{call}(c)$ represents crossing function call edge c with no matching return; $\mathsf{match}(l)$ represents crossing the function call edge with label l, traversing the body of the called function, and coming back along the return edge with label l. Although this last case actually visits many nodes within the called function body, a match segment encapsulates the entire invocation as a single hop directly from the call node to the return node. This lets us treat function invocation as atomic from the perspective of the caller.

A seed path $\mathsf{seed}(n)$ represents the empty path from node n to itself. This is not a self loop: it is an empty path which crosses zero edges. A compound path $\mathsf{compound}(n_0, s, n_1, n_2)$ represents an extended traversal which starts at node n_0, crosses segment s to reach node n_1, and then continues onward ultimately finishing at node n_2.

The details of how to travel from n_1 to n_2 are recorded externally, in a structure called the *route map*:

$$\mathsf{route} : N \times N \times \{\mathsf{matched}, \mathsf{unmatched}\} \rightarrow 2^{Path}$$

For any pair of nodes, the route map records the set of paths which start at the first node and end at the second. The third argument further distinguishes paths which must be fully matched from paths which are allowed to include zero or more unmatched call edges. Thus, a path term $\mathsf{compound}(n_0, s, n_1, n_2)$ coupled with a route map route represents not one path but rather the family of all paths which start at n_0, cross s to n_1, and then continue on to n_2 using any suffix path contained in $\mathsf{route}(n_1, n_2, \_)$.

$$\begin{tikzcd} n_1 \arrow[loop above, "e_1"] \arrow[r, "e_2"] & n_2 \end{tikzcd}$$

$$\begin{aligned}
\mathsf{route}(n_1, n_1, _) &= \{\, \mathsf{compound}(n_1, \mathsf{flow}(e_1), n_1, n_1), \\
&\qquad \mathsf{seed}(n_1)\} \\
\mathsf{route}(n_2, n_1, _) &= \{\} \\
\mathsf{route}(n_1, n_2, _) &= \{\, \mathsf{compound}(n_1, \mathsf{flow}(e_1), n_1, n_2), \\
&\qquad \mathsf{compound}(n_1, \mathsf{flow}(e_2), n_2, n_2)\} \\
\mathsf{route}(n_2, n_2, _) &= \{\mathsf{seed}(n_2)\}
\end{aligned}$$

Figure 2: Finite encoding of a path family

The level of indirection introduced by the route map allows for finite representation of infinitely large path families, such as those containing loops or recursion. Figure 2 presents a minimal example. Starting with $\mathsf{route}(n_1, n_2, \_)$, we have two choices. We might select the first, $\mathsf{compound}(n_1, \mathsf{flow}(e_1), n_1, n_2)$. This directs us to start at n_1, cross edge e_1, and continue onward using any suffix path from n_1 to n_2. We look up $\mathsf{route}(n_1, n_2, \_)$ again, and are presented with the same two choices. We can select the first and cross edge e_1 as many times as we like before eventually choosing $\mathsf{compound}(n_1, \mathsf{flow}(e_2), n_2, n_2)$, which takes us out of the loop and subsequently terminates with the empty seed path at n_2.

2.2 Path Discovery Algorithm

The standard demand-driven approach for computing CFL reachability starts with a basic graph and an initial "trigger" edge consisting of a self-loop at some node of interest. New edges are added starting at this trigger and expanding outward, using a work list to track frontier areas yet to be explored. New triggers are added for subroutines, and well-matched paths discovered within a function are propagated up to its callers. If multiple paths exist to a single node, an edge will be added only once; if all that matters is reachability, rediscovery by an alternate route is of no interest. The final answer is given by examining added edges having the original node of interest as one endpoint; the other endpoint of each such edge is a CFL-reachable node.

Our needs differ slightly. We are not interested in mere reachability: the fact that the program has run and crashed is an existence proof that the crash point is reachable. We are interested in the specific path or paths by which the crash point was reached. Thus, the step-by-step justification for reaching each reachable node must be recorded, and if multiple paths exist, then this too must be represented. Justifications are recorded by *Segment* constructors. Alternative paths are represented by non-singleton sets in the route map.

```
fun start(seed(n)) = return n
or start(compound(n, _, _, _)) = return n

fun finish(seed(n)) = return n
or finish(compound(_, _, _, n)) = return n

fun discover(p, m) =
    let
        n1 = start(p)
        n2 = finish(p)
    in
        if (n1, n2, m) ∉ domain(route)
            work := work ∪ {(n1, n2, m)}
        route(n1, n2, m) := route(n1, n2, m) ∪ {p}
```

Figure 3: Utility functions for path discovery

Figure 3 presents some basic utility functions used by the main algorithm. The $\mathsf{start}()$ and $\mathsf{finish}()$ functions extract the endpoints of a path term. While each compound path term represents a family of paths, that family must always share the same endpoints; thus, it is reasonable to talk about the endpoints of a compound path term. The $\mathsf{discover}(p, m)$ function manages discovery of a new matched or unmatched path. Here we see two subtle differences from a standard CFL reachability algorithm. First, instead of checking whether an edge has already been added, we check whether the new path's endpoints are already in the domain of the route map. We define $\mathsf{domain}(\mathsf{route})$ as the set of triples (n_1, n_2, m) such that $\mathsf{route}(n_1, n_2, m) \neq \emptyset$. Thus, $\mathsf{domain}(\mathsf{route})$ roughly corresponds to the set of edges added by a standard CFL reachability algorithm.

