2008 Sixth IEEE International Conference on Software Engineering and Formal Methods

PED: Proof-guided Error Diagnosis by Triangulation of Program Error Causes

Gogul Balakrishnan Malay Ganai
NEC Laboratories, Inc., USA
E-mail: {bgogul,malay}@nec-labs.com

Abstract

Error diagnosis, which is the process of identifying the root causes of bugs in software, is a time-consuming process. In general, it is hard to automate error diagnosis due to the unavailability of a full "golden" specification of the system behavior in realistic software development. We propose a repair-based proof-guided error diagnosis (PED) framework, that provides a first-line attack to find the root causes of the errors in programs by pin-pointing the possible error-sites (buggy statements), and suggesting possible repair fixes. Our framework does not need a complete system specification. Instead, it automatically "mines" partial specifications of the intended program behavior from the proofs obtained by static program analysis for standard safety checkers. It uses these partial specifications along with the multiple error traces provided by a model checker to narrow down the possible error sites. It also exploits inherent correlations among the program statements. To capture common programming mistakes, it directs the search to those statements that could be buggy due to simple copy-paste operations or syntactic mistakes such as using $\leq$ instead of $<$. To further improve debugging, it prioritizes the repair solutions. We implemented and integrated the PED tool as a plug-in module to a software verification framework. We show the efficacy of such a framework on public benchmarks.

1. Introduction

Model checking is one of the most successful techniques used to identify bugs in software and hardware [3]. One of the advantages of model checking is that it provides a concrete error trace in the program that shows how the error state is reachable (i.e., how the bad behavior manifests). Such error traces can be very long and complex, and therefore, it is quite cumbersome and time consuming to manually examine the trace to identify the root cause of the error. In several cases, a bug may be due to problems in more than one statement that are quite far apart in the error trace. Further, it is very hard to identify the root cause of an error just by examining a single error trace.

Error diagnosis is the process of automatically identifying the root causes of program failures. Several error diagnosis techniques have been proposed in the recent past [2, 4, 10, 11, 12, 15, 24, 26, 28]. Stumptner and Wotowa present an excellent survey of various error diagnosis techniques [25]. The main problem faced by all error diagnosis techniques is that it is not practical to have a "golden specification" of the correct program behavior against which the behavior of a buggy program can be compared to identify the root causes of an error. Previous methods rely on the availability of correct traces or derive such traces explicitly using model checking tools. The differences between erroneous and correct traces are used to infer the causes of the errors. Most often these differences do not provide an adequate explanation of the failures.

We propose a repair-based proof-guided error diagnosis (PED) framework that will assist a programmer in prioritizing or pin-pointing the root causes of program errors reported by model checkers. In such a repair-based approach (also, referred as replacement diagnosis [25]), the buggy program is first modified to obtain a repair program in which the behavior of the statements or branches can be controlled via auxiliary program variables referred to as *selector* variables. Analysis is carried out on the repair program to identify values for the selector variables such that the behavior of only a small set of statements or branches need to be modified in the repair program to prevent the error. Such an approach, in general, produces a large set of possible repair solutions. Our approach improves such replacement diagnosis by identifying the most relevant repair solutions in the following novel ways:

Mining Partial Specifications: We do not need a complete specification of the intended program behavior. Instead, we automatically extract the partial specification of the intended behavior from the results of static analysis. In many cases, static program-analysis algorithms can prove that standard safety checkers, such as array-bound violations checker and null-pointer dereferences checker, cannot be violated for all possible executions of the program. In

978-0-7695-3437-4/08 $25.00 © 2008 IEEE
DOI 10.1109/SEFM.2008.35

such cases, we extract the invariants relevant for the proofs efficiently and use them as partial specification of the intended behavior of the program. For more restrictive repair solutions, we also identify the program statements that are relevant for the proofs and mark them as "trusted". The idea is not to modify these statements in repair program, and restrict the search for error sites to untrusted program statements.

Syntactic Closeness: Significant number of errors in software are caused due to copy-paste operations [20]. Further, many errors are caused due to syntactic mistakes such as using $\leq$ instead of $<$. We give preferences to "syntactic closeness" of operators and expressions to improve error localization. (We use syntactic closeness as a heuristic to guide our search for repair solutions; our techniques are general and *not* restricted to just finding such syntactic mistakes in the program.)

Correlation: We also exploit the inherent correlation among the statements occurring in the *use-def* chains and statements corresponding to the unrolling of the loop body in an error-trace to reduce the set of repair solutions.

