Patches as Better Bug Reports

Westley Weimer
University of Virginia
weimer@virginia.edu

Abstract

Tools and analyses that find bugs in software are becoming increasingly prevalent. However, even after the potential false alarms raised by such tools are dealt with, many real reported errors may go unfixed. In such cases the programmers have judged the benefit of fixing the bug to be less than the time cost of understanding and fixing it.

The true utility of a bug-finding tool lies not in the number of bugs it finds but in the number of bugs it causes to be fixed.

Analyses that find safety-policy violations typically give error reports as annotated backtraces or counterexamples. We propose that bug reports additionally contain a specially-constructed patch describing an example way in which the program could be modified to avoid the reported policy violation. Programmers viewing the analysis output can use such patches as guides, starting points, or as an additional way of understanding what went wrong.

We present an algorithm for automatically constructing such patches given model-checking and policy information typically already produced by most such analyses. We are not aware of any previous automatic techniques for generating patches in response to safety policy violations. Our patches can suggest additional code not present in the original program, and can thus help to explain bugs related to missing program elements. In addition, our patches do not introduce any new violations of the given safety policy.

To evaluate our method we performed a software engineering experiment, applying our algorithm to over 70 bug reports produced by two off-the-shelf bug-finding tools running on large Java programs. Bug reports also accompanied by patches were three times as likely to be addressed as standard bug reports.

This work represents an early step toward developing new ways to report bugs and to make it easier for programmers to fix them. Even a minor increase in our ability to fix bugs would be a great increase for the quality of software.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—Debugging aids; I.2.2 [*Automatic Programming*]: Program Modification

General Terms Algorithms, Experimentation, Human Factors, Languages, Verification

Keywords bug, bug report, error, patch, counterexample, explanation, localization

1. Introduction

Tools and analyses that find bugs in software automatically are becoming increasingly prevalent. Such analyses usually report false positives, and wading through spurious error reports is one cost of using such tools. Even if false positives are controlled (e.g., by ranking or filtering [33]), many real errors go unfixed. Often the original programmer is unavailable, the available programmer does not understand the bug report or the specification, or the bug is viewed as rarely occurring. In such cases the time cost of understanding and fixing the bug is assumed to outweigh the benefit of fixing it.

Analyses that find bugs typically give error reports as annotated counterexamples or backtraces (i.e., feasible paths through the program that demonstrate the bug). Unfortunately, such backtraces are often long or difficult to interpret [6, 26, 28]. We propose that bug reports additionally contain a specially-constructed candidate patch describing an example way in which the program could be modified to avoid the reported policy violation without introducing any new bugs with respect to that policy. Our patches may either highlight the cause of the bug or focus on a local symptom; in either case the information is useful. Programmers viewing the analysis output can use such patches as guides, starting points, or as an additional way of understanding what went wrong where.

We present a novel algorithm for automatically constructing such patches given model-checking and policy information typically already produced or required by the analysis. The algorithm builds on nearest accepting strings [36] and path predicates [31]. We then apply our algorithm to bug reports produced by two off-the-shelf bug-finding tools [30, 43]. Our experiments indicate that bug reports also accompanied by our patches are more likely to be addressed than standard bug reports.

In some commercial environments every bug report from an official tool may be addressed one way or another. For example, some groups at Microsoft require that all `PREfast` or `ESP` [16] warnings be dealt with before a code change can be committed. We believe that in such a scenario our algorithm would reduce the time programmers spend addressing all of the bugs reports (rather than increasing the number of bug reports deemed worth spending time on).

There are two main contributions of this work:

- A new algorithm for constructing candidate patches from the counterexamples produced by bugfinding tools. The patches can suggest inserting code not present in the original program. This is the first algorithm we are aware of that produces patches from bug reports.
- A demonstration that our algorithm increases the usefulness of off-the-shelf bug-finding tools that find defects in large programs. We present experimental evidence to show that including such patches makes bug reports more likely to be addressed. We conclude that patches should be included in bug reports in practice.

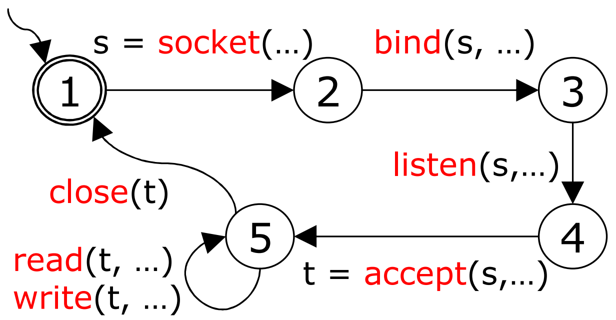

Figure 1. A simplified socket safety policy.

The true utility of a bug-finding tool lies not in the number of bugs it finds but in the number of bugs it causes to be fixed or in the amount of effort it saves on fixing those bugs. Our algorithm can be thought of as a strong "peephole optimizer" for a large class of bug-finding tools: it is an automatic post-processing step that dramatically increases their utility.

The rest of this paper is organized as follows. Section 2 gives a broad overview of the sorts of bug reports we will consider and the patches we would like to generate. In Section 3 we describe existing approaches to reporting bugs and highlight weaknesses in reporting bugs with backtraces. In Section 4 we present our algorithm for computing patches from error traces, control-flow graphs and safety policies. Section 5 presents experimental evidence to support our claim that patches make it more likely that bug reports will be addressed. In Section 6 we discuss related work in error explanation; in general our approach is complementary to, and can be used in conjunction with, such approaches.

2. Motivating Examples

An example simplified policy for the correct use of BSD-style server sockets is given in Figure 1. The policy is presented as a deterministic finite automaton [41]. The nodes represent temporal states for socket objects and the program's use of the socket API. The edges represent important events related to that API. Such a description is usually called a *safety policy*. Each socket instance starts in the start state must end up in an accepting state. All unlisted transitions (e.g., calling `read` from the start state) violate the policy.

Such a formulation is typically called *typestate:* each instance of type `socket` is also given a *dynamically-varying* association with a state in the policy state machine. In the Vault programming language [18] the compiler tracks the typestate of important objects to ensure that resources are used correctly. As another example, Shankar at al. [40] use a two-state policy (tainted and untainted) to track format strings and find security vulnerabilities in C programs.