The second change affects how a new path is recorded. Standard reachability would add an edge if none were already known. Our needs mandate that we remember all paths, so the new path is unconditionally added to the set of known paths for the given matching flag and endpoints. The route map is similar to a solution table from dynamic programming, but rather than each entry being set just once, entries may be updated (enlarged) as the main algorithm progresses. Observe here that $\mathsf{discover}$ maintains a critical route map invariant: the only paths added to $\mathsf{route}(n_1, n_2, \_)$ are those starting at n_1 and ending at n_2.

The main path discovery algorithm is parameterized in terms of six arguments:

N : the set of all nodes in the graph

E : a set of pairs from $N \times N$ defining interprocedural control flow edges

$\Lambda^{(}$: a set of triples from $N \times N \times L$ defining interprocedural call edges, where L is the set of call/return site labels

$\Lambda^{)}$: a set of triples from $N \times N \times L$ defining interprocedural return edges, where L is the set of call/return site labels

C : a set of pairs from $N \times N$ defining interprocedural call edges which are not to be matched up with corresponding return edges

ω : the specific node in N at which the program has crashed

The distinction between $\Lambda^{(}$ and C is somewhat nonstandard: in typical CFL reachability any call edge might be matched or unmatched. For our purposes, it will sometimes be useful to explicitly distinguish those call edges which must be matched ($\Lambda^{(}$) from those which must be unmatched (C); see Section 3.2 for a situation where this is useful.

Figure 4 presents the main path discovery algorithm which uses a work list to explore the graph, working backward from the crash site ω. (Although a forward search from program entry would discover the same paths, there are several practical reasons why working back from the crash site is preferable; we discuss this decision further in Section 3.) On entry to the main loop, the route map and work list contain a single unmatched seed path at node ω; this corresponds to the initial trigger edge used in standard CFL reachability. The loop repeatedly expands the set of discovered paths until there are no frontiers left to explore. New paths are discovered in any of five ways, corresponding to the five numbered calls to discover (see also Figure 5):

1. crossing an intraprocedural control flow edge
2. initiating a new query at the exit node of a function, in hope of finding the entry node
3. crossing from a return node to the corresponding call node, provided that the called function already has at least one known matched path
4. crossing from a return node to the corresponding call node at the moment that a matched path within the called function is discovered
5. crossing an unmatched call edge

Generalized CFL reachability is cubic in the number of graph nodes. Our algorithm exploits structural properties of the global control flow graph to avoid exploring any function more than once: the seed path added in case 2 is independent of which return edge led us to the function. This technique, first proposed by Reps *et al* [14], reduces the asymptotic complexity to $O(E + \Lambda^{(} + C)$. Since the size of C is bound by the size of $\Lambda^{(}$, this is equivalent to $O(E + \Lambda^{(})$.

3 Path Recovery Scenarios

The general path discovery algorithm presented above can be used in a variety of postmortem situations. In this section, we show that by modifying the graph on which backtrack is called, we can use a wide variety of post-crash artifacts.

Assume we are given a global control flow graph, consisting of a set of nodes (N), a set of interprocedural control flow edges (E), a set of labeled interprocedural call edges ($\Lambda^{(}$), a set of labeled interprocedural return edges ($\Lambda^{)}$), and a unique program entry node (α). Function invocations are split into node pairs: a call node n such that $(n, \_, l) \in \Lambda^{(}$ for some label l, and a corresponding return node n' such that $(\_, n', l) \in \Lambda^{)}$, with no direct flow from call to return: $(n, n') \notin E$. All of this information is static; it can be computed at compile time and stored offline until needed.

3.1 Crash Site Only

All we may know about a crash is the program counter where the crash occurred, with no information about program behavior before the crash or program state at the crash site.

Given just a crash node ω, best-effort path reconstruction proceeds as follows. Let C be the set of call edges stripped of their labels: $C = \{(n, n') : (n, n', \_) \in \Lambda^{(}\}$. Populate the route map by calling the main path discovery algorithm in Figure 4 as:

$$\mathsf{backtrack}(N, E, \Lambda^{(}, \Lambda^{)}, C, \omega)$$

The family of all possible paths leading to the crash is then given by:

$$\mathsf{route}(\alpha, \omega, \mathsf{unmatched})$$

In this special case, the paths discovered by the backtracking algorithm correspond to a context free grammar of matched parentheses (call/return pairs from $\Lambda^{(}, \Lambda^{)}$) with optional unmatched left parentheses (isolated call edges from $\Lambda^{(}$). For any single path from α to ω, the sequence of unmatched call edges corresponds to functions which had been called but not yet returned at the time of the crash. Given only a crash site and no additional information, we can present such possible stacks to the user but cannot automatically rule out one versus another.

```
    fun backtrack(N, E, Λ^(, Λ^), C, ω) =

        route(n_1, n_2, m) := ∅ for all n_1, n_2, m
        work := ∅
        discover(seed(ω), unmatched)

        repeat
            let
                t = (n_1, n_2, m) = any element of work
            in
                work := work − {t}

                foreach e = (n_0, n_1) in E
1:                  discover(compound(n_0, flow(e), n_1, n_2), m)

                foreach (n'_1, n_1, l) in Λ^)
2:                  discover(seed(n'_1), matched)
                    foreach (n_0, n'_0, l) in Λ^(
                        if (n'_0, n'_1, matched) ∈ domain(route)
3:                          discover(compound(n_0, match(l), n_1, n_2), m)

                if m
                    foreach (n'_1, n_1, l) in Λ^(
                        foreach (n_2, n'_2, l) in Λ^)
                            foreach (n'_2, n'_3, m') in domain(route)
4:                              discover(compound(n'_1, match(l), n'_2, n'_3), m')
                else
                    foreach c = (n'_1, n_1) in C
5:                      discover(compound(n'_1, call(c), n_1, n_2), m)
        until work = ∅
```

Figure 4: Main path discovery algorithm

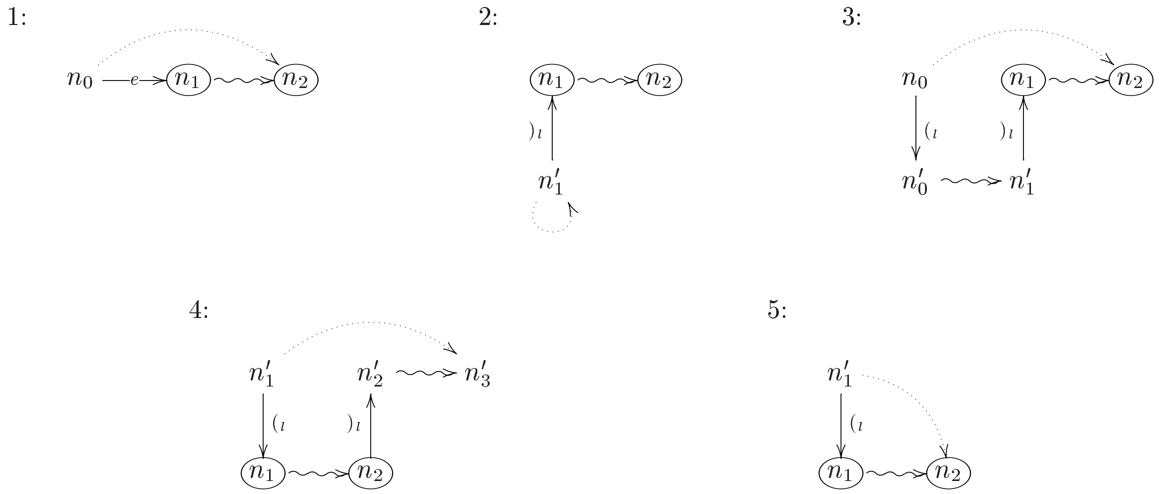


Figure 5: Case-by-case examples of path discovery. Straight, solid arrows represent single edges. Wavy arrows represent known extended paths. Dotted arrows represent newly discovered paths. Circled nodes n_1 and n_2 correspond to the task selected from the work list.

The asymptotic complexity is the same as that of the main algorithm: $O(E + \Lambda^{(})$.

3.2 Crash Site + Stack Trace

A slightly richer environment may identify not just the crash site but also the sequence of nested function calls which were executing at the time of the crash. Java exceptions, for example, record a stack trace at the point where the exception was thrown. Presentation of a stack trace is also standard fare for any source-level debugger, and stack traces can be readily extracted from Unix `core` files.

Each frame in a stack trace corresponds to a call edge with no matching return. Furthermore, these are the *only* call edges which may be unmatched: any other functions called before the crash must have successfully returned. Assume that the stack trace $\vec{S}$ is represented as a vector of call edges $\langle c_1, c_2, \ldots, c_{|\vec{S}|}\rangle$, where each $c_i \in \Lambda^{(}$. Instead of backtracking along the original global control flow graph $(N, E, \Lambda^{(}, \Lambda^{)}, \alpha)$, we construct a new graph $(\widehat{N}, \widehat{E}, \widehat{\Lambda^{(}}, \widehat{\Lambda^{)}}, \widehat{\alpha})$ as follows:

Let Σ be the set of stack positions, represented as integers in the range $[0, |\vec{S}|]$. Then the set of nodes $\widehat{N}$ consists of all nodes in the original graph paired with all possible stack positions. Edges are duplicated likewise, with both endpoints sharing the same stack position:

$$\widehat{N} = N \times \Sigma = \{(n, \sigma) : n \in N \wedge \sigma \in \Sigma\}$$
$$\widehat{E} = \{((n, \sigma), (n', \sigma)) : (n, n') \in E \wedge \sigma \in \Sigma\}$$

Crossing a matched call edge does not change the stack position, because a matched call edge must return before the crash. Thus:

$$\widehat{\Lambda^{(}} = \{((n, \sigma), (n', \sigma), l) : (n, n', l) \in \Lambda^{(} \wedge \sigma \in \Sigma\}$$
$$\widehat{\Lambda^{)}} = \{((n, \sigma), (n', \sigma), l) : (n, n', l) \in \Lambda^{)} \wedge \sigma \in \Sigma\}$$