Multiple Error Traces: Presence of a bug often results in violation of many checkers. By taking an intersection of the repair solutions corresponding to error traces for the violation of different checkers, we narrow down the set of possible error-sites, which improves debugging.

Ranking: We also propose several ranking criteria for the repair solutions (such as prioritizing solutions with minimal changes in the repair program) so that the user only has to examine the most relevant fixes.

We have implemented and integrated our PED tool as a plug-in module to a software verification framework F-SOFT [14] (as shown in Fig. 1). Using the PED tool, a programmer can fix the code by reviewing only the prioritized repair solutions before moving to next phase of time-consuming debugging. Note that our technique is not specific to the F-SOFT framework. We evaluated the PED tool on a set of publicly available benchmarks, and show that buggy statements can easily be found by manual debugging of a handful of repair solutions.

The rest of the paper is organized as follows: §2 provides an overview of the F-SOFT tool and describes how the proof-guided error diagnosis technique is incorporated into the F-SOFT tool chain. §3 describes the various components of the PED tool. §4 describes improvements to the basic PED technique. §5 presents various ways to rank the solutions provided by the error diagnosis tool. §6 presents our experience with using the PED tool on a collection of publicly available benchmarks.

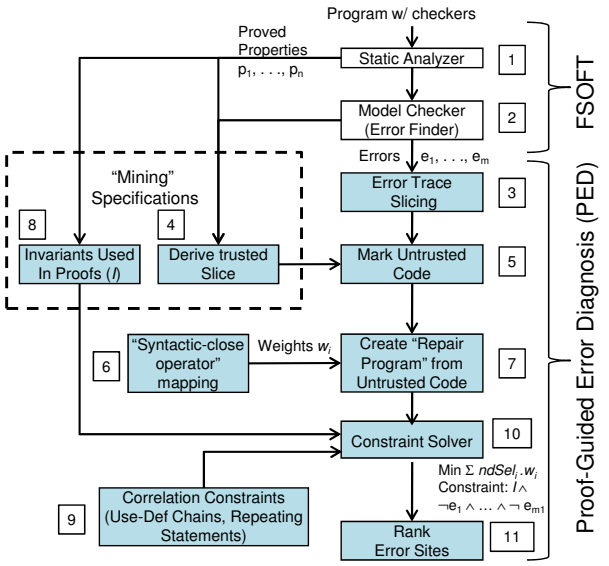

Figure 1. Proof-Guided Error Diagnosis Tool.

2. F-SOFT Overview

F-SOFT is a program verifier that works on C programs [14]. It uses a combination of static analysis and model checking techniques to identify array out-of-bound violations, null-pointer dereferences, improper usage of C String API, etc. The components of F-SOFT are shown in Fig. 1.

Example 2.1 *Consider the function* `sum` *shown in Fig. 2(a). It computes the sum of the elements in array* `a`*, which is of size* `n`*. The function has an array out-of-bounds error because the loop-terminating condition is* `i` $\leq$ `n` *(in bold) instead of* `i` $<$ `n`*.* □

As a first step, a given program is annotated with checks for the violation of safety properties that are of interest to the user. A *safety property* is a pair $\langle S, \varphi \rangle$, where S is a label in the program and φ is an assertion on the states that can reach the label S. A safety property is violated if an execution of the program reaches label S with a state that does not satisfy the assertion φ. For a safety property $\langle S, \varphi \rangle$, the statement "**`if`**$(\neg\varphi)$ `ERR`();" is inserted at label S in the program, where $\neg\varphi$ is the logical negation of φ, and `ERR`() is a function that aborts the program. Such annotations are referred to as *property checkers*.

Fig. 2(b) shows the program in Fig. 2(a) annotated with array out-of-bounds checkers at `P1` and `P2`. The variables `a_lo` and `a_hi` refer to the lowest and the highest possible addresses for array `a`, respectively. Property `P1` corresponds to the *underflow* out-of-bounds error, while property `P2` corresponds to the *overflow* out-of-bounds error.

```
int sum(int a, int n) {
  int i, s = 0;
  for(i=0; i<=n; ++i) {
    s = s + a[i];
  }
  return s;
}

  /* sum elements of */
  /* array a */
int main() {
  int a[10];
  sum(a, 10);
  return 0;
}
```

(a)

```
int sum(int a, int n) {
  int i, s = 0;
  for(i=0; i<=n; ++i) {
    P1: if(a+i<a_lo) ERR();
    P2: if(a+i>a_hi) ERR();
    s = s + a[i];
  }
  return s;
}

int main() {
  int a[10];
  sum(a, 10);
  return 0;
}
```

(b)

Figure 2. (a) Buggy code, and (b) with checkers.