Consider this buggy example pseudocode:

```
01: socket();   // state 1 -> state 2
02: bind();     // state 2 -> state 3
03: listen();   // state 3 -> state 4
04: read();
05: close();
```

The comments show the associated state transitions in the policy in Figure 1. A safety policy checker will typically report an error on line 4 (it is illegal to call `read` from state 3) and include a backtrace listing lines 1 through 3. We wish to help explain the error by additionally suggesting code like this:

```
01: socket();
02: bind();
03: listen();
04: accept();   /* should be inserted */
05: read();
06: close();
```

Note that we are not looking for the "smallest" program that adheres to the policy: removing all of the lines related to sockets yields a compliant program but is not helpful. On the other hand, we do not want to include anything unnecessary in our candidate patch. For example, we do not want a candidate patch that adds a new call to `"write"` between the calls to `"read"` and `"close"`.

Beyond the simple straight-line case, programs often violate safety policies along some paths while respecting them along others. Consider this example:

```
01: ... // socket in state 4
02: if (p) {
03:   accept();
04: }
05: ... // no change to p
06: close();
```

A safety policy checker will report an error on line 6 and a backtrace including lines 2 and 5 (skipping the `if`). We wish to suggest this code:

```
01: if (p) {
02:   accept();
03: }
04: ... // no change to p
05: if (p) {    /* should be inserted */
06:   close();
07: }
```

An alternative candidate patch might involve removing the original `"if (p)"` conditional. In general we cannot know which is correct without domain- or program-specific knowledge. Our patch-producing algorithm will make use of a distance metric that encodes our intuitions and heuristics about likely fixes (e.g., to favor insertions over deletions).

As a final, simple motivating example, Figure 2 shows source code from HSQLDB, an open-source Java SQL database engine. On line 2712, `os.write(...)` can raise an exception, causing `os` to be leaked. This bug was found by a tool and the patch shown was created automatically using the algorithm described in the Section 4; from our perspective the patch was created by

```
2708: URLConnection c = url.openConnection();
2709: c.setDoOutput(true);
2710: OutputStream os = c.getOutputStream();
2711:
2712: os.write(p.getBytes(ENCODING));
2713: os.close();
2714: c.connect();
```

```
      c.setDoOutput(true);
      OutputStream os = c.getOutputStream();
 +    try {
        os.write(p.getBytes(ENCODING));
 -    os.close();
 +    } finally {
 +      os.close();
 +    }
      c.connect();
```

Figure 2. Code from HSQLDB's jdbcConnection.executeHTTP() method and an explanatory patch produced by our algorithm.

inserting a call to `close` along that exceptional path. The patch was included in a bug report as part of the experiment described in Section 5. The examples in this section have been simple for expository purposes, but our algorithm works on more complicated errors in production code. For example, another patch we produce for HSQLDB involves making 34 changes over a span of 216 lines.

3. Finding and Reporting Bugs

Many software model-checkers and bug-finders have been developed in recent years. In general these tools all take as input a *program* (typically by interpreting the source code or some other representation of the control-flow graph) and a *safety policy* (typically a finite state machine that codifies something that might go wrong: null pointer dereferences [22, 30], resource leaks [43], locking or concurrency errors [12], API violations [7, 16, 29], security vulnerabilities [10, 40], high-level invariants [23], etc.).

Simple safety policies are widely available for many domains [7, 16, 22, 30] and numerous projects exists to mine or infer them automatically [3, 4, 43, 44, 45]. In general finding the right safety policy is a difficult problem in its own right; for this work we assume that a relevant safey policy is already available.

The output of such a bug-finding tool is a set of candidate *error reports* (generally including both false positives and real errors). At minimum an error report lists a program location and refers to the part of the safety policy that is violated there. In general the report also includes a *backtrace* or *counterexample* (a feasible path through the program that ends at the error location, possibly annotated with data values along the way) [11].

While different analyses have different precision/scalability tradeoffs, the usual interaction with such tools involves checking a program against an off-the-shelf safety policy and then sifting through the resulting list of error reports. False positives can be filtered out or assigned special rankings, but ultimately reports must be inspected manually. Counterexample backtraces often provide as much information as would be available when debugging a program crash at the error location. Despite this, traces can still be confusing to programmers who are not used to static analyses, are unfamiliar with the safety policy, are unfamiliar with the environment and fault models used by the tool, or are unfamiliar with the code under consideration (often the original developer is no longer available). Unfortunately, backtraces are not a panacea: "dealing with an error is often an onerous task, even with a detailed failing run in hand." [26] It is not always easy to reason about a bug using a backtrace: "there is significant room for improving users' experiences ... an error trace can be very lengthy and only indicates the symptom ... users may have to spend considerable time inspecting an error trace to understand the cause." [6] In general, "even a detailed trace of how a system violates a specification may not provide enough information to easily understand (much less remedy) the problem." [28]

While counterexample backtraces are good, developers could benefit from additional information in the the form of a *candidate patch*. Patches make it easier to understand the bug report and thus reduce the time cost of fixing the bug, making it more likely that the bug will be addressed. Such patches would not be applied blindly, but would show how to change the code to satisfy the tool: such positive examples implicitly explain the relevant parts of the safety policy (e.g., what events are important) and the assumptions made by the tool (e.g., what happens when a system function is invoked, whether exceptional control flow is being considered). In addition, such a patch gives the developer, previously told to "fix this bug," an additional way to approach the problem: "is this patch legal and would it really fix the bug?" Even if this does not add new information, research in psychology suggests that humans possess special cognitive facilities for detecting cheating and rules violations when properly cued [13, 25]. Such considerations are beyond the scope of this work, however, and in this paper we claim that providing additional information in the form of a patch makes it more likely that a bug report will be addressed; specific characterizations of why such patches help are left for future work.

```
1 : GenPatch(policy $P$, metric $M$, cfg $C$, path $v$, source $S$) {
2 :   string $c$ = NearestAcceptingString($P$, $v$, $M$);
3 :   return a minimum patch $d$ produced by {
4 :     foreach cfg $C' \in$ MapToSource($v$, $c$, $C$) {
5 :       source $S'$ = PrettyPrint($C'$);
6 :       $d$ = diff($S, S'$);
7 :     }
8 :   }
9 : }
```

Figure 3. Pseudocode for generating a patch. NearestAcceptingString and MapToSource are described in Section 4.

4. Algorithm

Our basic algorithm for producing candidate patches is intraprocedural and path sensitive. It takes as input a safety policy P, an edit distance metric M, a procedure C, a path v through that procedure that violates the policy, and the program source code S.

The *intuition* for the algorithm is as follows:

1. Make many copies of the safety policy state machine P and link them together such that a string is accepted by the kth copy if that string is a distance k from some string accepted by P.
2. Feed the violating path v to the constructed state machine to obtain insertions and deletions that would transform v into a string accepted by P. These changes correspond to adding or removing function calls to address the bug.
3. Map those insertions and deletions back to the source code. Use path predicates to guard them so that only the violating path is affected.
4. Construct a textual patch and include it with the bug report sent to the developer.