At program entry, the stack is empty. We have entered zero of the call sites listed in $\vec{S}$, and encode this as a node with stack position zero:

$$\widehat{\alpha} = (\alpha, 0)$$

Crossing an unmatched call edge is allowed only when that edge matches the next edge expected in the stack trace $\vec{S}$. After such an edge has been crossed, the node on the other side must record the fact that we have consumed one element of the stack. We enforce these restrictions by careful definition of the set of unmatched call edges:

$$\widehat{C} = \{((n, \sigma - 1), (n', \sigma)) : (n, n', \_) = c_\sigma\}$$

For example, if the stack trace consists of the sequence of edges $\langle (a, b, \_), (c, d, \_), (e, f, \_)\rangle$, then the set of unmatched call edges would be:

$$\{((a, 0), (b, 1)), ((c, 1), (d, 2)), ((e, 2), (f, 3))\}$$

This encodes precisely the desired restrictions: the only way from the start node $(\alpha, 0)$ to any node $(\_, 1)$ is to reach node $(a, 0)$ and cross down to node $(b, 1)$. There is no way back to layer 0: the crossing down to layer 1 represents the first function call that had not returned by the time of the crash. As a second example, consider the special case of a crash in the program's top-level procedure: this will yield an empty stack trace, producing an empty set $\widehat{C}$, forcing all paths to be fully matched before reaching the crash.

Paths proceed monotonically downward through the stack, consuming stack frames one by one. The crash site ω should only be considered once each and every frame in the stack trace is accounted for:

$$\widehat{\omega} = (\omega, |\vec{S}|)$$

We invoke the main path discovery algorithm as:

$$\mathsf{backtrack}(\widehat{N}, \widehat{E}, \widehat{\Lambda^{(}}, \widehat{\Lambda^{)}}, \widehat{C}, \widehat{\omega})$$

The family of all possible paths leading to the crash is given by:

$$\mathsf{route}(\widehat{\alpha}, \widehat{\omega}, \mathsf{unmatched})$$

Although the new graph is $|\vec{S}|$ times larger than the original, the layers are connected only by edges in C. In effect, we are doing $O(\widehat{E} + \widehat{\Lambda^{(}})$ work on each of $|\vec{S}| + 1$ layers, plus $O(\vec{S})$ work to chain them together. The complete process, then, scales linearly with respect to the size of the stack trace: $O(\vec{S}(\widehat{E} + \widehat{\Lambda^{(}}))$. Also, note that the full graph need not be represented explicitly: given the original global control flow graph, the new graph can be represented implicitly, using code rather than storing $|\vec{S}|$-way duplicated data structures.

Stack traces motivate our use of a backward analysis. Backward analysis reaches return edges before matching call edges. If we discover a call edge but have not already crossed the corresponding return edge, then this edge can only be unmatched: if it is the next unmatched call edge in the stack trace, we continue; otherwise, we have reached a dead end and must explore elsewhere. A forward analysis would reach calls before returns. Upon reaching the next call edge in the stack, forward analysis cannot tell whether or not that edge should be matched, and so must explore both possibilities. Backward analysis is more deterministic, and therefore explores fewer dead ends.

3.3 Crash Site + Event Trace

Some environments may not provide stack trace information, but may allow observation and logging of selected program actions before the crash. If these actions can be tied to individual nodes in the control flow graph, then they can be used to narrow the set of possible paths to the crash point. We call such actions *events*, and a sequential log of events is an *event trace*.

Event traces may come from many sources. A line of code added manually that prints out the message "`reached line 15 in function foo()`" is an event, provided that each such message identifies a unique program point. The messages appearing in a typical Unix system log form an event trace for the kernel. Many Unix variants have provisions for recording the system (kernel) calls made by a process. Security audit trails under Windows NT, 2000, and XP provide a similar trace. Debugger-managed breakpoints are events. In general, any observable action with can be tied to a program point is an event.

In most cases, events are not logged internally by the suspect program, but rather appear in some external repository such as a logging daemon or a terminal window. This can make the event trace more robust than a stack trace in the face of extreme program failure: a misbehaving C program might trash its stack before crashing, leaving a stack trace which is simply garbage. An externally logged event trace will survive such misbehavior. (Note, however, that we do assume that even a crashed C program obeys its own control flow graph. If a program directly modifies its program counter or overwrites a saved return address, it is effectively crossing graph edges that do not exist; such misbehavior is outside the scope of this paper.)

If each event uniquely identifies a program point, then we can augment our static global control flow graph with a set $N_V \subseteq N$ of event nodes: these are nodes which, if crossed, must emit a traced event. Following a crash, we collect an event trace $\vec{V}$, represented as a sequence of nodes $\langle n_1, n_2, \ldots, n_{|\vec{V}|} \rangle$ where each $n_i \in N_V$.

