Control-Flow Recovery from Partial Failure Reports

Peter Ohmann Alexander Brooks Loris D'Antoni Ben Liblit

University of Wisconsin–Madison, USA
{ohmann,albrooks,loris,liblit}@cs.wisc.edu

Abstract

Debugging is difficult. When software fails in production, debugging is even harder, as failure reports usually provide only an incomplete picture of the failing execution. We present a system that answers control-flow queries posed by developers as formal languages, indicating whether the query expresses control flow that is possible or impossible for a given failure report. We consider three separate approaches that trade off precision, expressiveness for failure constraints, and scalability. We also introduce a new subclass of regular languages, the unreliable trace languages, which are particularly suited to answering control-flow queries in polynomial time. Our system answers queries remarkably efficiently when we encode failure constraints and user queries entirely as unreliable trace languages.

CCS Concepts • **Software and its engineering → Software testing and debugging**; *Software post-development issues*; Software performance; • **Theory of computation →** *Formal languages and automata theory*; *Regular languages*

Keywords Deployed software, failure reporting

1. Introduction

When programs crash at user sites, debugging is difficult and detailed execution information is very valuable. However, deployed applications usually cannot gather full traces of run-time information, and instead leave behind failure reports with varying detail. For example, nearly all failure reports contain a stack trace from the failing execution, but most do not contain a trace of all executed statements. Developers must weigh run-time overhead (including time, memory, and disk space) against the benefits of traced data for postmortem failure analysis. Prior approaches to post-deployment monitoring [4, 9, 17, 28, 33, 34, 41] routinely give up perfect information, which necessarily results in an *incomplete picture* of any failing execution.

Nevertheless, obtaining answers to questions about the failing run is a key part of debugging. LaToza and Myers [22] find that developers commonly ask *reachability* questions, often based entirely on the program's control flow. For example, developers asked about possible transitive function callers/callees and possible calling contexts. Additional cases might arise when debugging deployed applications. If developers suspect missed initialization, they might ask, "Is it possible that `init()` did not run?" or, "Could `x` have been used here before being initialized there?" Developers might also ask whether an application has set the appropriate privilege level, locked appropriate resources, or opened appropriate streams. Each of these queries may consider operations across all of execution or only in specific stack contexts. Some of these questions may be interactive (e.g., during an active debugging session), and require very fast answers. Others may be non-interactive (e.g., a batch analysis job that runs overnight), and allow time for more precise answers.

In this paper, we formalize and evaluate analysis techniques that cope with incomplete data from post-deployment failure reports. Our system allows developers to ask yes/no control-flow questions (e.g., may a particular statement have executed on the failing run?), and provides answers based on failure report data. We consider three different underlying solvers that present trade-offs in expressiveness for failure constraints, precision in analysis results, and scalability.

This work substantially extends that of Ohmann et al. [30], who extracted best-effort program coverage data given possibly-incomplete failure report data of a very limited class. We show that the failure data considered in that work actually forms a special subclass of regular languages (which we title the *unreliable trace languages*; see section 4), and we present an algorithm that can answer the queries posed by Ohmann et al. much more efficiently. We also define encodings for a much wider range of failure constraints, and allow more precise analysis by ensuring calling-context sensitivity (via our solver based on visibly-pushdown languages [2]).

After providing a motivating example and describing our goal of encoding failure reports to answer user queries at a high level (section 2), we discuss our primary contributions:

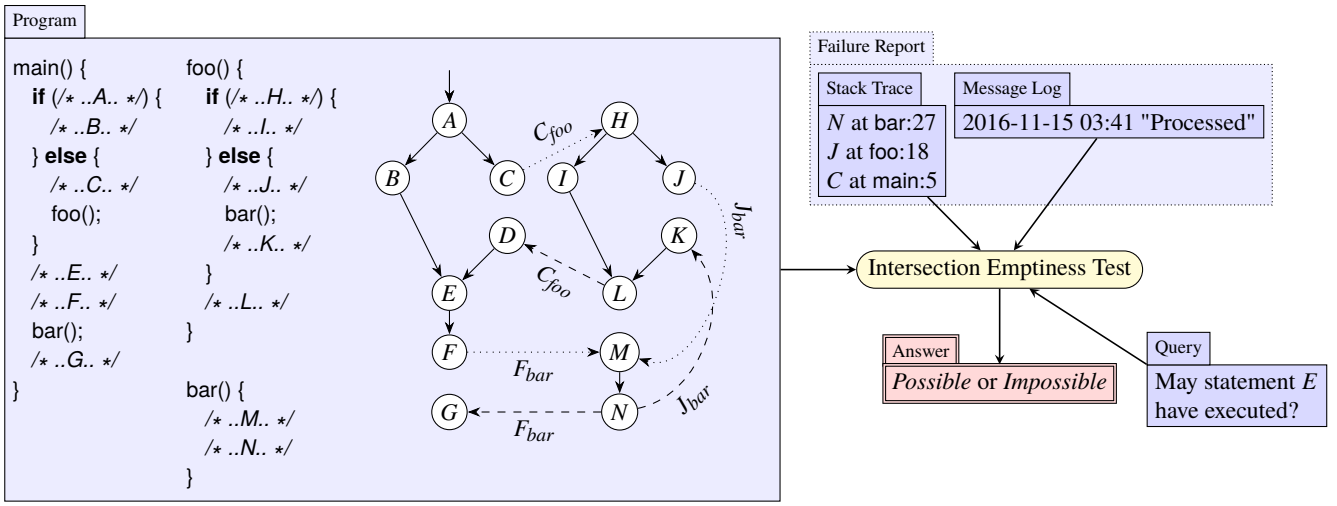


Figure 1. Overall system architecture and concrete example. Example inputs consisting of a program, failure report, and query flow into one of our main analysis algorithms, yielding an answer consisting of a single possible/impossible judgment.

1. We formally define the problem of answering control-flow queries from failure reports, and present a new system for answering these queries based on language intersection emptiness (section 3).
2. We introduce a new class of subregular languages, the unreliable trace languages, which, though unable to precisely express some failure report elements, are ideally suited to answering queries in polynomial time (section 4).
3. We describe encodings for various common features of failure reports, including stack traces, program coverage data, and call traces (section 5).
4. We perform an extensive evaluation of our techniques on a wide variety of failing applications (section 6). We can answer queries remarkably efficiently when we encode failure constraints entirely as unreliable trace languages.

Section 7 positions our approaches with respect to related work, and section 8 concludes.

2. Problem and Motivating Example

This section describes the structure of our system using the illustrative example given in fig. 1. Section 3 defines each shown element more formally.

The left side of fig. 1 shows a source code skeleton and its corresponding control-flow graph (CFG). Here, each node refers to a statement in the program, solid lines refer to intraprocedural control flow, dotted lines indicate calls, and dashed lines denote returns. Call and return edges are labeled with the call site and the called procedure. The CFG includes three procedures: main consists of nodes $\{A, B, C, D, E, F, G\}$, foo consists of $\{H, I, J, K, L\}$, and bar consists of $\{M, N\}$. Upon receiving a failure report like the one in the top right of fig. 1, we allow a developer to pose queries like the one shown in the bottom right of fig. 1.

The example failure report contains two elements: a failing stack trace (the program crashed at statement N in function bar, which was called from function foo at statement J, etc.), and a single log message from the failing run. For the purposes of this example, assume that we have statically determined that the message "Processed" could only be printed by either statement B or C. We discuss how to encode these and other common failure report elements in section 5.

The lower right corner of fig. 1 shows a user query as prose. We formally define our class of queries in section 3.3. In brief, we allow any control-flow query that can be expressed as a visibly-pushdown language [2]. Our system answers the query with respect to the provided CFG and failure report. That is, queries implicitly begin with: "On at least one path through the CFG consistent with the failure report, ..."

In each of our approaches, we first encode the CFG, each failure report element, and the query as formal languages whose member strings correspond to sequences of statements and call/return edge labels from the CFG. If the intersection of all of these languages is non-empty, then there exists such a sequence that (1) could occur in the CFG, (2) is consistent with the entire failure report, and (3) satisfies the query. Thus, we respond that the query is *Possible*. Otherwise, we answer *Impossible*. An imprecise solver may always answer *Possible* to a query. In fig. 1, a precise solver should answer *Impossible*, as there is no path through the CFG that passes through node E before resulting in the final crashing stack. The example contains neither loops nor recursion, and statement E must always follow the return from the in-progress call to foo seen in the stack trace. Thus, a precise solver will ensure that the language of the CFG respects matching labels on call and return edges (simulating a program stack).

Our three approaches vary in the machinery used to implement this high-level strategy. Specifically, the languages

for each element are encoded differently by each technique, and, therefore, the methods of computing intersections and checking for language emptiness differ as well.

3. Querying Using Formal Languages

In this section, we describe the general structure of our system. First, we formally define control flow graphs and the models of automata we will use. Second, we show how to encode the CFG as an automaton. Finally, we formally define our recovery problem, and present two approaches that answer queries by checking intersection emptiness of automata.

3.1 Preliminaries

We begin with the concept of a *tagged alphabet*. Given a set of symbols S, we use $(_S$ to denote the set of open symbols $\{(_s \mid s \in S\}$ and $)_S$ to denote the set of close symbols $\{)_s \mid s \in S\}$. The tagged alphabet of S is the set $\widehat{S} = S \cup (_S \cup)_S$.

Definition 1 *A control-flow graph (CFG) is a tuple* $G = (N, n_0, \mathbb{L}, E_i, E_c, E_r)$ *where:*

- N *is a finite set of nodes;*
- n_0 *is the entry node;*
- $\mathbb{L}$ *is a finite set of function names;*
- $E_i \subseteq N \times N$ *is a set of internal (intraprocedural) edges;*
- $E_c \subseteq N \times (N \times \mathbb{L}) \times N$ *is a set of function-call edges, such that for every* $(n, (n_1, \alpha), n') \in E_c$, $n = n_1$*; and*
- $E_r \subseteq N \times (N \times \mathbb{L}) \times N$ *is a set of function-return edges.*

In the following, we say that G operates over N and $\mathbb{L}$. Concretely, all edges in E_c are of the form $(n, (n, \alpha), n')$ denoting that control transfers from node n to n' through a call identified by function α. Notice that the call site is also part of the edge label. Similarly, edges in E_r are of the form $(n, (n_1, \alpha), n')$ where control transfers from node n to n' through a function return from α called from site n_1. In the CFG from fig. 1, edges are annotated with the calling node, subscripted by the called function label from $\mathbb{L}$ for brevity.