We now describe the algorithm formally. High-level pseudocode for the patch generation algorithm is given in Figure 3.

The violating path involves some number of important events in the policy but is not accepted by the policy. The computational heart of our algorithm considers the string (in the language of policy events) associated with the violating path v and finds a string c in the language generated by that safety policy P that is closest to it according to the distance metric M (line 2 in Figure 3 and Section 4.1). Once the nearest accepting string c has been found we then consider all of the ways to match up its suggested insertions and deletions with the original source S (lines 4-6 in Figure 3 and Section 4.2).

4.1 Nearest Accepting String

We formalize the core of the problem as follows: given a deterministic finite automaton P (the safety policy) that accepts at least one string over the alphabet Σ (important program events), a string v (the violating path's events) in Σ^*, and an edit distance metric $M : \Sigma^* \times \Sigma^* \rightarrow \mathcal{N}$, we must produce a string $c \in L(P)$ (the candidate) such that $M(v, c)$ is a global minimum.[1]

[1] Since the candidate is heuristic, "minimum" is not strictly necessary and "small" would suffice.

The safety policy DFA P is a five-tuple $\langle \Sigma, S, s_0, \delta, F\rangle$ where Σ is the alphabet of policy events, S is the set of states, $s_0 \in S$ is the start state, $\delta : S \times \Sigma \mapsto S$ is the transition function, and $F \subseteq S$ is a set of accepting states.

Our solution is conceptually related to the *bitap* fuzzy string searching algorithm used in `agrep` [36], but we also determine the operations required to convert v to c and accept a more general edit distance metric.

Let M assign insertions a maximum cost $i > 0$ and deletions a maximum cost $d > 0$. Let p be a string accepted by P. Then $M(v, p) \leq |v|d + |p|i$. An upper bound on edit distances we must explore to find c is thus $B = |v|d + |p|i$. We will first look for a string c within distance 1 of $L(P)$. If we do not find it we will look for a c within distance 2 of $L(P)$, and so on. We are guaranteed to terminate at distance B and find $c = p$, but in general we will find a much better c. Optionally, a better search algorithm such as iterative-deepening-A* [32] could be used to take explicit advantage of the previous iteration.

To find if there exists c within distance k of $L(P)$ we construct a non-deterministic finite automaton [41] P_k. We will make P_k such that it accepts v iff v is within k of some string accepted by P; so $v \in L(P_k) \iff \exists c \in L(P).\ M(c, v) \leq k$. Let $P = \langle \Sigma, S, s_0, \delta, F\rangle$. Then $P_k = \langle \Sigma, S', s'_0, \delta', F'\rangle$ with the components defined as follows.

The states in P_k are products $S' = \{\langle s, j\rangle \mid s \in S \ \wedge\ 0 \leq j \leq k\}$. We make one "copy" of the original machine for each possible distance. We will construct the transition function such that reaching $\langle s, j\rangle$ on input x means that there is a string x' with $M(x, x') \leq j$ such that the original DFA P reaches s on input x'. If $s \in F$ is also an accepting state then x is within distance j of a string in $L(P)$. Thus we let $s'_0 = \langle s_0, 0\rangle$ and $F' = \{\langle s, j\rangle \mid s \in F \ \wedge\ 0 \leq j \leq k\}$. If P_k accepts v then $\exists v' \in \Sigma^*.\ v \in L(P) \ \wedge\ M(v, v') \leq k$.

In P_k there are three conceptual kinds of transitions: normal transitions δ_N, deletion transitions δ_D, and insertion transitions δ_I. The final constructed transition function is given by $\delta' = \delta_N \cup \delta_D \cup \delta_I$.

The normal transitions δ_N stay within one "copy" and mimic P:

$$\delta_N = \{\langle\langle s, j\rangle, x, \langle t, j\rangle\rangle \mid \langle s, x, t\rangle \in \delta \ \wedge\ 0 \leq i \leq j\}$$

Deletion transitions δ_D consume input characters and move down a number of "copies" equal to $d(x)$, the edit distance cost of deleting event x (note that $d(x)$ may be undefined for some x, representing events that cannot be deleted by the policy):

$$\delta_D = \{\langle\langle s, j\rangle, x, \langle s, j{+}d\rangle\rangle \mid s \in S \wedge x \in \Sigma \wedge 0 \leq j \leq k{-}d(x)\}$$

Insertion transitions δ_I do not consume input characters (i.e., they are ϵ-transitions) but do move down a number of "copies" equal to $i(x)$, the insert cost for event x:

$$\delta_I = \{\langle\langle s, j\rangle, \epsilon, \langle t, j + i\rangle\rangle \mid \exists x.\langle s, x, t\rangle \in \delta \ \wedge\ 0 \leq i \leq k - i(x)\}$$

An example of this construction with $k = 2$, P accepting exactly "xyz", insert cost 1 and delete cost 2 is shown in Figure 4. To find a string c with $M(v, c) \leq 2$ we see if P_2 accepts v and note the transitions taken if it does. For example, the string "xz" is accepted using edges labeled "x", "(ins y)", and "z". Similarly, the string "zxyz" is accepted with "(del z)", "x", "y", and "z". The string "xx" is not accepted by P_2 because it is distance 4 from the string in $L(P)$.

In the worst case our algorithm will construct each P_B where B is on the order of $v + p$. The generated patch p is bounded by $|\delta|$ or $|S|^2$. Since P_k is a subset of P_{k+1} an efficient implementation can save work by building the P_k's incrementally. The deletion edges in the constructed P_B dominate its size: every node is potentially the

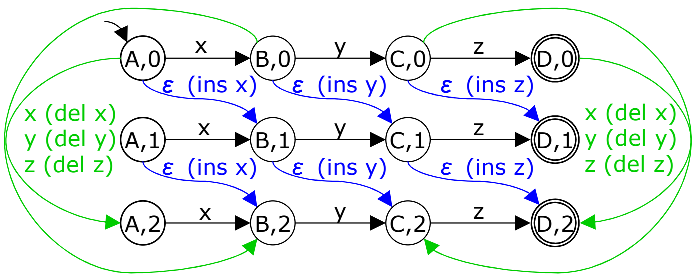

Figure 4. P_k construction for $k = 2$, P accepting exactly "xyz" with uniform delete cost $d = 2$ and insert cost $i = 1$.

source of $|\Sigma|$ deletion transitions and number of nodes is bounded by $|S| \times |B| = |S| \times (|v| + |S|^2)$. So the size of P_B is bounded by $|S| \times |\Sigma| \times (|v| + |S|^2)$. In practice, however, the error path $|v|$ is quite small after path slicing has been applied: Jhala and Majumdar report that the largest of 313 counterexample traces from the `gcc` Spec95 benchmark can be sliced from size 82,695 to size 43 [31].