Let Υ be the set of event trace positions, represented as integers in the range $[0, |\vec{V}|]$. We build a new graph by combining nodes with event trace positions, as for stacks. However, we can be somewhat more selective here, because events are mandatory: if the next event we expect to see is at node n, then we cannot cross any other node in N_V before reaching n; if we had reached some other event node in N_V, then that node would have appeared next in the event trace rather than n. This differs sharply from stack traces, where the expectation to see a given unmatched call edge does not block us from visiting (and returning from) arbitrary other functions first. We express the mandatory nature of event nodes by selectively knocking out certain elements of the new node set:

$$\begin{aligned}\widehat{N} = \quad & \{(n, \upsilon) : n \in N \wedge \upsilon \in \Upsilon\} \\ & - \{(n, \upsilon - 1) : n \in N_V \wedge n \neq n_\upsilon\}\end{aligned}$$

Equivalently:

$$\begin{aligned}\widehat{N} = \quad & \{(n, \upsilon) : n \in N - N_V \wedge \upsilon \in \Upsilon\} \\ & \cup \{(n_\upsilon, \upsilon - 1) : \upsilon \in \Upsilon\}\end{aligned}$$

Thus, if nodes a, b, and c are event nodes, and the event trace consists of $\langle a, b, b \rangle$, then $\widehat{N}$ would contain $(a, 0)$, $(b, 1)$, and $(b, 2)$, but no other clones of a, b, or c. This implies, for example, that node c can never be crossed. In general, exactly one node from N_V will appear for each event trace position $\upsilon \in [0, |\vec{V}|)$. No event node can ever appear with event trace position $|\vec{V}|$, because once we have observed all logged events, no more can be emitted before the crash.

The new edge sets are constructed so as to transition from one event trace position to the next when crossing an event node:

$$\begin{aligned}\widehat{E} = \quad & \{((n, \upsilon), (n', \upsilon)) : (n, n') \in E \wedge n \notin N_V \wedge \upsilon \in \Upsilon\} \\ & \cup \{((n_{\upsilon+1}, \upsilon), (n', \upsilon + 1)) : (n_{\upsilon+1}, n_2) \in E \wedge \upsilon \in \Upsilon\}\end{aligned}$$

This formulation transitions to the next event trace position upon departure from an event node, rather than upon arrival at it. That is correct provided that a crashing event node does not emit an event. If crashes happen before event emission, an adjusted formulation is straightforward. If event nodes cannot crash, the distinction is moot.

Without loss of generality, we can assume that function call nodes and function exit nodes are never event nodes. Therefore:

$$\begin{aligned}\widehat{\Lambda^{(}} &= \{((n, \upsilon), (n', \upsilon), l) : (n, n', l) \in \Lambda^{(} \wedge \upsilon \in \Upsilon\} \\ \widehat{\Lambda^{)}} &= \{((n, \upsilon), (n', \upsilon), l) : (n, n', l) \in \Lambda^{)} \wedge \upsilon \in \Upsilon\} \\ \widehat{C} &= \{((n, \upsilon), (n', \upsilon)) : (n, n', \_) \in \Lambda^{(}\}\end{aligned}$$

No events have been seen when the program first begins; all events in the trace must have been seen by the time the program crashes. Thus:

$$\widehat{\alpha} = (\alpha, 0) \qquad\qquad \widehat{\omega} = (\omega, |\vec{V}|)$$

Given these definitions, we build the route map by calling:

$$\mathsf{backtrack}(\widehat{N}, \widehat{E}, \widehat{\Lambda^{(}}, \widehat{\Lambda^{)}}, \widehat{C}, \widehat{\omega})$$

The set of paths that could have led to the crash is given by:

$$\mathsf{route}(\widehat{\alpha}, \widehat{\omega}, \mathsf{unmatched})$$

As in the previous scenario, the layered nature of the new graph limits the asymptotic increase in work. We do $O(\widehat{E}+\widehat{\Lambda^{(}})$ work on each of $|\vec{V}|+1$ layers, for $O(\vec{V}(\widehat{E}+\widehat{\Lambda^{(}}))$ work overall. As before, the generated graph can be represented implicitly rather than by literal duplication of the original graph.

3.4 Other Variations

The stack trace and event trace scenarios presented above serve to illustrate the key concepts whereby information about the crash is used to restrict the set of realizable paths. We briefly consider several variations here, to show how our approach can accommodate other typical debugging scenarios. In each case, the main path discovery algorithm remains the same; we simply call it with graphs constructed in slightly different ways.

3.4.1 Stack + Event Trace

The strategies used in Sections 3.2 and 3.3 are largely orthogonal, and can be combined without difficulty. Nodes in the constructed graph will form a subset of $N \times \Sigma \times \Upsilon$, where the start node is $(\alpha, 0, 0)$, the crash site node is $(\omega, |\vec{S}|, |\vec{V}|)$, and the edges are built in the obvious manner. In the worst case, we may need to explore every stack/event state pair, giving an overall asymptotic complexity which is linear in the product of stack and event trace sizes: $O(\vec{S}\vec{V}(\widehat{E}+\widehat{\Lambda^{(}}))$.