Step	Statement
1	$\texttt{i} = 0;$
2	Loop condition $(\texttt{i} \leq \texttt{n})$ is true.
3	$\texttt{i} = \texttt{i} + 1;$
4	Loop condition $(\texttt{i} \leq \texttt{n})$ is true.
⋮	⋮
19	$\texttt{i} = \texttt{i} + 1;$
20	Loop condition $(\texttt{i} \leq \texttt{n})$ is true.
21	$\texttt{i} = \texttt{i} + 1;$
22	Loop condition $(\texttt{i} \leq \texttt{n})$ is true.
23	Safety check $(\texttt{a} + \texttt{i} \leq \texttt{a\_hi})$ fails.

Figure 3. An error trace showing the violation of the property checker `P2` in Fig. 2(b).

Static Analyzer The annotated program is analyzed by a static analyzer, which uses various dataflow algorithms ranging from simple constant folding to more complex numerical analysis algorithms such as intervals [5], octagons [21], polyhedra [13], and disjunctive numerical domains [23] to compute state invariants for all the statements in the program. A safety property $\langle S, \varphi \rangle$ is proved by the static analyzer if the invariant ψ computed at label S is such that $\psi \wedge \neg\varphi$ is false.

For the program in Fig. 2, the static analyzer computes the invariant $\texttt{a} + \texttt{i} \geq \texttt{a\_lo}$ at `P1`, which implies that the underflow condition $\texttt{a} + \texttt{i} < \texttt{a\_lo}$ never occurs at `P1`. However, the static analyzer is not able to prove `P2`. In this case, it is because the program has an array out-of-bounds error. However, in general, the computed invariants may not be precise enough to establish the fact that a safety property is not violated in the program (even if that is the case).

Model Checker The property checkers for the safety properties that are proved by the static analyzer are pruned away, and the resulting program is analyzed by a model checker. F-SOFT performs bounded model checking to determine if any of the remaining safety properties are violated. If the model checker finds any violations, it generates an error trace showing how the property is violated. The error trace generated by the model checker is sliced with respect to the variables in the violated property, and the sliced trace is shown to the user.

Without loss of generality, we assume that there are only three kinds of steps in an error trace: (1) an assignment of the form `x := expr`, where `x` is a program variable and `expr` is an expression in the program, (2) an evaluation of a Boolean predicate `P` that corresponds to a control statement, such as **`if`**, **`while`**, and **`for`**, in the program, and (3) a step representing the violation of a safety property.

For the property `P2` in Fig. 2, the model checker finds the error trace shown in Fig. 3. The statements that are not relevant to the violated property have been sliced away. In this example, the assignments to variable `s` have been removed from the error trace. The error trace consists of 23 steps. Steps $1, 3, 5, \ldots,$ and 21 refer to the assignments to variable `i`, steps $2, 4, \ldots,$ and 22 refer to the evaluation of the loop condition in the program, and step 23 corresponds to violation of the property checker `P2`.

3. Proof-guided Error Diagnosis (PED)

We give the basic terminology, and discuss our repair-based error diagnosis framework. Later, we discuss how we improve the basic diagnosis using static analysis.

3.1 Terminology

An *error* (or *error symptom*) is the violation of a safety property. An *error trace* is the concrete trace provided by the model checker for an error. Given an error trace, the *root causes* (or *error sites*) of an error are the set of buggy statements or conditions that are responsible for the error. *Error localization* refers to the process of locating the error-sites. A *repair solution* to an error is a set of modified statements and/or conditions, i.e, *fixes*, that prevents the corresponding error symptom. For the program in Fig. 2(b), the violation of the property checker `P2` is an error. A root cause (i.e., an error site) for the error is the buggy terminating condition `i<=n` of the"for loop". A repair solution consists of a fix with the condition `i<=n` modified to `i<n`.

3.2 Basic Framework

In this section, we provide an overview of the PED tool, which is shown in Fig. 1. Let $e_1, \ldots, e_m$ be the violated

safety checkers i.e., errors found by a model checker. Let $T_1, \ldots, T_m$ be the corresponding error traces. Given an error trace T_j, the goal of PED is to identify the statements or conditions in the program that are responsible for error e_j. Let $p_1, \ldots, p_n$ be the properties proved by the static analyzer.