Let $E = E_i \cup E_c \cup E_r$ denote the set of all edges. We define the semantics of a control flow graph G using configurations over $N \times (N \times \mathbb{L})^*$, where G is in configuration $c = (n, \langle l_1 \cdots l_k \rangle)$ if the current node is n and the call stack contains the labels $l_1 \cdots l_k$. The initial configuration is $c_0 = (n_0, \varepsilon)$ where ε is the empty stack. A sequence of edges $\pi = \langle e_1 \cdots e_j \rangle$, such that $\pi \in E^*$ forms a *valid context-sensitive path* in the CFG G iff there exists a sequence of configurations $\langle (n_0, \bar{l}_0) \cdots (n_j, \bar{l}_j) \rangle$ such that for every $i \geq 1$:

1. If $e_i \in E_i$ and $e_i = (v, v')$, then $n_{i-1} = v$, $n_i = v'$, and $\bar{l}_{i-1} = \bar{l}_i$;
2. If $e_i \in E_c$ and $e_i = (v, (v, \alpha), v')$, then $n_{i-1} = v$, $n_i = v'$, and $\bar{l}_i = \bar{l}_{i-1}(v, \alpha)$;
3. If $e_i \in E_r$ and $e_i = (v, (v_1, \alpha), v')$, then $n_{i-1} = v$, $n_i = v'$, and $\bar{l}_{i-1} = \bar{l}_i(v_1, \alpha)$.

Informally, a valid context-sensitive path only allows return edges to be triggered for the last unmatched function call. Note that this definition allows paths to terminate at any node reachable from n_0, and in configurations that contain a non-empty stack, i.e., with pending function calls.

A sequence of edges $\pi = \langle e_1 \cdots e_j \rangle$, such that $\pi \in E^*$ forms a *valid context-insensitive path* in the CFG G iff there exists a sequence of configurations $\langle n_0 \cdots n_j \rangle$ such that for every $i \geq 1$:

1. If $e_i \in E_i$ and $e_i = (v, v')$, then $n_{i-1} = v$, $n_i = v'$;
2. If $e_i \in E_c$ and $e_i = (v, (v, \alpha), v')$, then $n_{i-1} = v$, $n_i = v'$;
3. If $e_i \in E_r$ and $e_i = (v, (v_1, \alpha), v')$, then $n_{i-1} = v$, $n_i = v'$.

Informally, a valid context-insensitive path does not force function calls and returns to be well-matched.

Definition 2 *Define the projection* $proj(\langle e_1 \cdots e_j \rangle)$ *of a path as* $\langle n_0\ proj(e_1) \cdots proj(e_j) \rangle$*, where for each edge e,*

$$proj(e) = \begin{cases} v' & \text{if } e \in E_i \text{ and } e = (v, v') \\ (_{v,\alpha} v' & \text{if } e \in E_c \text{ and } e = (v, (v, \alpha), v') \\ )_{v_1,\alpha} v' & \text{if } e \in E_r \text{ and } e = (v, (v_1, \alpha), v') \end{cases}$$

Informally, the projection of a path is the sequence of CFG nodes, calls, and returns traversed by the path. Formally, it is a sequence over the tagged alphabet $\widehat{N \cup (N \times \mathbb{L})}$.

In the rest of the paper, we use $N_{\mathbb{L}}$ to denote the alphabet $N \cup N \times \mathbb{L}$ and $\widehat{N_{\mathbb{L}}}$ to denote the alphabet $\widehat{N \cup (N \times \mathbb{L})}$.

Definition 3 (Traces) *Let G be a CFG. Given a valid context-sensitive path* π *in G,* $proj(\pi) \in \left(N \cup (_S \cup)_S\right)$ *is a valid context-sensitive trace in G. Given a valid context-insensitive path* π *in G,* $proj(\pi) \in \left(N \cup (_S \cup)_S\right)$ *is a valid context-insensitive trace in G.* $L_s(G)$ *and* $L_i(G)$ *denote the set of all context-sensitive and context-insensitive traces in G, respectively.*

The following theorem is immediate from our definitions and shows that every context-sensitive trace is also a context-insensitive trace.

Theorem 1 *For every CFG G,* $L_s(G) \subseteq L_i(G)$.

3.1.1 Symbolic Visibly-Pushdown Automata

Symbolic visibly-pushdown automata (s-VPAs) describe languages of nested words over large or infinite alphabets [11]. Nested words are linear encodings of words with hierarchical structure, such as traces of procedural programs. An s-VPA operates over a tagged alphabet, $\widehat{\Sigma}$, and manipulates a stack. The s-VPA pushes onto the stack only when reading symbols in $(_\Sigma$, and pops only when reading symbols in $)_\Sigma$.

This partition of symbols is a visibly pushdown alphabet, since it instructs the automaton on how to manipulate the stack: call symbols push onto the stack while return symbols pop from the stack (similar to function calls and returns in

a program trace). A nested word over the alphabet Σ is a sequence of symbols from $\widehat{\Sigma}^*$. To avoid explicit transition on all possible symbols, s-VPA transitions carry predicates describing sets of symbols from Σ. The set of predicates must be closed under Boolean operators, and checking satisfiability of a predicate must be decidable [11].[1] We let $\mathbb{P}_X$ represent the set of predicates in our algebra with free variables X. For example, if our alphabet is the set of numbers $\{1, \ldots, 100\}$, the predicate $\psi(x, y) = x \neq y \wedge x \leq 10 \wedge y \geq 5$ is in $\mathbb{P}_{x,y}$.

The class of *visibly-pushdown languages* recognized by s-VPAs maintain some of the expressiveness of context-free languages (including the ability to express matched calls and returns), but with important properties of regular languages. In particular, they are closed under intersection [2].

Definition 4 *A symbolic visibly-pushdown automaton is a tuple* $A = (Q, q_0, P, F, \Sigma, \delta_i, \delta_c, \delta_r)$ *where:*

- Q *is a finite set of states,*
- $q_0 \in Q$ *is the initial state,*
- P *is a finite set of stack symbols,*
- $F \subseteq Q$ *is the set of final (i.e., accepting) states,*
- Σ *is a set of input symbols,*
- $\delta_i \subseteq Q \times \mathbb{P}_x \times Q$ *is a finite set of internal transitions,*
- $\delta_c \subseteq Q \times \mathbb{P}_x \times Q \times P$ *is a finite set of call transitions, and*
- $\delta_r \subseteq Q \times \mathbb{P}_{x,y} \times P \times Q$ *is a finite set of return transitions.*

A transition $(q, \varphi, q') \in \delta_i$, where $\varphi(x) \in \mathbb{P}_x$, when reading a symbol a such that $\varphi(a)$ is *true*, starting in state q, updates the state to q'. A transition $(q, \varphi, q', p) \in \delta_c$, where $\varphi(x) \in \mathbb{P}_x$, and $p \in P$, when reading a symbol $(_a$ such that $\varphi(a)$ is *true*, starting in state q, pushes the symbol (p, a) on the stack and updates the state to q'. A transition $(q, \varphi, p, q') \in \delta_r$, where $\varphi(x, y) \in \mathbb{P}_{x,y}$, is triggered when reading an input $)_b$, starting in state q, and with $(p, a) \in P \times \Sigma$ on top of the stack such that $\varphi(a, b)$ is *true*; the transition pops the element on the top of the stack and updates the state to q'.

An accepting path for A is a sequence of configurations over $Q \times (P \times \Sigma)^*$, defined similarly to valid context-sensitive paths through a CFG. Informally, an accepting path is a sequence of transitions that matches return symbols to the most recent unmatched call symbol where all predicates on transitions are satisfiable. A nested word, w, is accepted by A iff there exists an accepting path for w in A. We use $L(A)$ to denote the set of all words accepted by A.

In figures and descriptions, we largely follow the style of D'Antoni and Alur [11], and label transitions "$I\colon \varphi_x$," "$C\colon \varphi_x/p$," and "$R\colon \varphi_{x,y}/p$" for Internal, Call, and Return respectively. Here, φ_x denotes a unary predicate on the input symbol x (for internal and call transitions), while $\varphi_{x,y}$ denotes a binary predicate (for return transitions) with y as the current input symbol and x as the input symbol for the matched call transition. For calls and returns, p is the pushed or popped stack symbol.

[1] In this paper, we only focus on the algebra where predicates are unions of intervals over integers and we allow equalities between variables. Concretely, each symbol in our alphabet—e.g., a node in the CFG—is mapped to a unique integer. This algebra has all the desired properties.

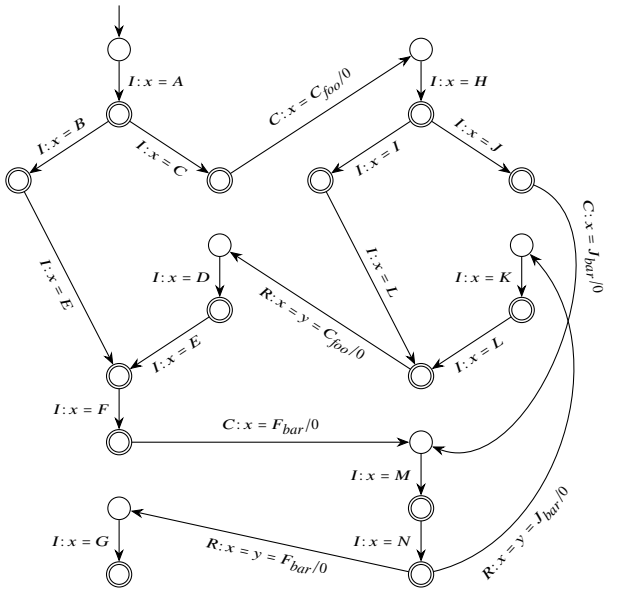


Figure 2. s-VPA formulation for the CFG from fig. 1

An s-VPA that contains only internal transitions is a Symbolic Finite Automaton (s-FA) [42, 43]. An s-FA that operates over a finite alphabet accepts *regular* languages. Notice that an s-FA can still operate over an alphabet $S' = \widehat{S}$ for some set S and therefore not use the stack.