So the algorithm takes time proportional to the time to check if an NFA of edge size $\mathcal{O}(|\Sigma| \times |S|^3)$ accepts a string of size $|v|$ where S is the state size of the input policy DFA. In practice safety policies are small (four states is a common size) and the largest policy we have seen had under 30 states [7].

4.2 Mapping Back To The Source

The algorithm in Section 4.1 computes a string c in the language of accepted policy events that is close to the events of original buggy path v. It also computes the changes (e.g., insertions and deletions) that should be made to transform the original string to the new string. We must map those abstract changes to changes in the control-flow graph (CFG), which can then be pretty-printed to generate a candidate textual patch. Our soundness condition requires that the proposed changes to the control-flow graph must prevent the policy violation reported on path v and must not introduce any new violations along paths that do not contain v as a prefix.

The major challenge is thus proposing a change that does not interfere with other paths. Minor challenges include matching up argument values and dealing with exception handling. Our algorithm uses `if` statements to guard insertions and deletions with special predicates to ensure that only the failure path v is affected.

In general there will be multiple ways of mapping the changes suggested by the nearest accepting string back to the original code. The two main sources of ambiguity are (1) freedom in inserting missing events (which can often be placed earlier or later along v) and (2) multiple ways to guard paths (with path predicates directly when possible, with path profiling variables when not). We compute a set of possible valid changes; the change that results in the smallest textual patch, as computed by whitespace-ignoring `diff(1)` [21], will be used.

It would also be reasonable to compute the distances over the abstract syntax tree instead of at the textual (`diff`) level. The approaches usually yield identical results: we chose to favor text size in order to generate present small patches to developers.

4.2.1 Path Predicates

We must ensure that our candidate change only influences the behavior of the program along the path v. Consider the following simplified example (as in Section 2), in which `accept` must come between `listen` and `read`.

```
01: if (q)
02:   bind();
```

```
03: // do some work, q is not changed
04: if (q)
05:   listen();
06:
07: if (q)
08:   read();
```

Given the violating path v (visiting lines $1-2-3-4-5-6-7-8$), the Nearest Accepting String algorithm will suggest inserting an `accept` event somewhere after line 5 but before line 8. Any point in that range would be reasonable. Our algorithm works regardless of which point is chosen. We will return to this choice later; for now we will choose to insert at line $L=6$.

Inserting `"accept()"` directly on line 6 will introduce a new error on the path $1-3-4-6$ where `q` is false. Instead, we want to insert `"if (p) accept()"` where `p` is a *path predicate* that is true at points 6 and 7 on the path v but is not true along other paths that reach points 6 and 7. Such path predicates are well-studied for purposes including profiling [5], instruction scheduling [8], program slicing [39] and model-checking [31]. Our algorithm can use any available method for computing a valid path predicate. In the unlikely event that no symbolic path predicate can be found, we can apply the profiling technique of [5] directly and include in our candidate patch all the instrumentation required to dynamically update a newly-introduced variable such that it will have a unique value at line 6 along the path v. We assume that inserting or removing events will not change path predicate values (i.e., in practice we assume that API calls will not change the values of local variables).

In the simple case where the program predicates are side-effect free and are not modified elsewhere (as conservatively determined by standard dataflow analyses [14] and alias analyses [15]) we can re-use them directly. The path predicate `p` is the conjunction of all of the `if`-predicates along v (with a predicate negated if the corresponding `else` branch was taken). Standard boolean minimization techniques are then used to simplify `p`. In the example above, `p` is `"q && q && q"` which reduces to `"q"`. In general, however, more advanced techniques are required. Our implementation follows Robschink and Snelting [39] for computing basic path predicates, Ranganath et al. [38] for computing control dependences in loops, and Jhala and Majumdar [31] for minimizing the resulting predicates. In rare cases where this approach fails we follow Ball and Larus [5] and emit a patch creating a new variable to track the path; this was never required in the experiments in Section 5, however. We do not claim any new results in the generation of path predicates and view it as an advantage that our approach can make use of off-the-shelf solutions to that problem.

If we instead chose to insert at $L = 8$, the candidate patch is simplified. We need only guard the inserted event with conjuncts from the path predicate `q` that are not implied by the path predicate for L (as determined by the path predicate algorithm itself or by an automated theorem prover [19]). Empty or statically true conjuncts need not be checked at run-time. In this example `q` is always established at 8 by the `if (q)` at 7, and thus rather than inserting `"if (true) accept()"` we insert `"accept()"`.

We choose the best insertion point L by computing the candidate patch for each potential point and choosing the point that yields the smallest textual patch. The path predicate information can be shared by all of these trails, so the extra $\mathcal{O}(|v|)$ patch calculations are not a key performance factor. In this example line 8 would be favored over line 6 because the patch at line 8 does not involve inserting a new `if` statement.

When the Nearest Accepting String indicates that an event must be removed, we know exactly what part of the program is being referenced. To avoid interfering with other paths we effectively remove the event from only v by guarding it with the negation of v's path predicate. For example:

```
01: if (q)
02:   open();
03: close();
```

In the path $1-3$ where the socket is not initialized, the Nearest Accepting String may suggest that `close` on line 3 be removed. The path predicate for $1-3$ is `"not q"`, so we remove `close` by guarding it with `"not not q"` (which reduces to `"q"`). A statement that would be guarded by a provably false predicate is instead removed completely.

4.2.2 Dataflow and Arguments

Safety policies, especially typestate policies, are typically presented generically and apply to every instance of a resource. Events are usually function calls related to a special API. When modifying the CFG to insert an event we must either suggest arguments for each inserted function call or explicitly leave holes for the programmer to fill in. We only infer arguments and assignments directly related to the safety policy; all other arguments are left blank (and a candidate patch with such a blank will not yield a valid program). For example:

```
01: int s1 = accept(server, client1, addrlen1);
02: int s2 = accept(server, client2, addrlen2);
03: close(s1);
04: // forgot close for s2
```

If we are inserting a `close` event at line 4, we must either insert `"close(s2)"` or some variant on `"close(/* FIXME */)"`. In particular, we may *not* suggest `"close(s1)"`. The dataflow tracking required to know that `s1` is closed but `s2` is not must already be carried out by the client bug-finding analysis. In general we can obtain that information from the analysis or its normal backtrace bug report. We may also require that the specification be annotated in such a way as to make the dataflow clear (i.e., by stating that the return value from `accept` must be the argument to `close`, as in Figure 1). In the unlikely case that neither source of information is available we rediscover the specification variable, in any, using a simple adaptation of the flow dependence annotation and scenario extraction algorithm from the Strauss specification miner [4], which is designed to infer such dataflow from traces in a safety policy setting. If we are unable to establish that an argument or return value is directly related to the safety policy, we leave it blank.