Marking untrusted code Certain statements or conditions in the program cannot be a root cause for a given error. For example, the assignments to `s` in the program in Fig. 2 are not responsible for the violation of the property checker at `P2`. Similarly, some of the assignments and conditions in the error trace cannot be a root cause for the error. We refer to those statements and conditions that are not possible root causes as *trusted* and the other statements and conditions as *untrusted*. For a given error trace, an error diagnosis tool only has to consider the statements or conditions in the program that are untrusted. The simplest way to determine the *untrusted* code is to compute a slice [27] of an error trace with respect to the given safety property. In §4.1, we discuss how to identify untrusted statements and conditions in the program using the results of static analysis.

Repair Program After identifying the trusted and untrusted code in the program, PED creates a *repair program* for the given error trace.

Statements: For a trusted statement `S` in the error trace, the repair program has the statement `S` as is. For an untrusted assignment "`x := expr`" at step `k` in a given error trace, the repair program has the following assignment:

`x := ndSel_k ? ndRes_k : expr;`

Variable `ndSel_k` is a *new* non-deterministic Boolean input variable. Variable `ndRes_k` is a non-deterministic input variable that has the same type as the result of `expr`. Variable `ndSel_k` is referred to as a *selector* variable, and variable `ndRes_k` is referred to as the *result* variable.

Conditions: For a trusted condition `P` in the error trace, the repair program has the following **`if`** statement:

```
if(!P) goto END;
```

Label `END` in the **`goto`** statement refers the last statement in the repair program.

For an untrusted condition `P` at step `k` in the error trace, the repair program has the following **`if`** statement:

```
if(ndSel_k?ndRes_k:!P) goto END;
```

`ndSel_k` and `ndRes_k` are *new* non-deterministic Boolean input variables, `!P` refers to the logical negation of condition `P`, and `END` refers to the last statement in the repair program. As in the case of the untrusted assignment, variable `ndSel_k` is referred to as a *selector* variable, and variable `ndRes_k` is referred to as the *result* variable.

For a step that represents the violation of a safety condition `P`, the repair program has the following statement:

```
if(P) goto END;
```

Finally, a call to `ERR()` is added to the repair program after adding the statements for each step in the error trace. A call to `ERR()` aborts the program. Note that setting all the selector variables to the value *false* in the repair program corresponds to the original error trace. Therefore, the call to `ERR()` is always reachable if *false* is assigned to all the selector variables in the repair program.

```
 1:  i = ndSel1?ndRes1:0;
 2:  if (ndSel2?ndRes2:(i>n)) goto END;
 3:  i = ndSel3?ndRes3:i + 1;
 4:  if (ndSel4?ndRes4:(i>n)) goto END;
                   ⋮
19:  i = ndSel19?ndRes19:i + 1;
20:  if (ndSel20?ndRes20:(i>n)) goto END;
21:  i = ndSel21?ndRes21:i + 1;
22:  if (ndSel22?ndRes22:(i>n)) goto END;
23:  if (a+i <= a_hi) goto END;
24:  ERR();
END:
```

Figure 4. Repair program for the error trace shown in Fig. 3.

The repair program has no loops as it is based on an unrolled error trace. The `if` conditions in the repair program are referred to as *branch* statements. Also, the values of input variables in the original program are fixed based on the error trace. Fig. 4 shows the repair program for the error trace in Fig. 3. Statement at label k in the repair program corresponds to step k in the error trace.

Error Localization After creating the repair program, PED performs error localization using the algorithm in Fig. 5. For a given error trace T, the algorithm creates a repair program R. The algorithm also creates the Static Single Assignment (SSA) form R' of the repair program R, and converts R' into a Satisfiability Modulo Theory (SMT) formula M [8]. For each branch B with predicate P, the algorithm checks if the formula $M \wedge P$ (ignore D in Fig. 5 for now) is satisfiable using a SMT Solver. If the formula is satisfiable, the solver provides a satisfying assignment β for the formula. The satisfying assignment provided by the solver is referred to as the *repair solution*.

The repair solution provides a way to identify the possible root causes for an error trace. If the condition $M \wedge P$ is satisfied, then the assignments to the variables in the repair solution provide an execution trace of the repair program such that the predicate P is true when the branch statement B is executed. In such an execution, the call to `ERR()` is never reached because the target of the branch statement is `END`. In other words, the repair solution provides a way for the repair program to avoid the error.