3.2 Encoding CFGs as Automata

We consider two encodings of a control-flow graph G (from definition 1) as an automaton. The first preserves calling-context sensitivity by encoding G as an s-VPA. However, due to the complexity of checking intersection-emptiness for s-VPAs, we also consider a second approach that encodes G as an s-FA. The s-FA encoding cannot ensure calling-context sensitivity in analysis results, reducing precision but improving scalability. Prior work in model checking has used similar approaches to encode transition systems [3].

Figure 2 shows the s-VPA encoding for the CFG from fig. 1. States and transitions in fig. 2 match the nodes and edges of the CFG, plus one additional state and transition for each return site and each procedure entry. These additional transitions allow us to express failure report constraints and queries over procedure entries and return points.

Recall that an s-VPA containing only internal transitions over a finite alphabet recognizes a regular language. Regular languages can be recognized by (non-symbolic) finite state automata (FSA), and we omit the "$I\colon x =$" prefix on transitions in all context-insensitive automata figures. Figure 3 shows the FSA encoding for the CFG from fig. 1. Here, our translation results in a calling-context-insensitive representation of G,

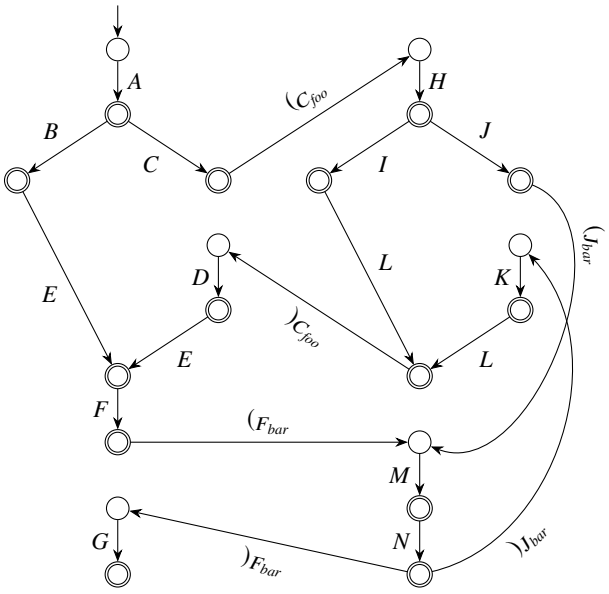


Figure 3. FSA formulation for the CFG from fig. 1

since we are unable to express call/return pairing relations. This automaton accepts strings over $\Sigma = \widehat{N_{\mathbb{L}}}$. Our translation is essentially identical to that for s-VPAs, except that call and return transitions are instead replaced by internal transitions on the corresponding tagged symbol from $\widehat{N_{\mathbb{L}}}$.

3.3 Problem Definition

We can now define the query recovery problem, which is the task tackled in this paper. We use the word "constraint" to describe a language. A constraint C is s-VPA-definable (resp. s-FA-definable) if there exists an s-VPA (resp. s-FA) A_C such that $L(A_C) = C$. Concretely, constraints will be given as effective models such as automata, regular expressions, or similar models. Consider the example queries given in section 1. Constraints involving whether an application was executing at the appropriate privilege level during a failure would be s-VPA-definable but not s-FA-definable if the privilege is established by the calling context (e.g., by a function such as doPrivileged()). However, s-FA are perfectly capable of expressing the query "Is it possible that init() did not run?" where calling context is not necessary.

The inputs of our problem are:

- G, a control-flow graph
- $\{FP_1, \ldots, FP_n\}$, a set of s-VPA-definable *failure constraints* describing the failure report. These can be given in various forms such as s-VPAs, s-FAs, or regular expressions.
- R, an s-VPA-definable *query constraint* describing a property we want to check against our failure report. For example, we might want to ask whether the node E was traversed, given our failure report.

Table 1. Complexity of the best known algorithm for computing intersection emptiness of n languages for various classes of languages

	s-VPA	s-FA	UTL
CS-CFG	EXPTIME	EXPTIME	EXPTIME
CI-CFG	EXPTIME	PSPACE [20]	PTIME (new)

Definition 5 (Context-sensitive recovery) *Given a control flow graph G with nodes in N and labels in $\mathbb{L}$, a set of s-VPA-definable* failure constraints $\{FP_1, \ldots, FP_n\}$ *of nested words over the alphabet $N_{\mathbb{L}}$, and an s-VPA-definable* query constraint *R of nested words over the alphabet $N_{\mathbb{L}}$, the context-sensitive query recovery problem is to check whether*

$$L_s(G) \cap \bigcap_i FP_i \cap R \neq \emptyset.$$

Since CFGs can be encoded as s-VPAs, and all the languages FP_i and R are s-VPA-definable, the query recovery problem is trivially decidable, though expensive. Intersection emptiness for s-VPA can be solved in exponential time. Moreover, since reachability is known to be co-PTIME, it is unlikely that a PSPACE algorithm exists for computing s-VPA intersection emptiness. The following problem has friendlier complexity.

Definition 6 (Context-insensitive recovery) *Given a control flow graph G with nodes in N and labels in $\mathbb{L}$, a set of s-FA-definable* failure constraints $\{FP_1, \ldots, FP_n\}$ *of words over the alphabet $\widehat{N_{\mathbb{L}}}$, and an s-FA-definable* query constraint *R of words over the alphabet $\widehat{N_{\mathbb{L}}}$, the context-insensitive query recovery problem is to check whether*

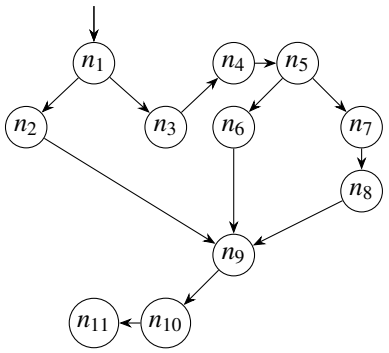
$$L_i(G) \cap \bigcap_i FP_i \cap R \neq \emptyset.$$

This definition only uses s-FAs. Since all the alphabets used in this paper are finite, the intersection emptiness problem for s-FAs has complexity PSPACE [20]. This paper investigates new algorithms that can efficiently solve the context-insensitive query recovery problem when all the languages FP_i and R, fall in a restricted class of languages called *unreliable trace languages* (UTL). In the next section, we present UTL and show that if all FP_i and R are definable as UTL, the query recovery problem can be solved in polynomial time. Table 1 summarizes these complexity results.

Interestingly, the context-insensitive analysis can be used to provide an unsound but complete algorithm for the context-sensitive problem. Observe that a nested word over the alphabet $N_{\mathbb{L}}$ is also a word over the alphabet $\widehat{N_{\mathbb{L}}}$. Ohmann et al. [32, section 3] formally states and proves this completeness result.

4. Unreliable Trace Languages

This section introduces a new class of subregular languages: the *unreliable trace languages*. The class is so named because



n_1.nodes = $\{A\}$ n_7.nodes = $\{J\}$
n_2.nodes = $\{B\}$ n_8.nodes = $\{(_{J_{bar}}\}$
n_3.nodes = $\{C\}$ n_9.nodes = $\{D, E, F, K, L, M, N, (_{F_{bar}},)_{C_{foo}},)_{J_{bar}}\}$
n_4.nodes = $\{(_{C_{foo}}\}$ n_{10}.nodes = $\{)_{F_{bar}}\}$
n_5.nodes = $\{H\}$ n_{11}.nodes = $\{G\}$
n_6.nodes = $\{I\}$

Figure 4. Condensation of CFG from fig. 1

the accepted languages correspond with traces of observed program points that are "unreliable" in that they do not guarantee that we have seen *every instance* of each observed point. Section 4.2 gives a polynomial-time algorithm for checking intersection-emptiness of the language induced from a context-insensitive CFG with n unreliable trace languages. Our approach does not encode the CFG and failure constraints as automata, instead operating directly over the CFG and testing for intersection-emptiness over context-insensitive traces as in definition 3.

4.1 Formal Characterization

For an alphabet Σ, unreliable trace languages are of the class

$$UTL = \left\{\Sigma^* \sigma_1 \Sigma^* \sigma_2 \Sigma^* \ \ldots \ \Sigma^* \sigma_n \Sigma^* \right. \\ \left. \text{for } n \geq 0 \text{ and such that all } \sigma_i \in \Sigma\right\}$$

So, for example, the unreliable trace language

$$\Sigma^* M \Sigma^* L \Sigma^* M \Sigma^*$$

(defined over $\Sigma = \widehat{N_{\mathbb{L}}}$ from fig. 1) indicates that accepted strings must contain an M at some point before an L which itself precedes another occurrence of M. However, other occurrences of M or L may occur at any point before, during, or after this sequence. Note that this characterization indicates that an unreliable trace language can be represented by a vector of symbols from Σ; the above example corresponds to the vector $\langle M, L, M\rangle$. The remainder of this paper uses this compact vector notation.

The unreliable trace languages are a sub-class of the piecewise-testable languages [35, 40], which can be characterized as any Boolean combination of unreliable trace languages [18]. They also precisely correspond with the failure report *obsYes* constraints from Ohmann et al. [30]. While very restricted, unreliable trace languages are able to express many possible elements of failure reports including program coverage data [17, 28, 33, 34, 41] and sampled observations [24]; with some loss of precision, they can also encode many other common failure report elements (see section 5).

Unreliable trace languages are closed under concatenation, but not under other Boolean operations such as negation, complementation, conjunction, and disjunction. Unfortunately, there are constraints that cannot be expressed (even imprecisely) as unreliable trace languages. For example, they cannot express disjunction. Thus, the printed log message from fig. 1 is inexpressible since the CFG node that printed it is ambiguous. The property of PTIME decidability for our context-insensitive recovery problem (see definition 6) is tightly tied to this language class. For more details, see Ohmann et al. [32, section 4], where we argue for the tightness of this class by proving that two small extensions to the unreliable trace languages result in NP-hard recovery problems.

4.2 Checking Emptiness

We check intersection-emptiness directly over the program's context-insensitive CFG, G (from definition 1), and do not encode $L_i(G)$ using automata. Inputs consist of:

- $G = (N, n_0, \mathbb{L}, E_i, E_c, E_r)$, a control-flow graph;
- $crash \in N$, the stopping point from the failing run; and
- $constraints = \langle c_1, \ldots, c_m\rangle$, a vector of unreliable traces where each c_i is a vector over $\widehat{N_{\mathbb{L}}}$.