4.2.3 Exception Handling

As noted in Ranganath et al. [38], existing dependence analyses are often ill-suited to languages with exception handling. Our algorithm can be extend to work on languages with explicit support for exception handling. Consider the following Java code:

```
01: Socket s = new Socket(...);
02: s.read(...); // can raise exceptions
03: s.close();
```

Along the path $1-2-$ *exception* the socket is not closed. We number the statements in the program and extend the language of path predicates with the special predicate $\mathsf{Exc}(i)$, meaning that an exception was raised in statement i.[2] To insert a statement s guarded with the predicate $\mathsf{Exc}(i)$ we catch the exception at i, execute s, and then re-raise that exception.

```
01: Socket s = new Socket(...);
```

[2] We could refine $\mathsf{Exc}(i)$ to include the type of the exception raised, but that information is not necessary given the template `catch` and `finally` code we insert.

```
02: try {
03:   s.read(...); // can raise exceptions
04: catch (Exception e) {
05:   try { s.close(); } finally { throw e; }
06: }
07: s.close();
```

As an optimization, if inserting an event e would make it occur on all paths leading out of a statement i and i dominates each e on those other paths, we can remove the other e events and insert a `finally` block:

```
01: Socket s = new socket(...);
02: try {
03:   s.read(...); // can raise exceptions
04: } finally {
05:   s.close();
06: }
```

We can also make use of an existing `finally` block surrounding i provided that i post-dominates all statements that come before it in that block. Many of the resource leaks reported in Section 5 involved creating or patching such `try-finally` blocks for programs that disposed of resources properly along some, but not all, paths.

The Nearest Accepting String output may demand that we remove an event by guarding it with the negation of $\mathsf{Exc}(i)$. This would typically arise because of an extraneous event in an exception handler, as in:

```
01: Socket s = new socket(...);
02: try {
03:   s.read(...); // can raise exceptions
04:   s.close();
05:   throw new Exception();
06: } catch Exception (e) {
07:   s.close();
08: }
```

The path $1 - 3 - 4 - 5 - \textit{exception} - 7$ contains a double close. The Nearest Accepting String result may suggest deleting `"s.close()"` at line 7, but linguistically there is no simple way to guard that statement with "only if an exception was not raised on line 5". Note that removing line 7 entirely is also wrong, as it introduces an error on the previously-safe path $1-3-\textit{exception}-7$. In such cases our general solution is to insert a profiling variable to track the path taken through the procedure [5] and guard based on that value of that variable. This approach always produces patches that adhere to our notion of safety but may produce awkward patches; this case did not arise in practice in our experiments.

As an optimization we query the NFA P_k produced by the Nearest Accepting String algorithm and compute $L(P_k)$. If another nearest accepting string in $L(P_k)$ produces a simpler patch we use it instead. In the example above, deleting `"s.close()"` from line 4 has the same edit distance cost as deleting it from line 7 but yields a much simper patch (since line 4 can be removed entirely). In the experiments in Section 5 deletions guarded by exception predicates were rare, but when they did occur this optimization always succeeded and path-profiling variables were never necessary.

4.3 Distance Metric

We require that the distance metric M describe an edit distance with specific costs for insertions and deletions. The edit distance is based on the safety policy and heuristics related to the likely correctness of existing code. The edit distance chosen does not affect the *correctness* of the algorithm, merely the chance that it will produce appealing patches.

The edit distance selected must provide a maximum insertion cost and a maximum deletion cost. Specific insertion and deletion costs can vary by event (e.g., if forgetting a particular function call is known to be a common mistake, its insertion cost can be made lower than all other insertion costs) or can be left undefined. An undefined insertion (deletion) cost for an event x means that the Nearest Accepting String algorithm will never suggest inserting (deleting) x. This is useful for certain safety policies. For example, when "do not dereference null pointers" is phrased as a typestate policy on pointers, there is typically a transition from a state representing "this pointer is null" to a state representing "this pointer is valid". The event associated with that edge is "this pointer is assigned any clearly non-zero value". Null-pointer dereferences are usually the result of failing to check for null (and our algorithm will suggest just such a check by "deleting" the dereference from some paths) rather than the result of failing to include a key assignment. Preventing that event from being inserted ensures that only patches that add null checks will be produced.

Note that if insertions or deletions are disallowed the Nearest Accepting String algorithm may fail to find a nearest accepting string for certain policies and violations. For example, if all insertions are disallowed and the start state of the policy is not an accepting state, some (short) violations will have no nearest accepting string. In such cases we can either fail to produce a candidate patch or we can reset each undefined costs to the maximum cost and try again. The distance metric encodes heuristics; violating them to produce an optional patch may be preferable to producing nothing.

In the experiments in Section 5 we used a uniform insertion cost of 1, a deletion cost of 2, and we handled null-pointer dereferences as above. Favoring insertions over deletions represents our belief that most mistakes are caused by the programmer forgetting to do something good in some cases rather than by the programmer doing something extraneous in some cases. It also showcases our algorithm's ability to produce patches that refer to code that is not present in the original buggy source.

4.4 Algorithm Summary

Given a safety policy, an edit distance metric, a control-flow graph, a path through that CFG that violates the policy, and the program source, we construct a textual candidate patch. The key components of our algorithm are computing a nearest accepting string, which encodes our desired changes, and then mapping those changes back to the source using path predicates to avoid introducing new errors. This is the first algorithm we are aware of that produces candidate patches from safety policy bug reports.

The patch is meant to make it more likely that the bug be addressed and not to be applied automatically. The patch may highlight the root cause of the bug or it may focus on a symptom; in either case it provides useful information to the programmer. The patch may have holes where arguments and return values for inserted function calls are not covered by the specification. Modulo such holes, the patch comes with the guarantee that if it were applied it would fix that particular violating path and would not introduce any new violations of that policy on other paths.[3]

5. Experiments

In this section we describe an experiment conducted to test the hypothesis that bug reports that also contain candidate patches are more likely to be addressed than bug reports with only backtraces. Our experiments focus on open source software because of its availability and because of the accessibility of its development teams (i.e., we can email bug reports directly rather than going

[3] The proposed patch could violate other policies that involve the same events but that are not given as input to the algorithm.

through a customer service department). The general experimental methodology is to submit some bug reports as normal and some bug reports with candidate patches and then measure which are addressed. We must take special care to control all of the variables except for the presence of the patch.