3.4.2 Multiple Event Traces

Section 3.3 listed some of the many potential sources of event trace information. A single program may actually be under observation using several of these mechanisms at once. If all events are timestamped, they can be uniquely ordered and treated as a single event trace. If timestamps are not available, then effectively we have a vector of t uncorrelated event streams $\langle \vec{V}_1, \vec{V}_2, \ldots, \vec{V}_t \rangle$. We can allow arbitrary interleavings of events by applying the strategy used in Section 3.3 t times, effectively giving us nodes of the form $(n, \upsilon_1, \upsilon_2, \ldots, \upsilon_t)$. Asymptotic complexity is linear in the product of all trace sizes: $O(\vec{S}\vec{V}_1\vec{V}_2 \ldots \vec{V}_t(\widehat{E}+\widehat{\Lambda^{(}}))$

3.4.3 Unknown Crash Site

All scenarios we have considered thus far assume that the crash site is known. If it is not, then we can still perform a best-effort analysis based on other available information. If only a stack trace is given, then the crash node can be any node within the last called function. We can perform backward path discovery from the entry node of this terminal function, and then consider all well matched forward extensions of these paths to other nodes within the same function. Given just an event trace, we perform backward path discovery from the last event node seen in the trace, and then consider all (matched or unmatched) forward extensions of these paths that do not cross any additional event nodes.

3.4.4 Ambiguous Stack Trace

It may not always be possible to translate a real stack trace into the idealized form used in Section 3.2. A typical example is the stack report given by uncaught Java exceptions. Each frame is described in terms of a fully qualified method name, a source file name, and a line number within that file. Suppose a single function is called twice on one line, as in "`foo(bar(), bar())`". If an exception is raised during either call to `bar()`, we will not be able to tell which call site was active at the time of the error.

Recall that adherence to the observed stack trace is enforced via $\widehat{C}$, the set of unmatched call edges. When the stack trace is unambiguous, $\widehat{C}$ contains one and only one edge of the form $((\_, \sigma-1), (\_, \sigma))$ for each $\sigma \in \Sigma$, corresponding to the fact that there is one and only one call edge for each stack position. If any given stack position σ is found to be ambiguous, we add one unmatched edge transitioning from $\sigma-1$ to σ for each candidate call site. We expect such situations to be rare in practice.

3.4.5 Ambiguous Event Trace

This scenario is similar to that just presented. If a given event cannot be unambiguously associated with a unique node in the control flow graph, we add additional edges to $\widehat{E}$ reflecting the various possibilities. Thus, if the σ'th event is ambiguous, then each candidate event node n and successor n' will yield one edge $((n, \sigma-1), (n', \sigma))$ in the generated graph.

3.4.6 Incomplete Event Trace

Event nodes will generally be sparse with respect to the complete program. Even so, a complete event trace from launch to crash may become excessively long if the program is long lived, or if event nodes occur within heavily trafficked loops. Many of the sources for event traces listed in Section 3.3 have finite logging capacity: text scrolls off of windows; log files are periodically rotated and removed; an in-program event log might use a fixed-size rotating buffer; and so on. The Intel Pentium 4 family of processors is capable of recording the source and destination address of the four most recently taken branches [8].

Such suffix traces may be used as follows. Once the first event in the trace has been seen, all remaining events must have been logged faithfully. Thus, nodes of the form $(\_, \sigma), \sigma > 0$ can be treated as originally suggested, along with their related edges. For nodes of the form $(\_, 0)$, we retain the complete set of cloned nodes given by $N \times \{0\}$, and all associated edges. We cannot omit any event nodes at trace position zero, because it is possible that these nodes were reached before the first event in our incomplete trace. Instead, for the first event node in the trace (n_1) and for each edge (n_1, n), we include both $((n_1, 0), (n, 0))$ as well as $((n_1, 0), (n, 1))$ as edges in the generated graph. The first of these edges represents any "forgotten" crossings of node n_1, while the second edge represents the crossing of n_1 that appears as the first event in the event trace.

Suffix traces work well with a backward analysis: the first event node we see must be the last one in the trace. In a forward analysis, we would discover event node n_1 but be unable to tell if this represents the first node in the trace, or some earlier crossing which has since been forgotten. The forward analysis would have no choice but to pursue both possibilities. As seen earlier, backward analysis is more deterministic and therefore explores fewer dead ends.

3.4.7 Program Counter Sampling

Periodic sampling of the program counter is a well known strategy for profiling code. This is typically implemented in software using clock interrupts, although the Alpha processor contains hardware support for sampling with such low overhead that it can be left running continuously even on production systems [2]. Sampled program counter values are not strictly events, because no program counter is guaranteed to be sampled each time it executes. However, we can exploit such information using a slightly more permissive formulation than that used in Section 3.3:

$$\begin{aligned} \widehat{E} = \quad & \{((n, \upsilon), (n', \upsilon)) : (n, n') \in E \wedge \upsilon \in \Upsilon\} \\ & \cup \{((n_{\upsilon+1}, \upsilon), (n', \upsilon + 1)) : (n_{\upsilon+1}, n') \in E \wedge \upsilon \in \Upsilon\} \end{aligned}$$

We have removed the "$n \notin N_V$" restriction from the first set of edges. This gives us more edges and requires more graph exploration, reflecting the less precise information provided by program counter sampling compared to mandatory event nodes.

3.4.8 Dynamic Dispatch

Our algorithm assumes that the global control flow graph includes complete static call information. Dynamic dispatch, such as through function pointers or virtual method invocations, can be managed using any of several well-known techniques for forming conservative static approximations of dynamic call graphs. While such approaches are fairly standard in static program analysis, there is one respect in which our post-mortem viewpoint is uniquely beneficial: stack traces reveal the real addresses of unique call/return sites. There is no indirection, and therefore each stack frame can be treated as a completely static call. The unmatched call edges placed in $\widehat{C}$ need never be dynamic.