We answer *Possible* if there exists a context-insensitive trace in G, $proj(\pi)$, such that $proj(\pi)_{|proj(\pi)|} = crash$ and each c_i is a subsequence of $proj(\pi)$ (i.e., $L_i(G) \cap \bigcap_i (c_i) \neq \emptyset$). Otherwise, we answer *Impossible*.

We first split all labeled edges in G, adding a new node named from the tagged symbol of the split edge label, forming graph G'. For the example CFG from fig. 1, this results in 6 new nodes: $(_{C_{foo}}$, $(_{F_{bar}}$, $(_{J_{bar}}$, $)_{C_{foo}}$, $)_{F_{bar}}$, and $)_{J_{bar}}$. We then collapse all strongly-connected components of G' to form the condensation of G', *sccG*. Note that *sccG* is always a directed acyclic graph (DAG). Each node of *sccG* is labeled with its set of collapsed nodes. Figure 4 shows the condensation for the CFG from fig. 1.

Intuitively, our goal is to walk forward through G, consuming symbols from each vector in *constraints* in order as we cross the corresponding nodes and edges. If every vector

in *constraints* has been completely consumed on some path to *crash*, then we have found a possible execution that is consistent with the program and failure report. Shifting our attention from the original CFG, G, to the condensation graph, *sccG*, has two key effects. First, passing through a nontrivial condensation node *scc* consumes all *scc*.nodes symbols appearing in the unconsumed prefix of any constraint. This is valid because each node in a nontrivial strongly-connected component can reach all other nodes in the same component as often as we like. Second, the lack of cycles in *sccG* means that if we do not consume some symbol when passing through the corresponding node, we will never have any future opportunity to do so. Thus, we proceed greedily: consume as many symbols as possible, as early as possible, because we will never get the chance to do so again.

More formally, checking intersection emptiness for $L_i(G)$ from definition 3 (i.e., the context-insensitive CFG language) with a set of unreliable trace language *constraints* (given as vectors per section 4.1) can be cast as a standard iterative data-flow analysis problem over *sccG*. The data-flow facts are vectors of vectors over $\widehat{N_{\mathbb{L}}}$, forming a finite lattice where the maximal element $\top$ = *constraints*, and there exists a unique minimal element $\bot$ that indicates that the provided constraints are *Impossible* to satisfy. The second smallest element is the vector of m empty vectors, which indicates that all constraints have been satisfied. Elements are ordered such that $f1 \sqsubseteq f2$ iff for all i, $f1_i$ is a trailing substring of $f2_i$ (i.e., $f2_i = v \mathbin{\shortparallel} f1_i$ for some vector v where "$\shortparallel$" is vector concatenation). Informally, this indicates that fact *f1* has consumed more of each initial constraint than *f2* has.

The standard [26] data-flow equations for a node *scc* are

$$in(scc) = \begin{cases} Init & \text{if } n_0 \in scc.\text{nodes} \\ \bigsqcap_{P \in Pred(scc)} out(P) & \text{otherwise} \end{cases}$$

$$out(scc) = F_{scc}(in(scc))$$

Instantiating this for our particular problem,

$$Init = constraints$$

$$f_1 \sqcap f_2 = \text{merge}(f_1,\ f_2)$$

$$F_{scc}(in(scc)) = \text{consume}(scc,\ in(scc))$$

Figure 5 shows function consume(). This function iterates through each element of the input vector, marking as many observations as possible as being now satisfied by passing through this strongly-connected component of *sccG*. For example, $\text{consume}\big(n_9, \big\langle \langle E \rangle, \langle F, E, F, G \rangle \big\rangle\big)$ returns $\big\langle \langle\rangle, \langle G \rangle \big\rangle$. Trivial strongly-connected components may only consume one element of each constraint vector, as paths through G may only traverse that node once. Note that no later information can ever invalidate prior satisfaction of a constraint.

Function consume(*scc, inFact*)
  input: *scc*, a node from the condensation graph
  input: *inFact*, the input fact from the predecessors of *scc* as a vector of vectors over $\widehat{N_{\mathbb{L}}}$ (indicating remaining constraints)
  outFact = $\langle\rangle$;
  foreach *constraint* $\in$ *inFact* **do**
    start = 1;
    while $constraint_{start} \in scc$.nodes **do** *start*++ ;
    if *start* > 2 and *scc* is trivial **then return** $\bot$;
    outFact $\shortparallel$= $\big\langle \langle constraint_{start} \ldots constraint_{|constraint|} \rangle \big\rangle$;
  return *outFact*

Figure 5. Update constraints satisfied by consuming as much of each constraint as possible

Function merge(*f1, f2*)
  input: *f1*, a data-flow fact as a vector of vectors over $\widehat{N_{\mathbb{L}}}$
  input: *f2*, a data-flow fact as a vector of vectors over $\widehat{N_{\mathbb{L}}}$
  assert $|f1| = |f2|$;
  if $\forall i \in 1 \ldots |f1|,\ f1_i = \alpha_i \mathbin{\shortparallel} f2_i$ *for some* α_i **then**
    return *f1*
  else if $\forall i \in 1 \ldots |f2|,\ f2_i = \alpha_i \mathbin{\shortparallel} f1_i$ *for some* α_i **then**
    return *f2*
  else return $\bot$;

Figure 6. Merge information from predecessor facts

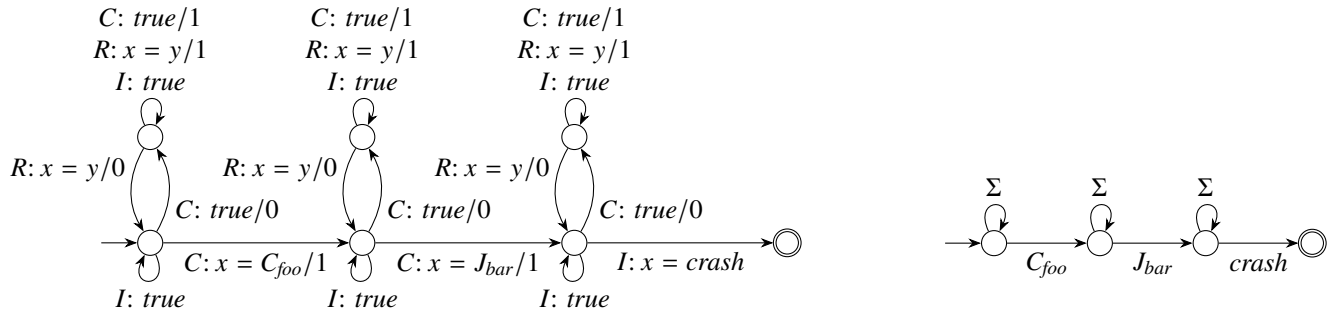
Figure 6 shows function merge(). Given a pair of input facts, this procedure selects the minimal fact via our definition of $\sqsubseteq$ above. That is, it finds the fact that is an element-wise trailing substring of the other. If neither fact qualifies, then the provided constraints are mutually unsatisfiable, and we return $\bot$. Since *sccG* is a DAG, this situation indicates that no possible path through *sccG* can satisfy all constraints, since each symbol from $\widehat{N_{\mathbb{L}}}$ appears in at most one strongly-connected component. For example,

$$\text{merge}\big(\big\langle \langle B, F \rangle, \langle C, E \rangle \big\rangle, \big\langle \langle F \rangle, \langle C, E \rangle \big\rangle\big) = \big\langle \langle F \rangle, \langle C, E \rangle \big\rangle$$

$$\text{merge}\big(\big\langle \langle F \rangle, \langle C, E \rangle \big\rangle, \big\langle \langle B, F \rangle, \langle E \rangle \big\rangle\big) = \bot$$

All of our data-flow functions are clearly monotonic and distributive; we only remove constraints from vectors during each F_{scc}, and always minimize remaining constraints when merging parent facts. Hence, our definition above computes the optimal meet-over-all-paths solution [26].

Note that our data-flow analysis is operating over a DAG; therefore, we topologically order the nodes in *sccG* and only compute the data-flow fact for each node once. Thus, we will need to perform merge() and consume() at most once for each strongly-connected component. Suppose that S is the number of nodes in *sccG*, C is the size of *constraints*, and L is the length of the longest constraint vector in *constraints*. In the worst case, merge() operates on a number of predecessor facts



(a) Represented as an s-VPA **(b)** Represented as an FSA

Figure 7. Encodings of crashing stack $[C_{foo}, J_{bar}, crash]$ as automata

on the order of S (all nodes in $sccG$), each of which contains C constraints of length L. We can process all predecessors in one merge() operation, which will find a predecessor vector that is an element-wise trailing substring of all others. This requires a linear scan over S predecessor facts, each with C constraints, each of size L; this requires $S \times C \times L$ total comparisons. Next, the call to consume() for a given strongly-connected component, scc, must find the first element of each $c \in C$ that is not a member of scc.nodes. In the worst case, each constraint still contains L elements, so this requires $C \times L$ set membership checks. Thus, merge() and consume() are each polynomial in the size of $sccG$, *constraints*, and the longest constraint vector. Since we perform at most S calls to each, the whole approach is polynomial.

For the final result, we answer *Possible* if $out(scc_i)$ is the vector of empty vectors (i.e., all constraints have been satisfied) and $crash \in scc_i$.nodes. Otherwise, we answer *Impossible*. For partial correctness proofs (including a proof of soundness with respect to definition 6), see Ohmann et al. [32, section 5].

5. Encoding Failure Reports

We now describe how to encode common failure report elements as s-VPA, FSA, and unreliable trace languages. Each failure element supplies a *constraint* that limits which runs can be consistent with the failure. All elements are defined over a CFG, $G = (N, n_0, \mathbb{L}, E_i, E_c, E_r)$. Unless stated otherwise, all FSA constraints are identical to their s-VPA counterparts but with all transitions internal over $\widehat{N_{\mathbb{L}}}$.