A number of additional factors help to determine whether a bug is addressed. For example, conventional wisdom in the bug-finding community claims that an email listing 10 real bugs is more likely to result in a single bugfix than an email listing 100 real bugs. Similarly, certain classes of bugs, like security vulnerabilities or standards compliance, are viewed as more likely to be addressed than others.

We applied two off-the-shelf bug-finding tools to various open-source programs using a variety of simple safety policies. Our first tool is FindBugs [30], abbreviated FB, a light-weight pattern-and-heuristic bug-finding tool for Java (similar to Metal [22]). Our second tool is WN, a tool for finding mistakes made in exceptional situations in Java programs [43]. All of the bugs reported in this experiment dealt with leaked resources, correct API usage, and avoiding null-pointer dereferences.

In some cases FindBugs may not explicitly produce a counterexample backtrace for an error report and may instead only indicate a particular expression (e.g., for a null-pointer dereference). Many lightweight bug-finding tools are similarly terse; in such cases we can either modify the tool (often the analysis keeps an internal representation of the path to discover the bug but does not expose it) or use PSE [37] to automatically generate a backtrace given the program and the failure. We thus assume that a violating path is always available.

Running a given bug-finding tool on a given program to check a given safety policy produces a set of candidate bug reports. For the purposes of controlling this experiment, we first manually inspect the bug reports and remove all false positives. Dealing with or ranking false positives is an orthogonal problem to producing candidate patches. If the number of remaining bug reports is odd we discard one at random. Within each set of bug reports we choose half of the reports at random and annotate them with candidate patches using the algorithm in Section 4. We then randomize the order of the bug reports within that set and email or otherwise submit all of the bug reports.[4]

It is difficult to determine whether a bug has been truly *fixed*. In this experiment we say that a bug report has been *addressed* when either (1) the developer we submitted the bug report to says so or (2) manual inspection reveals that the appropriate code has been changed in the next release or in the project version control system. Similar in spirit would be (3) the bug-finding tool, when applied to the next release, no longer reports the error. Given the number of bugs in this experiment we chose manual inspection in order to rule out the effects of bug-finding tool imprecision; a larger-scale experiment might rely on (3) alone. We measured the number of reported bugs that had been addressed within two weeks of being reported. It is possible that some of these bugs would have been fixed in any event within two weeks, even without a bug report. Since we chose at random which reports to extend with patches, however, such a rising tide of software quality would affect both the patched reports and the normal reports equally.

The results are shown in Figure 5. We submitted 76 bug reports in total; 33 of them (or 43%) were addressed within two weeks. Bug reports that also included patches were addressed 66% of the time while normal bug reports were addressed 21% of the time.

[4] We emailed reports to the first developer listed on the project page or documentation. The emails were form letters with the project and developer names substituted in. Candidate patches were made against the most recent version of the source code available (e.g., a CVS checkout taken that day).

Program	LOC	Tool	Bugs	Normal	**Patch**
hsqldb	65k	WN	20	4	**5**
ssl-explorer	102k	WN	10	0	**0**
mckoi sql	116k	WN	6	0	**0**
openwfe	128k	FB	4	0	**2**
jboss	145k	WN	26	1	**13**
jasper reports	152k	FB	4	0	**2**
azureus	232k	WN	6	3	**3**
Total	940k		76	8	**25**

Figure 5. Bugs reported and addressed. The "Bugs" column counts total bug reports (no false positives) per program. Each bug report included a backtrace or counterexample; for each program half of the reports (at random) also included a candidate patch. The "Normal" column counts non-patch reports addressed within two weeks. The "Patch" column counts patch-included reports addressed within two weeks.

If candidate patches had no effect on whether a bug report was addressed we would expect to see roughly the same number of patched bug reports addressed as normal bug reports addressed. One *null hypothesis* for this experiment is that candidate patches did not matter at all and that each bug was fixed with probability 43%. In statistical hypothesis testing a p-value is the probability, assuming that the null hypothesis is true, of getting a result *less* favorable to the null hypothesis than the observed value. A standard cutoff $p < 0.05$ is used to judge *statistically significant* values. For this experiment ($\chi^2 = 4.378$), $p = 0.0364$ and thus by conventional standards the results are statistically significant [1, 24, 34].

This experiment does not measure how "close" the candidate patch was to the final fix. In addition, it does not support the claim that the bug reports with explanatory patches take less time or effort to fix. Such probing studies, presumably involving human subjects, remain as future work. More strictly, this experiment only shows that patches work, it does not show why. We acknowledge that this experiment, while significant, is too small draw general findings from and we hope to encourage similar experiments in the future.

This experiment also does not directly address the claim that a similar benefit might be gained by including some other form of additional feedback with bug reports. For example, it is reasonable to assume that including 10 lines from the "minimal cause" of the bug (an important subset of the counterexample trace [6, 28]) might also cause more bug reports to be addressed. However, the "Normal" bug reports in our experiment did include complete counterexample traces, which were on average only 6 events long (and at most 34 events long). Our intuition that for this class of bugs, explanatory patches are more likely to help get bugs addressed than minimal causes. However, since the approaches are complementary and we are interested in improving software quality, both can be used. In addition, this is the first experiment we are aware of that concretely demonstrates a link between addressed bug reports and such additional feedback.

In this study manual inspection found that the bug reports addressed using explanatory patches did not introduce new bugs with respect to any other safety policy we were aware of. The patches and the developer fixes typically involved inserting or moving `close` calls and null-pointer checks, and were local enough to avoid breaking other invariants.

To perform the experiments, bug reports from the two bug-finders were manually tranformed into an intermediate format. The basic algorithm in Section 4 was applied (using an iterative computation of P_k, not IDA*) to half of the reports. The control-flow graph was obtained with a standard Java front-end. The textual `diff` calculations were done between a pretty-printed version of the original source CFG and a pretty-printed version of the patch being considered. Annotating the safety policy with dataflow information (as in Figure 1) was sufficient to capture typestate dataflow in these examples. The final candidate patch was then hand-applied to the original source to create the candidate patch (to avoid issues with line renumberings and stripped comments). Path predicate tools for Java were not always available, so path predicates were computed by hand as necessary. In our experiments the most complicated path predicate necessary was "`text != null && length > 0`"; all other non-empty predicates were of the form "`localVar != null`" or Exc(i).