3.4.9 Demand Driven Analysis

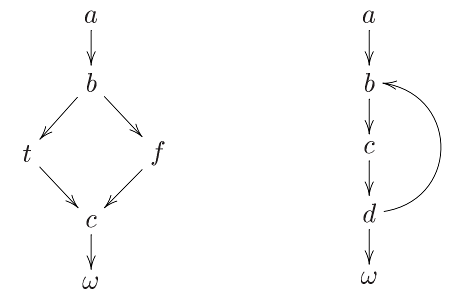
Although the main algorithm is fairly efficient, event traces can grow quite large, and therefore it may be desirable to reconstruct paths on demand, rather than exhaustively. If the work list is managed as a first in, first out queue, then paths will be discovered starting close to the crash site and gradually radiating outward. Code closer to the crash is more likely to be related to the error, so a backward analysis works well here. It is easy to detect when one has transitioned to the previous position in a stack or event trace, and this forms a logical unit of work to compute (in $O(\widehat{E} + \widehat{\Lambda^{(}})$ time) and present to the user before pausing or continuing to search in the background. If reconstruction is using just a stack or just one event stream, then preliminary results for a suffix of the trace are guaranteed to participate in the final $\widehat{\alpha}$-to-$\widehat{\omega}$ solution. If multiple traces are used, per Section 3.4.1 and 3.4.2, then it is possible that some preliminary paths may dead end later on; demand driven analysis can display preliminary results as a conservative superset which may be narrowed down by further exploration.

4 Path Summarization

The techniques described above populate the route map, with $\mathsf{route}(\widehat{\alpha}, \widehat{\omega}, \mathsf{unmatched})$ as the basis for identifying all realizable paths to the crash site. However, a route map is not immediately useful to a human software engineer searching for a bug. Even the minuscule route map given in Figure 2 requires careful, detailed reading to understand the family of paths it encodes. Enumerating all paths is no solution, as that set will in general be infinite. Summarization is needed to condense the route map to an accessible form.

One useful presentation is to show the node-by-node steps that all paths share in common. These represent program activity which must have occurred. When several paths are available, we should explicitly show the user that we do not know which was taken; once those paths join back together, though, we can pick up again along this common continuation.

Figure 6 gives two examples of useful summaries of intraprocedural paths. In the first example, it is clear



$\boxed{\langle a, b, \ldots, c, \omega \rangle}$

$\langle a, \ldots, b, c, d, \omega \rangle$
$\langle a, b, \ldots, c, d, \omega \rangle$
$\langle a, b, c, \ldots, d, \omega \rangle$
$\langle a, b, c, d, \ldots, \omega \rangle$

Figure 6: Sample path summaries

where a gap appears due to two choices for how to continue on from b to reach ω. This appears in the route map as two elements in the set $\mathsf{route}(b, \omega, \_)$. In the second example, gaps represent loops which may be traversed any number of times. Any of the four summaries offered is correct, though they vary in which parts of the execution they emphasize.

We produce path summaries of this form for general multi-procedure graphs as follows. Build a new graph whose nodes are in one-to-one correspondence with those in the graph that was used during path discovery. Add edges to this graph only where the corresponding edges appear as single-segment hops in the route map. That is, if $\mathsf{route}(n_1, n_2, \_)$ appears anywhere in the route map, then create an edge from n_1 to n_2 in the newly constructed graph.

Because we don't care what the single-step segment is, this new graph directly connects call nodes to their matched return nodes without detouring through the called function. Unmatched call edges do appear as edges in the new graph, but matched call/return pairs are reduced to a single direct edge from call to return. This can be thought of as a view of the realizable paths with zero detail inside function calls.

We now perform a series of dominator computations on this new graph. The last node in the summary must be ω, the crash site. The immediate dominator of ω with respect to root node α will be the closest predecessor of ω along every path from α. This node, $\mathsf{idom}(\omega)$, must be the second-to-last node in the summary. The third to last node will be $\mathsf{idom}(\mathsf{idom}(\omega))$, and so on, until eventually we reach α, the first node in the summary. The nodes in the summary are exactly those nodes on a root-to-leaf walk of the dominator tree from α to ω, computable in time which is nearly linear with respect to the number of graph nodes [12].

At each stage in the tree walk, we can check whether the immediate dominator of a node is also the unique predecessor of that node. If so, then the transition from $\mathsf{idom}(n)$ to n was the only possible progression. If not, then there must be a gap between $\mathsf{idom}(n)$ and n which can be spanned by more than one path. Inject a gap marker into the summary at this point to inform the user that we do not know which path was taken. Alternately, at each stage in the tree walk check the size of the route map entry from that node to ω; if it lists more than one path, then insert a gap. These two strategies produce identical results for the first example in Figure 6. For the second example, gap insertion before nodes with multiple predecessors yields $\langle a, \ldots, b, c, d, \omega \rangle$, while gap insertion after non-singleton route map entries yields $\langle a, b, c, d, \ldots, \omega \rangle$.