5.1 Crashing Stack

Many failure reports include a stack trace at the point of failure. A crashing stack trace consists of a sequence of symbols

$$[\ell_0, \ell_1, \ldots, \ell_n, crash]$$

where each $\ell_i \in N \times \mathbb{L}$ and $crash \in N$. Each ℓ_i indicates a call that remains on the program stack at the time of the failure. Note that we require call edge labels (rather than call sites from N) because even calls through function pointers are unambiguous when the call remains on the final crashing call stack. Intuitively, a stack trace constrains the ordered, unmatched calls on any corresponding execution. Concretely, we represent this constraint as

$$L(stack) = \mathbb{W}\ \ell_0\ \mathbb{W}\ \ell_1\ \mathbb{W}\ \ldots\ \ell_n\ \mathbb{W}\ crash$$

where $\mathbb{W}$ corresponds to a "well-matched" region of execution (corresponding to the rest of the program's execution between the calls that appear in the crashing program stack).

Automata For visibly-pushdown languages, $\mathbb{W}$ perfectly encodes matched sequences of calls and returns. Accepted runs end with unmatched calls exactly matching those in the final crashing stack. Figure 7a shows the s-VPA encoding of the example crashing stack from fig. 1. Regular languages cannot express matched call/return sequences, so we replace $\mathbb{W}$ with a simple Σ^* self-loop. Figure 7b shows the corresponding FSA for our running example.

Unreliable Trace Languages The FSA encoding of stacks very nearly expresses an unreliable trace language. The corresponding unreliable trace language is

$$\langle C_{foo}, J_{bar}, crash \rangle$$

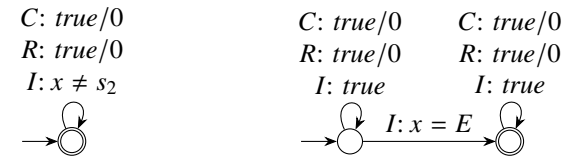
This vector does not directly encode that execution must halt after reading *crash* as the final symbol. Instead, this is managed via the parameter *crash* in section 4.2. This difference does not impact the precision of results.

5.2 Statement Coverage

Some production-run failure reports may include statement coverage data. For example, developers may gather coverage data for untested code [33, 34], or at specific program locations (e.g., call sites) to aid program analysis tools [15, 27, 28, 31]. Binarized statement coverage data consists of a set of independent observations, where each is a binary indicator (valued *true* or *false*). So, for example,

$$\{s_1\colon true, s_2\colon false\}$$

indicates that statement s_1 executed at least once during the failing run, and statement s_2 did not execute. We impose one constraint for each covered or uncovered statement.



(a) Encoding of "s_2: *false*" **(b)** Encoding of "E: *true*"

Figure 8. Encodings of coverage data entries as s-VPAs

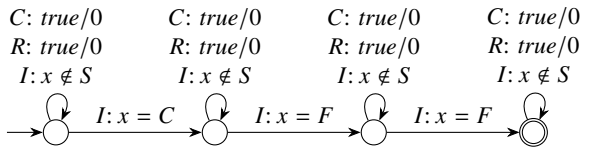


Figure 9. Call trace "$[C, F, F]$" as an automaton, where $S = \{C, J, F\}$

One could encode *false* entries via an automaton similar to that in fig. 8a. However, note that this automaton simply ensures that s_2 appears in no accepted strings (representing paths through the CFG). Thus, in practice, we use a much simpler approach, and remove s_2 from the CFG prior to encoding other failure constraints.

Automata For "s_1: *true*", we must ensure that any accepted string contains at least one occurrence of s_1. If we let $s_1 = E$, we are encoding precisely the query from fig. 1. Figure 8b shows the s-VPA to enforce this constraint.

Unreliable Trace Languages The above constraint (for s-VPAs and FSAs) expresses an unreliable trace language. Specifically, the constraint we want to impose for "E: *true*" corresponds to the vector $\langle E \rangle$.

5.3 Call Traces

Call traces record occurrences of specific call sites during a program's execution. These traces come in many forms, from logs of every call and return during a program's execution [9, 39, 44] to data gathered in short bursts [4]. Here we focus on a traces over some set of call sites $S = \{s_1, \ldots, s_n\}$ (not necessarily every call site in the program), where each $s_i \in N$. Bursty traces are encoded similarly, and add unconstrained execution (i.e., Σ^*) before and after each burst.

A call trace is of the form:

$$[c_1, c_2, \ldots, c_m]$$

where each $c_i \in S$. Crucially, such a trace indicates that we have seen *every* instance of each c_i during the traced execution.

As an example, consider the call trace $[C, F, F]$ with respect to the CFG from fig. 1. Assume that $S = \{C, J, F\}$ (i.e., all call sites). This failure constraint cannot possibly be satisfied in any context-sensitive paths through G.

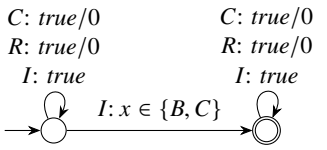


Figure 10. Ambiguous coverage data entry "$\{B, C\}$: *true*" as an automaton

Automata To encode the above constraint, we form an automaton that accepts precisely the observed sequence of calls, with no other instances of symbols from S. Figure 9 shows the s-VPA encoding for $[C, F, F]$. Note that this call trace constraint is regular, since it does not encode the matching relations between calls and returns, but, rather, simply their order.

Unreliable Trace Languages Call trace constraints are *reliable* traces. Unfortunately, as proven in Ohmann et al. [32, section 4.2], we cannot precisely encode reliable traces and use any known polynomial-time solver to obtain query answers. Instead, we can encode the "$[C, F, F]$" constraint as the unreliable trace language $\langle C, F, F \rangle$ with some loss of precision. Specifically, this constraint no longer enforces that we have seen all instances of the call sites from S.

5.4 Ambiguous Observations

Failure report data may be ambiguous. For example, when a program logs some of its activity, an analysis tool may be unable to determine precisely which output statement printed the message. Here, *exactly one* of the output statements executed on the failing run. In fact, all of the above constraints have generalized variants, where each observation is a *set* of symbols from Σ, rather than a single symbol. This subsection considers the specific case of ambiguous coverage data.

Ambiguous binarized statement coverage data consists of a set of independent, Boolean-valued observations of statement groups. For example, for $S_1, S_2 \subseteq N$,

$$\{S_1\text{: } \textit{true}, S_2\text{: } \textit{false}\}$$

indicates that at least one statement from S_1 executed during the failing run, and at least one statement from S_2 did not execute. The failure report from fig. 1 contains a log message that was written by either statement B or statement C. This corresponds to the ambiguous coverage observation "$\{B, C\}$: *true*".

Automata For "S_1: *true*," all accepted strings must contain at least one occurrence of some $s \in S_1$. Figure 10 shows the s-VPA to enforce the constraint "$\{B, C\}$: *true*."

The encoding for "S_2: *false*" is less elegant, and potentially requires an exponential number of states in the size of S_2. We create one unambiguous *true* coverage automaton for each $s \in S_2$ per section 5.2. Then, using this set of automata, A,

Table 2. Evaluated applications

			Mean Count Across Variants			
Application	Type	Variants	LoC	Functions	Basic Blocks	SCCs
tcas	Siemens	41	173	12	163	163
schedule2	Siemens	9	373	25	268	228
schedule	Siemens	9	413	23	229	169
replace	Siemens	31	563	27	437	245
tot_info	Siemens	23	564	18	231	129
print_tokens2	Siemens	10	568	27	429	395
print_tokens	Siemens	7	727	25	385	318
ccrypt	Linux utility	1	5,280	116	1,677	1,409
gzip	Linux utility	20	8,114	135	3,704	2,411
space	ADL interpreter	34	9,563	158	3,895	3,389
sed	Linux utility	31	14,314	219	6,612	3,481
flex	Linux utility	53	14,946	184	6,408	3,992
grep	Linux utility	19	15,460	177	7,121	2,886
gcc	C compiler	1	222,196	2,267	142,121	51,668

the desired encoding is

$$\neg\Big(\bigcap_{a \in A} a\Big)$$

Unreliable Trace Languages Recall that unreliable trace languages (defined in section 4.1) do not allow disjunction. In fact, ambiguous constraints (such as ambiguous coverage data) cannot be expressed, even with loss of precision, as unreliable trace languages. Put another way, since unreliable trace languages cannot support character classes in constraints, the above coverage constraints can only be expressed as Σ^*.

6. Evaluation

Our empirical evaluation assesses the precision and scalability of each of our solving techniques. Specifically, we compared the time it takes our system to answer queries when encoding the CFG and all failure constraints as s-VPA, each of these elements as FSA, and of answering unreliable trace language queries over the context-insensitive CFG. We also compared the precision of query answers when encoding constraints using each of the formulations. (Recall that a solver is more precise if it answers *Impossible* to more user queries.)

We compiled programs using Clang/LLVM 3.5 [23], augmented by csi-cc [28] for optimized coverage instrumentation. Our s-VPA solver builds upon D'Antoni's symbolic automata library [10], while our FSA solver uses OpenFst [1]. The analysis infrastructure and our solver for unreliable trace languages (UTL) from section 4.2 consist of 5,016 lines of Python code. We ran all experiments on a quad-core Intel Core i5-3450 CPU clocked at 3.10 GHz with 32 GB of RAM and running Red Hat Enterprise Linux 6.7.

We tested our solvers on a large selection of buggy C benchmark programs listed in table 2. ccrypt and gcc are real, released programs; the remainder are from the Software-artifact Infrastructure Repository (SIR) [12, 38] and were used in prior work [28, 31]. The small Siemens applications contain seeded faults, as do flex, grep, and gzip. sed contains a mix of seeded and real faults, while ccrypt, gcc, and space all contain real faults. Some applications have multiple versions and multiple faults which can be enabled independently; the "Variants" column of table 2 shows the total number of builds across all versions and faults for each application. In all experiments, we only enabled one fault at a time. The programs' test suites triggered a variety of failures. Some produced core dumps due to invalid memory accesses. For those that failed by producing invalid output, we forced the program to abort and dump core at the first byte of incorrect output. Table 2 also provides information about the CFGs for each application: the number of functions, basic blocks, and strongly-connected components (SCCs). We gathered these statistics prior to incorporating any failure constraints which cause us to split basic blocks or SCCs.