We conclude that including an additional candidate patch with a bug report makes it more likely for that bug report to be addressed. We presume (but do not test here) that programmers use some sort of implicit cost-benefit analysis when considering whether to address a bug and that the inclusion of a candidate patch lowers the perceived cost.

6. Related Work

Related work in computing path predicates is described in Section 4.2. In addition, there has been much work on error-correcting compilers and parsers (e.g., [2, 17]). While such frameworks are conceptually similar to our approach in mechanism (i.e., given an invalid sequence, produce the nearest valid sequence according to some metric), their goals are quite different. The patches we generate, although equipped with a particular safety guarantee, are not intended to be applied automatically. In spirit our work is more similar to attempts to explain type errors in polymorphic languages like ML (e.g., [20, 42]).

The PSE backward dataflow analysis [37] produces execution traces or paths (not patches) along which the program can be driven to reach a given failure. PSE is complementary to our work and we use its algorithm in Section 5 to generate error traces for some bug reports. We believe that reports should come both with backtraces and with candidate patches.

Leino et al. [35] present an approach for generating error traces (not patches) from verification-condition violations in axiomatic semantics tools. They generate special labeled verification conditions that encode source code locations. Their approach is complementary to ours and allows our technique to be used with error reports from tools like ESC/Java [23].

The localization work of Ball et al. [6] is closely related to ours. They narrow down a large counterexample backtrace (the error symptom) to a few lines (a potential cause). Their algorithm is also complementary to ours: a bug report could consist of a *localized* backtrace and a candidate patch. The key conceptual difference is that their algorithm is limited to the given trace and source code and will never localize the "cause" of an error to a missing statement or suggest that a statement be moved. Our approach can infer new code that should be added. Ten of the 38 patches in Section 5 involved inserting completely new code, and we think the ability to present a patch involving code that is not present is worthwhile.

Producing clearer counterexamples is the most common alternative to producing explanatory patches. Most such work is evaluated in terms of counterexample size (e.g., a minimized counterexample might include only two critical statements instead of over a hundred and is thus said to localize or explain the error) and not by the number of bugs fixed. Including a clear, minimal counterexample in a bug report as a witness to the error should intuitively make the report more likely to be addressed, but we are aware of no published results directly supporting that claim. Although many tools list false positive rates or count bugs reported and fixed, we are unaware of any other published work that relates bug reporting techniques to the number of bugs eventually addressed.

Groce et al. have done much work in minimizing [27] and explaining [9] counterexamples for bounded model checking. Their explanation algorithms rapidly find the closest non-spurious valid program execution path to a given error path. An explanation is then generated by computing a difference (either abstractly or concretely) between the close valid path and the actual error path. This work is also complementary to ours: an ideal error report could contain a minimal localized backtrace, an explanation, and a patch. In addition, since they compute differences with respect to runs through the existing program, they will generally not be able to explain bugs resulting from missing code. On the other hand, their work makes use of special counterfactual dependence distance metrics [26] to help locate the *cause* of an error. In our approach the overarching metric is patch size, with the assumption that smaller candidate patches make it more likely that the bug will be addressed. While there are no extant comparative experiments either way, their approach may be more likely to isolate the root cause of an error (provided that the cause is present in the program text). In general, however, our work is not directly concerned with automatically synthesizing a perfect error explanation but instead with adding information to an error report in order to get that report addressed.

A second point of comparison with Groce et al. is that while both approaches work on general (model-checking) backtraces, the typical domains are quite different. For example, their approach has been applied to critical multithreaded aircraft operating systems code and was successfully used to explain a complicated bug with a multiline minimal cause [28]. Our approach was designed for a setting where there are many bugs, where the software ships with known bugs, or where bugs must be fixed before each commit. In such a scenario getting the developer up to speed on the bug report is of primary importance to increasing the chance that a bug is fixed and to decreasing the time spent fixing each bug.

7. Future Work

This work represents a first step to making automatically-generated bug reports more likely to be addressed by developers; many avenues of investigation remain unexplored.

The relationship between the relevance of the produced patches and the chosen edit distance metric should be evaluated. We selected a simple metric that works well in practice, but particular domains might favor others. For example, if memory leaks are known to be much more common than double frees, a higher cost ratio between insertions (allocations) and deletions (frees) would be reasonable.

Collecting feedback from the developers would help to untangle some of the reasons why bugs were addressed or were not addressed (e.g., perhaps one project had more available programmers or the technique produced better patches for one code base than for another). In particular, comments on the quality of the produced patches and the reasons for not addressing a bug would be helpful. More insight into patch readability and understandability might help to explain the wide variation between different projects in our experiments. It would also be reasonable to submit fake patches or bug reports that claim to have additional information to see if they are also more likely to be addressed; we have not done so because we did not wish to begin by burdening or antagonizing developers.

8. Conclusion

Many existing bug-finding analyses and tools produce (or can be extended to produce) counterexample backtraces that witness safety policy violations (i.e., bugs) in programs. These backtraces are included in bug reports that are presented to developers. In practice software ships with known bugs and not all bug reports are addressed.

We present an algorithm for automatically constructing an explanatory patch for each bug. Fuzzy string matching approaches are used to find a valid sequence of events similar to those on the buggy path. Using a number of existing techniques related to path predicates and program slicing, the program is changed to incorporate the new sequence of events, but only along the violating path. A textual patch is then created to represent the differences between the original program and the modified program. This patch may suggest the inclusion of new code that was not in the original program. The patch comes with a guarantee that applying it will not introduce any new errors along paths unrelated to the reported violation with respect to the given safety policy. The patch is used as a starting point for understanding and addressing the problem.

We present experiments demonstrating that bug reports that also contain explanatory patches are more likely to be addressed in practice. In our experiments, bug reports with patches were three times as likely to be addressed. We believe that the ultimate purpose of bug-finding tools and software model checkers is to increase the quality of software by getting bugs fixed. Our patch generation algorithm works with most software bug-finding tools and serves as a generic post-processing step that makes it more likely that the bugs they find will actually be addressed. These enriched bug reports make it easier for maintainers to address defects.

References

[1] M. Abramowitz and I. A. Stegun. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. Dover, New York, ninth Dover printing, tenth GPO printing edition, 1964.

[2] A. V. Aho and T. G. Peterson. A minimum-distance error-correcting parser for context-free languages. *SIAM Journal of Computation*, 1(4):305–312, Dec. 1972.