Because each step in the summary corresponds to a path segment from the route map, each step forward or backward can be attributed to a concrete program action: intraprocedural flow, call to a function that does not return, or function call and return. In the later case, the user can "unfold" the invocation and interactively browse paths within the called function, selectively increasing the level of interprocedural detail as much as is useful for the problem at hand.

5 Related Work

The idea of exploring execution chronologies has received attention before in the form of *replay debuggers* [4, 5, 11, 13, 15, 17]. Replay debuggers periodically checkpoint program execution to allow fast incremental replay. This essentially creates the ability to travel backward in time: the unique, exact execution path is known, and all data values are available at any previous point in time.

Dynamic slicing identifies the subset of an executed program's statements which actually affect the value of a single selected variable at some point of interest [1, 9]. Recent work in this area uses static analysis of the program's data and control dependencies to reduce execution overhead [7] and can exploit information such as the dynamic call graph and debugger breakpoints [6]. Our approach is more lightweight. We reconstruct overall program flow rather than focusing on an individual variable, and present a family of possible paths without guaranteeing uniqueness. In exchange for reducing the level of detail, we can exploit a wider variety of post-crash artifacts with as little as zero run-time overhead.

Our use of paths to describe program executions relates to earlier work on program tracing and path profiling [3, 10]. Profiling builds an aggregate statistical model of the program over many runs in order to drive subsequent optimization or performance tuning. Program tracing can be far more detailed, but risks blowing up in time or space if too much information is retained. Techniques for presenting and navigating our path summaries remain an open area of study; current research in visualizing large traces and understanding dynamic program behavior may offer useful insights [16].

The principal ways in which we differ from these previous works are that (a) we assume no modifications of the program before execution and (b) we are able to take advantage of external information such as logs to give more precise information.

6 Conclusions

We have described a family of techniques for analyzing crashed programs and reconstructing the set of possible executions that may have led up to the crash. Though based on algorithms from static analysis, our approach takes advantage of unique information which is only available from actual program execution. We are able to glean evidence from a wide variety of postmortem artifacts, and offer flexible trade offs between the amount of information collected at run time and the specificity of its results. Full, deterministic program tracing is possible, but even a stack trace extracted from a *core* dump can be used to recover information about why a program has died.

References

[1] H. Agrawal, R. A. DeMillo, and E. H. Spafford. Debugging with dynamic slicing and backtracking. *Software—Practice and Experience*, 23(6):589–616, June 1993.

[2] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: Where have all the cycles gone? *ACM Transactions on Computer Systems*, 15(4):357–390, Nov. 1997.

[3] T. Ball and J. R. Larus. Programs follow paths. Technical Report MSR-TR-99-01, Microsoft Research, Redmond, Washington, Jan. 6 1999. Also Bell Labs/Lucent Technical Report BL0113590-990106-01.

[4] S. P. Booth and S. B. Jones. Walk backwards to happiness — debugging by time travel. In *3rd International Workshop on Automated Debugging*, pages 1–11, Linköping, Sweden, May 26–27 1997. University of Linköping.

[5] B. Boothe. Efficient algorithms for bidirectional debugging. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 299–310, Vancouver, British Columbia, June 18–21, 2000. *SIGPLAN Notices*, 35(5), May 2000.

[6] R. Gupta, M. L. Soffa, and J. Howard. Hybrid slicing: integrating dynamic information with static analysis. *ACM Transactions on Software Engineering and Methodology*, 6(4):370–397, Oct. 1997.

[7] K. Inoue, M. Jihira, A. Nishimatsu, and S. Kusumoto. Call-mark slicing: An efficient and economical way of reducing slices. In *Proceedings of the 21st International Conference on Software Engineering*, pages 422–431. ACM Press, May 1999.

[8] Intel Corp. *System Programming Guide*, volume 3 of *IA-32 Intel Architecture Software Developer's Manual*. Intel Corp., Mt. Prospect, Illinois, 2001.

[9] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, Oct. 1988.

[10] J. R. Larus. Whole program paths. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 259–269, Atlanta, Georgia, May 1–4, 1999. *SIGPLAN Notices*, 34(5), May 1999.

[11] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, C-36(4):471–482, Apr. 1987. September 1986 Also available as BPR 12, Computer Science Department, University of Rochester, September 1986.

[12] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flow graph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, July 1979.

[13] R. H. B. Netzer and M. H. Weaver. Optimal tracing and incremental reexecution for debugging long-running programs. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 313–325, Orlando, Florida, June 20–24, 1994. *SIGPLAN Notices*, 29(6), June 1994.

[14] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95)*, pages 49–61, San Francisco, January 22–25, 1995. ACM Press.

[15] A. P. Tolmach and A. W. Appel. Debugging standard ML without reverse engineering. In *1990 ACM Conference on Lisp and Functional Programming*, pages 1–12. ACM, ACM Press, June 1990.

[16] R. J. Walker, G. C. Murphy, B. Freeman-Benson, D. Wright, D. Swanson, and J. Isaak. Visualizing dynamic software system information through high-level models. *ACM SIGPLAN Notices*, 33(10):271–283, Oct. 1998.

[17] P. R. Wilson and T. G. Moher. Demonic memory for process histories. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 330–343, Portland, Oregon, June 21–23, 1989. *SIGPLAN Notices,* 24(7), July 1989.