Our evaluation followed the experimental procedure of Ohmann et al. [30], whose work provided the basis of our FSA solver. For each failure, we used our system to recover "best-effort" program coverage based on the failure report by posing two queries for each basic block *b*:

1. May *b* have executed during the failing run?
2. May *b not* have executed during the failing run?

These queries are encoded precisely as statement coverage per section 5.2, where we represent each basic block by the first statement in that block. Recall that each of these queries can return an answer of *Possible* or *Impossible*. Hence, if we obtain a result of *Possible* to both queries for basic block *b*, we say that *b* "Maybe" executed on the failing run. If only one query is *Possible*, we can say definitively "Yes" or "No" to whether *b* executed on the failing run. Thus, the precision of an approach improves if it can more often answer "Yes" or "No" based on the two above queries. Posing two queries

Table 3. Mean number of constraints intersected

		Call Coverage Constraints	
Application	Stack	False	True
tcas	2	8	7
schedule2	2	17	24
schedule	2	13	22
replace	2	30	20
tot_info	2	12	14
print_tokens2	2	27	40
print_tokens	2	29	26
ccrypt	2	164	25
gzip	2	381	23
space	2	390	108
sed	2	546	113
flex	2	590	132
grep	2	448	89
gcc	2	12,649	585

Table 4. Incomplete analyses using stacks only

		Time, Memory		
Application	Variants	s-VPA	FSA	UTL
ccrypt	1	1, 0	0, 0	0, 0
gzip	20	20, 0	0, 0	0, 0
space	34	34, 0	0, 0	0, 0
sed	31	28, 3	0, 0	0, 0
flex	53	50, 3	0, 0	0, 0
grep	19	14, 5	0, 0	0, 0
gcc	1	0, 1	1, 0	1, 0

per basic block is a thorough test of our system, and also simulates the needs of an IDE when color-coding code based on its execution status [29].

6.1 Crashing Stack Trace

Our first set of experiments used failure reports containing only crashing program stacks. We randomly selected one failing run from the test suite of each program for each version and fault. Thus, the number of analysis runs for each solver is identical to the number of variants listed in table 2, and each failure involves intersecting the query language with exactly 2 other languages: the language of the program's CFG and the failing stack trace. This is shown in the "Stack" column of table 3. We limited solvers to 3 hours of running time and 30 GB of memory. We tracked whether each analysis run completed all queries within the time limit, ran out of time, or ran out of memory. Table 4 shows these results, omitting rows for applications that solved all queries for all solvers. For each solver, we report the comma-separated pair of the number of analysis runs that ran out of time and ran out of memory.

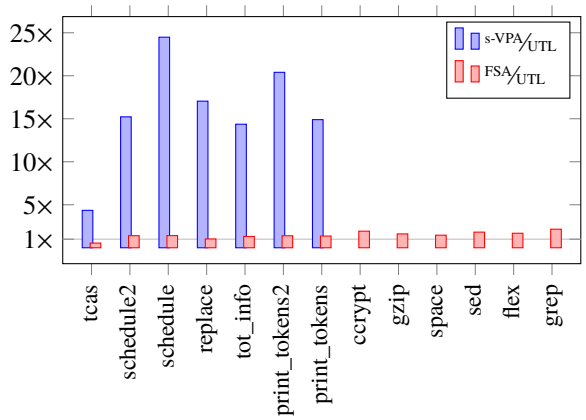


Figure 11. UTL-relative analysis time using stacks only

The s-VPA solver times out for all larger applications. This solver completes at least one query for many of these applications, but note that checking emptiness is significantly more complex for an s-VPA than for an FSA (where it is simply unconstrained state reachability). Hence, we find that **precisely answering queries with s-VPA does not scale to larger programs.** gcc's CFG has over 140,000 basic blocks, resulting in over 280,000 queries. No solver completed all gcc queries within the 3-hour time limit. Both the FSA solver and the unreliable trace languages (UTL) solver, however, are able to answer queries in this case. The FSA and UTL solvers averaged 7.88 and 5.19 seconds, respectively, to answer each query for those that completed within the time limit.

Next we compared analysis times for those analysis runs that multiple solvers completed. Figure 11 shows these results, plotted as analysis time relative to the UTL solver, averaged across each application. While each s-VPA bar summarizes fewer runs than the corresponding FSA bar, note that all runs that completed with the s-VPA solver also completed with the FSA solver. Missing s-VPA bars toward the right side of fig. 11 echo our earlier finding that s-VPAs do not scale to large programs. Even for small programs, the s-VPA approach takes much more time, with slow-downs up to 25× for schedule (where all solvers completed all failures, per table 4). **The FSA solver is slightly slower than the UTL solver on all non-trivial applications, even when incorporating only the crashing stack as the lone failure constraint.** The largest slow-down here is 2.2× for grep.

Finally, we measured the improvement in precision from using the more expressive s-VPA solver. Per section 5.1, s-VPAs allow a more precise encoding of the conditions producing a failing stack than the regular language encoding. Figure 12 shows the mean percentage of basic blocks that each solver definitively categorized as "Yes" or "No" for each failing run. We only show results for the s-VPA and UTL solvers here; FSA results are always identical to those of the UTL solver, as the language encoding of the CFG and crashing stack are identical for these solvers. Two patterns

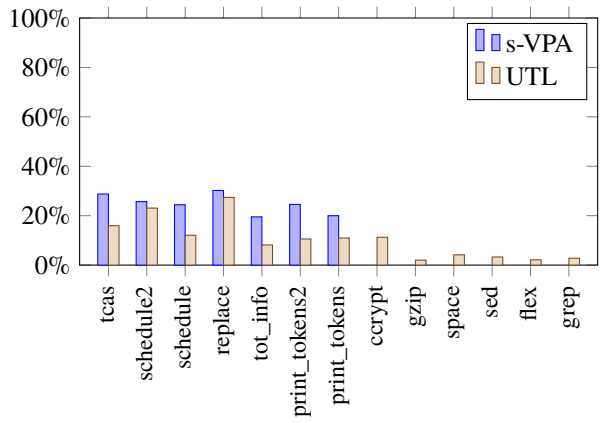


Figure 12. Basic blocks definitively categorized as "Yes" or "No" using stacks only

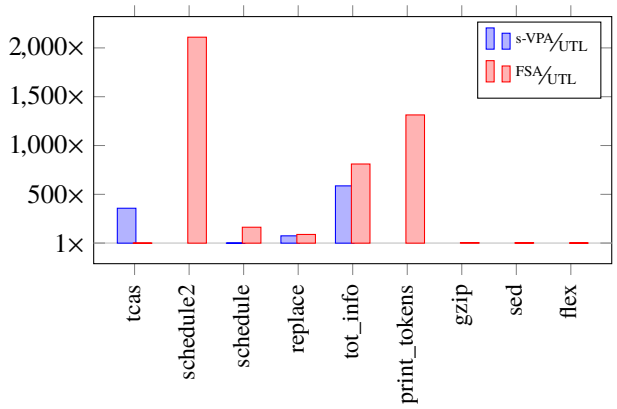


Figure 13. UTL-relative analysis time using stacks and call coverage

Table 5. Incomplete analyses using stacks and call coverage

		Time, Memory		
Application	Variants	s-VPA	FSA	UTL
schedule2	9	0, 9	2, 3	0, 0
schedule	9	0, 8	0, 5	0, 0
replace	31	0, 28	0, 0	0, 0
tot_info	23	0, 17	0, 0	0, 0
print_tokens2	10	0, 10	0, 10	0, 0
print_tokens	7	0, 7	1, 4	0, 0
ccrypt	1	0, 1	0, 1	0, 0
gzip	20	0, 20	4, 7	0, 0
space	34	0, 34	0, 34	0, 0
sed	31	3, 28	2, 26	0, 0
flex	53	18, 35	0, 52	0, 0
grep	19	0, 19	1, 18	0, 0
gcc	1	0, 1	1, 0	1, 0

are clear. First, **for those failures that can tolerate its cost, the s-VPA solver gives substantially more precise results.** For example, for tot_info, classified basic blocks grow from 8.1% to 19.5%: a 2.4× improvement. Second, **information about the crashing stack alone is not enough to reduce execution ambiguity significantly for the larger programs.** In the following section, we address this by considering more detailed failure reports.

6.2 Crashing Stack And Call Coverage

For our second set of experiments, we used failure reports containing the crashing program stack and program coverage data gathered at call sites. Prior approaches [27, 28] have used call-site coverage for program analysis. We configured csi-cc to gather coverage at call sites, per the dominator-based optimization of Ohmann et al. [31]. Thus, some optimized fraction of each program's call sites were actually instrumented. We encoded the resulting coverage data for each failing run as described in section 5.2. The "Call Coverage Constraints" columns of table 3 show the number of constraints resulting from the *true* and *false* coverage entries for each failing run, averaged by application. Thus, the sum of these two columns indicates the total number of call sites instrumented, averaged by application. As in section 6.1, we ran one failing run per (application, version, fault) triple, and limited solvers to 3 hours of running time and 30 GB of memory.

Table 5 gives the number of analysis runs that ran out of time or memory for each application. Here, both the s-VPA and FSA solvers ran out of either time or memory analyzing most failure reports, even for the smaller Siemens applications. Memory issues are especially prevalent, as intersecting large numbers of constraints results in a worst-case exponential increase in the size of automata. **The UTL solver was able to solve every single failure except for gcc,** where it averaged 8.12 seconds to answer each query that completed within the time limit. The gcc failure had 585 *true* coverage entries in its failure report (per table 3), so an automata-based solver would need to compute an intersection over 587 automata (including the CFG and crashing stack) to answer even a single query. In our experiments, the FSA solver exceeded the 3-hour time limit while performing intersections, but would certainly have exceeded the 30 GB memory limit if given more time.

Next we again examined the total analysis time for those runs that completed with each solver. Figure 13 plots these results, again relative to UTL solve time. Note that s-VPA and FSA bars cannot be directly compared, because the s-VPA solver did not solve all of the failures solved by the FSA solver. (This is especially relevant for replace and tot_info results, where the s-VPA solver was slower, but timed out on longer-running examples.) Overall, these results indicate that **the UTL solver is dramatically faster.** The only exceptions are for the small number of runs that completed for the FSA solver in gzip, sed, and flex. Here, the particular failures occurred very early in each application, so the majority of the CFG was

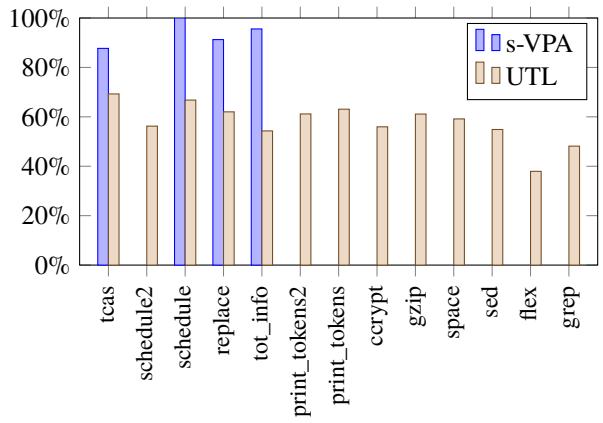


Figure 14. Basic blocks definitively categorized as "Yes" or "No" using stacks and call coverage

quickly marked as "No" by each solver. For all more complex failures that completed with both solvers (including those for the smaller test subjects), UTL outperforms FSA, often by several orders of magnitude. These results are unsurprising, given the number of timeouts recorded in table 5.