[3] R. Alur, P. Cerny, P. Madhusudan, and W. Nam. Synthesis of interface specifications for Java classes. In *ACM Symposium on Principles of Programming Languages*, 2005.

[4] G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. In *Principles of Programming Languages (POPL)*, pages 4–16, 2002.

[5] T. Ball and J. R. Larus. Efficient path profiling. In *International Symposium on Microarchitecture*, pages 46–57, 1996.

[6] T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: localizing errors in counterexample traces. *SIGPLAN Not.*, 38(1):97–105, 2003.

[7] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 2001, Workshop on Model Checking of Software*, volume 2057 of *Lecture Notes in Computer Science*, pages 103–122, May 2001.

[8] L. Carter, B. Simon, B. Calder, L. Carter, and J. Ferrante. Path analysis and renaming for predicated instruction scheduling. *International Journal of Parallel Programming*, 28(6):563–588, 2000.

[9] S. Chaki, A. Groce, and O. Strichman. Explaining abstract counterexamples. In *Foundations of Software Engineering (FSE)*, pages 73–82, New York, NY, USA, 2004. ACM Press.

[10] H. Chen, D. Dean, and D. Wagner. Model checking one million lines of C code. In *Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2004.

[11] E. M. Clarke, O. Grumberg, K. L. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *Conference on Design Automation (DAC)*, pages 427–432, New York, NY, USA, 1995. ACM Press.

[12] J. C. Corbett, M. B. Dwyer, J. Hatcliff, and Robby. Bandera: a source-level interface for model checking java programs. In *International Conference on Software Engineering (ICSE)*, pages 762–765, New York, NY, USA, 2000. ACM Press.

[13] L. Cosmides, J. Tooby, A. Montaldi, and N. Thrall. Character counts: Cheater detection is relaxed for honest individuals. In *11th Annual Meeting of the Human Behavior and Evolution Society*, Salt Lake City, Utah, June 1999.

[14] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[15] M. Das. Unification-based pointer analysis with directional assignments. In *Programming Language Design and Implementation (PLDI)*, pages 35–46, New York, NY, USA, 2000. ACM Press.

[16] M. Das, S. Lerner, and M. Seigle. ESP: path-sensitive program verification in polynomial time. *SIGPLAN Notices*, 37(5):57–68, 2002.

[17] P. Degano and C. Priami. Comparison of syntactic error handling in LR parsers. *Software - Practice and Experience*, 25(6):657–679, 1995.

[18] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *ACM Conference on Programming Language Design and Implementation*, pages 59–69, 2001.

[19] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[20] D. Duggan and F. Bent. Explaining type inference. *Sci. Comput. Program.*, 27(1):37–83, 1996.

[21] P. Eggert, M. Haertel, D. Hayes, R. Stallman, and L. Tower. diff - compare files line by line. In *http://www.gnu.org/software/diffutils/diffutils.html*, 2005.

[22] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Symposium on Operating Systems Design and Implementation*, 2000.

[23] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *Programming Language Design and Implementation (PLDI)*, pages 234–245, New York, NY, USA, 2002. ACM Press.

[24] D. Freedman, R. Pisani, and R. Purves. *Statistics*. Third edition. W. W. Norton, 1998.

[25] G. Gigerenzer and K. Hug. Domain-specific reasoning: social contracts, cheating and perspective change. In *Cognition*, volume 43, pages 127–171, 1992.

[26] A. Groce. Error explanation with distance metrics. In *Lecture Notes in Computer Science*, volume 2988, pages 108–122, Jan. 2004.

[27] A. Groce and D. Kroening. Making the most of bmc counterexamples. In *Electronic Notes in Theoreitcal Computer Science*, volume 119, pages 67–81, 2005.

[28] A. Groce and W. Visser. What went wrong: Explaining counterexamples. In *Lecture Notes in Computer Science*, volume 2648, pages 121–135, Jan. 2003.

[29] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Principles of Programming Languages (POPL)*, pages 58–70, 2002.

[30] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA)*, pages 132–136, New York, NY, USA, 2004. ACM Press.

[31] R. Jhala and R. Majumdar. Path slicing. In *Programming Language Design and Implementation (PLDI)*, pages 38–47, New York, NY, USA, 2005. ACM Press.

[32] R. E. Korf. Depth-first iterative deepening: an optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, Sept. 1985.

[33] T. Kremenek, K. Ashcraft, J. Yang, and D. Engler. Correlation exploitation in error ranking. In *Foundations of Software Engineering (FSE)*, pages 83–93, New York, NY, USA, 2004. ACM Press.

[34] T. Kremenek and D. Engler. Z-ranking: Using statistical analysis

to counter the impact of static analysis approximations. In *Static Analysis, 10th International Symposium (SAS)*, pages 295–315, Jan. 2003.

[35] K. R. M. Leino, T. Millstein, and J. B. Saxe. Generating error traces from verification-condition counterexamples. In *Science of Computer Programming*, volume 55, pages 209–226, Mar. 2005.

[36] U. Manber and S. Wu. Fast text search allowing errors. *Communications of the ACM*, 35(10):83–91, 1992.

[37] R. Manevich, M. Sridharan, S. Adams, M. Das, and Z. Yang. Pse: Explaining program failures via postmortem static analysis. In R. N. Taylor, editor, *Foundations of Software Engineering (FSE)*. ACM, nov 2004.

[38] V. P. Ranganath, T. Amtoft, A. Banerjee, M. B. Dwyer, and J. Hatcliff. A new foundation for control-dependence and slicing for modern program structures. In *European Symposium on Programming (ESOP)*, pages 77–93, Jan. 2005.

[39] T. Robschink and G. Snelting. Efficient path conditions in dependence graphs. In *International Conference on Software Engineering (ICSE)*, pages 478–488, New York, NY, USA, 2002. ACM Press.

[40] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *USENIX Security Symposium*, pages 201–220, 2001.

[41] M. Sipser. *Introduction to the Theory of Computation*. Second edition. PWS, 1997.

[42] F. Tip and T. B. Dinesh. A slicing-based approach for locating type errors. *ACM Trans. Softw. Eng. Methodol.*, 10(1):5–55, 2001.

[43] W. Weimer and G. Necula. Finding and preventing run-time error handling mistakes. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Vancouver, British Columbia, Canada, Oct. 2004.

[44] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *International Symposium of Software Testing and Analysis*, 2002.

[45] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: mining temporal API rules from imperfect traces. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 282–291, New York, NY, USA, 2006. ACM Press.