Figure 14 shows the number of basic blocks marked as either "Yes" or "No" execution status for failure reports containing the crash stack plus coverage data. Again, all failure report data is recorded with equal precision for FSA and UTL, so we display only the latter. The plot affirms that **we gain significant precision by encoding constraints to respect context sensitivity (via the s-VPA solver).** However, recall that we were able to solve very few failure report analyses with this solver. Another important result is that **precision improves significantly, even for imprecise solvers, when we incorporate more detailed failure reports.** For example the s-VPA solver disambiguates the execution of 20% of the basic blocks in print_tokens failures using only stack data (fig. 12). Using call coverage data, the UTL solver disambiguates 63% of basic blocks (fig. 14) despite being unable to respect calling-context sensitivity during analysis.

6.3 Discussion and Practical Implications

Note that we set a 3-hour timeout for solvers to complete *all* queries for a given failure (2 per basic block). Nearly all subjects in tables 4 and 5 that fail due to timeout complete at least one query. The only exceptions are in table 5, where all s-VPA runs on sed and flex, and FSA on gcc, reach the 3-hour time limit, but would have exceeded the 30 GB memory limit if allowed to complete more intersections. Per section 6.1, both the FSA and UTL solvers handle queries over gcc (the largest subject) within seconds using a stack trace. Per section 6.2, the UTL solver can even answer queries over gcc with the full failure report (consisting of over 13,000 constraints) in seconds. Similarly, the s-VPA solver answers queries for most failures on the larger applications using only a stack trace. The larger SIR applications (sed, flex, and grep) averaged 2 minutes per query. For *overnight* batch analyses (e.g., extracting coverage data for an entire program), 3 hours is quite reasonable, and vastly outperforms a developer trying to reconstruct this information manually. Our solvers are also fast enough to answer individual, interactive queries, such as the examples from section 1.

Overall, our results indicate that we can answer many control-flow queries based on failure data in reasonable time, and point to different use cases for our three different solvers. The UTL solver answers queries remarkably efficiently, even when we include very large failure reports (with over 13,000 failure constraints for gcc). However, it sacrifices some expressiveness in the types of failure report elements that it can precisely encode (see section 5). Thus, we consider the UTL solver to be an excellent choice for answering *interactive* queries during a debugging session, where speed trumps precision. In contrast, the s-VPA solver can answer queries more precisely in many cases, but requires more time to do so, and struggles with memory constraints on very large failure reports. Thus, we consider the s-VPA solver to be well-matched to longer, overnight batch analyses. The FSA solver is a compromise between these extremes. As with s-VPA, the FSA solver needs to constrain the size of failure reports to avoid memory issues in its automata intersections, but allows more expressiveness than the UTL solver while often maintaining similar efficiency for answering interactive queries.

In this work, we only evaluated program coverage queries run as batch analyses. While this evaluation provides insight into the scalability of our solvers to large numbers of failure constraints, future work could more closely investigate other scenarios. Our results suggest that failure reports with a limited number of constraints are good targets for improving precision by using the s-VPA and FSA solvers. This means that detailed failure reports may need to intelligently drop some constraints in order to make use of these solvers. Gathering somewhat more expensive failure data—e.g., call *traces* rather than call *coverage* (see section 5.3)—often results in a smaller number of failure constraints. We expect this would improve the performance and precision of the FSA and s-VPA solvers because the complexity of checking automata intersection emptiness depends on the *number* of constraints. Our results using only a crashing stack trace suggest that reducing the number of failure constraints is important in practice.

6.4 Validation and Threats to Validity

We have attempted to minimize threats to the external and internal validity of our results. While it is obviously impossible for us to test our approach on all possible programs and failures, we selected applications that vary widely in size and functionality. To control for other factors (outside of the choice of solver and failure data) which could impact our memory usage and execution time results, we ran all experiments on a single machine under minimal load.

We also validated our results in two ways. First, each of our analyses should safely approximate complete, directly-observed coverage. We spot-checked analysis results against full coverage data for a selection of failing runs. All analyses safely approximated the complete information: in every case that we checked, if an analysis reported "Yes" for a basic block, then that block did actually run; similarly, blocks reported as "No" did not run. Our second approach to validation compares the solvers to each other, one basic block at a time. Note that our three solvers have theoretical relationships that should be reflected in our results. Because all of our queries and coverage constraints are precisely definable in UTL, and, per section 5.1, our encoding of the crashing stack is equivalent for FSA and UTL, the results of FSA and UTL should be identical in our experiments. The s-VPA solver can encode any s-VPA-definable constraint (subsuming all definable constraints for FSA and UTL), more precisely encodes the crashing stack (see section 5.1), and ensures well-matched calls and returns for accepted strings; hence, its result should never be less precise than FSA and UTL. For the subset of blocks where multiple solvers' results are available to compare, FSA and UTL always agree while s-VPA either agrees or is strictly more precise. That is, s-VPA never answers "Maybe" where FSA or UTL answered "Yes" or "No." This perfectly matches the expected theoretical relationships among the three solvers.

7. Related Work

LaToza and Myers [22] find that developers commonly ask reachability questions while debugging. This supports the usefulness of our technique, as many of the examples they consider are control-flow questions that we support.

Explorer [13] uses demand-driven pointer analysis to allow users to pose interprocedural control-flow queries over a program's call graph. Our system allows queries at the statement (rather than procedure) level, and answers queries with respect to all runs *consistent with a given dynamic failure report*, rather than (statically) all runs *whatsoever*.

Some prior approaches rely on complete tracing of executions [19] or environment interactions [7]. Intensive data collection of this sort allows for rich functionality, such as deterministic replay or detailed causal inquiries. However, the resulting overheads may be prohibitive for deployed software. We explicitly trade off detail for efficiency, and demonstrate that much of value can be learned even from quite impoverished dynamic feedback. Manevich et al. [25] use backward data-flow analysis to reproduce failing executions based on only a failure location and typestate information regarding the failure. While efficient, this approach is limited to solving specific typestate problems with simple types.

Many prior approaches [5, 6, 8, 16, 37, 45] use symbolic execution to replay failures based on varying styles of failure data. Matching failures via symbolic execution is expensive, and undecidable in general. We answer queries about *any* run matching failure data, while replay produces *one specific* run that may or may not completely match the traced failure.

Our weakening of s-VPA constraints to unreliable trace languages resembles work by Place et al. [36] for automatically separating regular languages by piecewise-testable languages. UTL is a strict subset of the piecewise-testable languages; similar techniques may apply here.

Gabow et al. [14] and Lal et al. [21] find paths through CFGs that reach desired nodes. Our use of CFG condensation graphs is similar to that of Gabow et al., but we generalize to larger classes of constraints and cast the problem in data-flow terms. Whereas Lal et al. allow incorporation of other *data*-flow constraints, we dramatically broaden the class of *control*-flow constraints, and offer time/precision trade-offs.

While our approach is complex, it has notable advantages over simpler alternatives. Most prior approaches are limited to specific categories of failure data [5, 8, 16, 28]. In contrast, our approach can immediately take advantage of any s-VPA-definable constraint. A simpler alternative approach might simply mark compulsory statements based on failure data and the user's query, and then use a backward reachability analysis to find a path satisfying the compulsory nodes, in the style of Gabow et al. [14]. If we used only coverage information, this approach would suffice for our problem, and is essentially equivalent to UTL emptiness from section 4.2 (except that we explore paths forward rather than backward). However, the UTL solver allows a broader class of constraints, which is why it maintains vectors of constraints, rather than simply marking compulsory nodes along the path. Our full suite of approaches is much more general, and allows an expansive family of constraints and queries.

8. Conclusion

Failure reports from production runs generally paint an incomplete picture of the failing execution. We present a system that allows a developer to ask *Possible*/*Impossible* questions about the control flow of program runs consistent with observed failure data. We also introduce a new class of subregular languages, the unreliable trace languages, that are uniquely suited to answering these questions in polynomial time. We propose three separate solvers that present trade-offs in scalability, precision of results, and expressiveness in encoding failure constraints. Experimental evaluation shows that we can answer a broad class of queries remarkably efficiently when encoding all failure constraints and user queries as unreliable trace languages.

Acknowledgments

This research was supported in part by DoE contract DE-SC0002153 and NSF grants CCF-0953478, CCF-1217582, CCF-1318489, CCF-1320854, and CCF-1420866. Opinions, findings, conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the sponsoring agencies.

References

[1] C. Allauzen, M. Riley, J. Schalkwyk, W. Skut, and M. Mohri. OpenFst: A general and efficient weighted finite-state transducer library. In *CIAA*, 2007. URL http://www.openfst.org.

[2] R. Alur and P. Madhusudan. Adding nesting structure to words. In *Developments in Language Theory, 10th International Conference, DLT 2006, Santa Barbara, CA, USA*, volume 4036 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2006. ISBN 3-540-35428-X. URL http://dx.doi.org/10.1007/11779148_1.

[3] R. Alur, A. Bouajjani, and J. Esparza. Model checking procedural programs. *Handbook of Model Checking. Springer*, 2015.

[4] D. M. Berris, A. Veitch, N. Heintze, E. Anderson, and N. Wang. XRay: A function call tracing system. Technical report, Google Inc., 2016.

[5] Y. Cao, H. Zhang, and S. Ding. SymCrash: selective recording for reproducing crashes. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden*. ACM, 2014. ISBN 978-1-4503-3013-8. URL http://doi.acm.org/10.1145/2642937.2642993.

[6] N. Chen and S. Kim. STAR: stack trace based automatic crash reproduction via symbolic execution. *IEEE Trans. Software Eng.*, 41(2):198–220, 2015. URL http://dx.doi.org/10.1109/TSE.2014.2363469.

[7] J. Clause and A. Orso. A technique for enabling and supporting debugging of field failures. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 261–270. IEEE Computer Society, 2007. ISBN 0-7695-2828-7. URL http://dx.doi.org/10.1109/ICSE.2007.10.

[8] O. Crameri, R. Bianchini, and W. Zwaenepoel. Striking a new balance between program instrumentation and debugging time. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11. ACM, 2011. ISBN 978-1-4503-0634-8. URL http://doi.acm.org/10.1145/1966445.1966464.

[9] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight defect localization for Java. In *ECOOP 2005 - Object-Oriented Programming, 19th European Conference, Glasgow, UK*, volume 3586 of *Lecture Notes in Computer Science*, pages 528–550. Springer, 2005. ISBN 3-540-27992-X. URL http://dx.doi.org/10.1007/11531142_23.

[10] L. D'Antoni. The symbolic automata library. https://github.com/lorisdanto/symbolicautomata, 2016.

[11] L. D'Antoni and R. Alur. Symbolic visibly pushdown automata. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria*, volume 8559 of *Lecture Notes in Computer Science*, pages 209–225. Springer, 2014. ISBN 978-3-319-08866-2. URL http://dx.doi.org/10.1007/978-3-319-08867-9_14.

[12] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10 (4):405–435, 2005. URL http://dx.doi.org/10.1007/s10664-005-3861-2.

[13] Y. Feng, X. Wang, I. Dillig, and C. Lin. Explorer: query-and demand-driven exploration of interprocedural control flow properties. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, Pittsburgh, PA, USA*, pages 520–534. ACM, 2015. ISBN 978-1-4503-3689-5. URL http://doi.acm.org/10.1145/2814270.2814284.

[14] H. N. Gabow, S. N. Maheswari, and L. J. Osterweil. On two problems in the generation of program test paths. *IEEE Trans. Software Eng.*, 2(3):227–231, 1976. URL http://dx.doi.org/10.1109/TSE.1976.233819.

[15] R. Gupta, M. L. Soffa, and J. Howard. Hybrid slicing: integrating dynamic information with static analysis. *ACM Trans. Softw. Eng. Methodol.*, 6(4):370–397, 1997. ISSN 1049-331X. URL http://doi.acm.org/10.1145/261640.261644.

[16] W. Jin and A. Orso. BugRedux: reproducing field failures for in-house debugging. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 474–484. IEEE Press, 2012. ISBN 978-1-4673-1067-3. URL http://dl.acm.org/citation.cfm?id=2337223.2337279.

[17] B. Kasikci, T. Ball, G. Candea, J. Erickson, and M. Musuvathi. Efficient tracing of cold code via bias-free sampling. In *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA*, pages 243–254. USENIX Association, 2014. URL https://www.usenix.org/conference/atc14/technical-sessions/presentation/kasikci.

[18] O. Klíma and L. Polák. Hierarchies of piecewise testable languages. In *Developments in Language Theory, 12th International Conference, DLT 2008, Kyoto, Japan*, volume 5257 of *Lecture Notes in Computer Science*, pages 479–490. Springer, 2008. ISBN 978-3-540-85779-2. URL http://dx.doi.org/10.1007/978-3-540-85780-8_38.

[19] A. J. Ko and B. A. Myers. Extracting and answering why and why not questions about Java program output. *ACM Trans. Softw. Eng. Methodol.*, 20(2), 2010. URL http://doi.acm.org/10.1145/1824760.1824761.

[20] D. Kozen. Lower bounds for natural proof systems. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA*, pages 254–266. IEEE, 1977. URL http://dx.doi.org/10.1109/SFCS.1977.16.

[21] A. Lal, J. Lim, M. Polishchuk, and B. Liblit. BTrace: Path optimization for debugging. Technical Report 1535, University of Wisconsin-Madison, Oct. 2005.

[22] T. D. LaToza and B. A. Myers. Developers ask reachability questions. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa*, pages 185–194. ACM, 2010. ISBN 978-1-60558-719-6. URL http://doi.acm.org/10.1145/1806799.1806829.

[23] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. IEEE, Mar. 2004. URL http://dl.acm.org/citation.cfm?id=977395.977673.

[24] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI 2003*. ACM, 2003. ISBN 1-58113-662-5. URL http://doi.acm.org/10.1145/781131.781148.

[25] R. Manevich, M. Sridharan, S. Adams, M. Das, and Z. Yang. PSE: Explaining program failures via postmortem static analysis. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, SIGSOFT '04/FSE-12, pages 63–72, New York, NY, USA, 2004. ACM. ISBN 1-58113-855-5. URL http://doi.acm.org/10.1145/1029894.1029907.

[26] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997. ISBN 1-55860-320-4.

[27] A. Nishimatsu, M. Jihira, S. Kusumoto, and K. Inoue. Call-mark slicing: an efficient and economical way of reducing slice. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, pages 422–431, New York, NY, USA, 1999. ACM. ISBN 1-58113-074-0. URL http://doi.acm.org/10.1145/302405.302674.

[28] P. Ohmann and B. Liblit. Lightweight control-flow instrumentation and postmortem analysis in support of debugging. In *28th International Conference on Automated Software Engineering (ASE 2013)*. IEEE and ACM, Nov. 2013. URL http://dx.doi.org/10.1109/ASE.2013.6693096.

[29] P. Ohmann and B. Liblit. CSIclipse: presenting crash analysis data to developers. In *Proceedings of the 2015 Workshop on Eclipse Technology eXchange, ETX 2015, Pittsburgh, PA, USA*, pages 7–12. ACM, 2015. ISBN 978-1-4503-3904-9. URL http://doi.acm.org/10.1145/2846650.2846651.

[30] P. Ohmann, D. B. Brown, B. Liblit, and T. W. Reps. Recovering execution data from incomplete observations. In *Proceedings of the 13th International Workshop on Dynamic Analysis, WODA 2015, Pittsburgh, PA, USA*, pages 19–24. ACM, 2015. ISBN 978-1-4503-3909-4. URL http://doi.acm.org/10.1145/2823363.2823368.

[31] P. Ohmann, D. B. Brown, N. Neelakandan, J. Linderoth, and B. Liblit. Optimizing customized program coverage. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore*, pages 27–38. ACM, 2016. ISBN 978-1-4503-3845-5. URL http://doi.acm.org/10.1145/2970276.2970351.

[32] P. Ohmann, A. Brooks, L. D'Antoni, and B. Liblit. Supporting proofs for control-flow recovery from partial failure reports. Technical Report 1845, University of Wisconsin–Madison, Apr. 2017.

[33] A. Orso, D. Liang, M. J. Harrold, and R. Lipton. Gamma system: continuous evolution of software after deployment. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '02, pages 65–69, New York, NY, USA, 2002. ACM. ISBN 1-58113-562-9. URL http://doi.acm.org/10.1145/566172.566182.

[34] C. Pavlopoulou and M. Young. Residual test coverage monitoring. In *Proceedings of the 1999 International Conference on Software Engineering, ICSE' 99, Los Angeles, CA, USA*, pages 277–284. ACM, 1999. ISBN 1-58113-074-0. URL http://portal.acm.org/citation.cfm?id=302405.302637.

[35] J.-E. Pin. Syntactic semigroups. In *Handbook of formal languages*, pages 679–746. Springer, 1997.

[36] T. Place, L. van Rooijen, and M. Zeitoun. Separating regular languages by piecewise testable and unambiguous languages. In *Mathematical Foundations of Computer Science - 38th International Symposium, MFCS 2013, Klosterneuburg, Austria*, volume 8087 of *Lecture Notes in Computer Science*, pages 729–740. Springer, 2013. ISBN 978-3-642-40312-5. URL http://dx.doi.org/10.1007/978-3-642-40313-2_64.

[37] J. Rößler, A. Zeller, G. Fraser, C. Zamfir, and G. Candea. Reconstructing core dumps. In *ICST '13: Proceedings of the Sixth IEEE International Conference on Software Testing, Verification and Validation*. IEEE, Mar. 2013. URL http://dx.doi.org/10.1109/ICST.2013.18.

[38] G. Rothermel, S. Elbaum, A. Kinneer, and H. Do. Software–artifact infrastructure repository, Sept. 2006. URL http://sir.unl.edu/portal/.

[39] A. Rountev, S. Kagan, and M. Gibas. Static and dynamic analysis of call chains in Java. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2004*, pages 1–11. ACM, 2004. ISBN 1-58113-820-2. URL http://doi.acm.org/10.1145/1007512.1007514.

[40] I. Simon. Piecewise testable events. In *Automata Theory and Formal Languages, 2nd GI Conference, Kaiserslautern*, volume 33 of *Lecture Notes in Computer Science*, pages 214–222. Springer, 1975. ISBN 3-540-07407-4. URL http://dx.doi.org/10.1007/3-540-07407-4_23.

[41] M. M. Tikir and J. K. Hollingsworth. Efficient online computation of statement coverage. *Journal of Systems and Software*, 78(2):146–165, 2005.

[42] M. Veanes. Applications of symbolic finite automata. In *Implementation and Application of Automata - 18th International Conference, CIAA 2013, Halifax, NS, Canada*, volume 7982 of *Lecture Notes in Computer Science*, pages 16–23. Springer, 2013. ISBN 978-3-642-39273-3. URL http://dx.doi.org/10.1007/978-3-642-39274-0_3.

[43] M. Veanes, P. de Halleux, and N. Tillmann. Rex: Symbolic regular expression explorer. In *3rd International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France*, pages 498–507. IEEE, 2010. ISBN 978-0-7695-3990-4. URL http://dx.doi.org/10.1109/ICST.2010.15.

[44] R. Wu, X. Xiao, S. Cheung, H. Zhang, and C. Zhang. Casper: an efficient approach to call trace collection. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016*, pages 678–690. ACM, 2016. ISBN 978-1-4503-3549-2. URL http://doi.acm.org/10.1145/2837614.2837619.

[45] C. Zamfir and G. Candea. Execution synthesis: a technique for automated software debugging. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 321–334. ACM, 2010. ISBN 978-1-60558-577-2. URL http://doi.acm.org/10.1145/1755913.1755946.