Recall that if the value *false* is assigned to the selector variables in the repair program, `ERR()` is always executed. Therefore, if $M \wedge P$ is satisfied, at least one of the vari-

```
1:  proc LocalizeError(P: Program, T: Error Trace)
2:    Let R be the repair program for error trace T.
3:    Let R' be the SSA form of R.
4:    Let M be the SMT formula representing R'.
5:    F = ∅. // Set of repair solutions.
6:    for each branch statement B in R' do
7:      D = ∅.
8:      Let P be the predicate on the branch statement B.
9:      while (M ∧ P ∧ D) is satisfiable do
10:       Let β be the satisfying assignment.
11:       H = {S|S is a statement in R' ∧ the selector
                  variable of S is true in β.}
12:       Add set H to F.
13:       D = D ∧ ¬β
14:   return F.
```

Figure 5. Algorithm to localize error.

ables in the repair program has the value *true*. Assigning the value $true$ to a selector variable at step k corresponds to changing the semantics of the statement at step k in the error trace. That is, changing the semantics of the statements for which the selector variable has the value $true$ has enabled the program to avoid the error. The error localization algorithm reports these statements as possible error sites to the user. The process is repeated after adding a blocking clause $\neg\beta$ to the condition $M \wedge P$ to find other error sites. Formula D represents the blocking clause for all the fixes reported by the algorithm for branch B so far.

For the program in Fig. 2, the algorithm provides the following solution (among others): *false* for `ndSel1`, `ndSel2`, ..., `ndSel21`, *true* for `ndSel22`, and *true* for `ndRes22`. This solution corresponds to changing the loop condition `i <= n` in the program such that the loop exits at an earlier step, thereby avoiding the array out-of-bound error. Therefore, `i <= n` is a possible error site for the violation of property `P2`.

4. Improving Error Diagnosis

The basic framework, described in §3, generates all possible repair solutions. In this section, we describe various ways in which these repair solutions can be pruned to obtain the solutions that are the most relevant for debugging.

4.1. Mining Partial Specifications

One problem with the technique described in §3 is that the constraint solver may provide solutions that violate one or more of the safety properties proved by the static analyzer. For instance, one of the solutions provided by the constraint solver for the repair program in Fig. 4 assigns *true* to `ndSel1` and -1 to `ndRes1`. While this solution provides a fix for property `P2`, it violates the underflow property `P1`. In this section, we describe how one can extract a partial specification of the intended behavior of the program based on the properties that are proved by abstract interpretation.

The aim of abstract interpretation [6] is to determine the set of states that a program reaches in *all* possible executions, but without actually executing the program on specific inputs. To make this feasible, abstract interpretation explores several possible execution sequences at a time by running the program on descriptors that represent a collection of states. The universal set A of state descriptors is referred to as an *abstract domain*. The set A forms a mathematical lattice $\langle A, \subseteq_A, \sqcup_A, \sqcap_A \rangle$, where $\subseteq_A$ is the lattice ordering, $\sqcup_A$ is the lub operator, and $\sqcap_A$ is the glb operator.

Abstract interpretation is performed on a control-flow graph (CFG) of the program. A CFG G is a tuple $\langle N, E, V, \mu, n_0, \varphi_{n_0} \rangle$, where N is a set of nodes, $E \subseteq N \times N$ is a set of edges between nodes, V is a set of variables, $n_0 \in N$ is the initial node, φ_{n_0} is an initial condition specifying the values that variables in V may hold at n_0, and each edge $e \in E$ is labeled with a condition or update $\mu(e)$.

An abstract interpreter annotates each node n in the CFG with an abstract state descriptor from the abstract domain. The abstract state descriptor φ_n represents an over-approximation for the set of states that a program reaches at the node n in *all* possible executions. During abstract interpretation, the effects of executing an edge $e \in E$ with label $\mu(e)$ in the program is modeled by an *abstract transformer* $\mu^\sharp(e)$ that computes an over-approximation to the effects of executing the original statement in the program.[1] Fig. 6 shows a CFG and the reachable states computed by abstract interpretation using the octagon abstract domain [21].

Definition 4.1 ***(Safety Projection)*** *Let $n \in N$, $\langle n_S, \varphi \rangle$ be a safety property, and P be the set of paths from n to n_S in CFG G. The safety projection of a property $\langle n_S, \varphi \rangle$ onto a node n, denoted by χ_n is the disjunction of the weakest preconditions of φ with respect to every path in P: $\chi_n = \bigvee_{p \in P} WP(p, \varphi)$, where $WP(p, \varphi)$ is the weakest precondition of φ with respect to the path p.*

Intuitively, the safety projection of a property $\langle n_S, \varphi \rangle$ onto a node n represents the set of states at n that *cannot* reach an error state at node n_S. For instance, consider the safety projection of property $\langle n_4, \mathtt{e} \leq 2 \rangle$ onto node $n1$ shown in Fig. 6. If the value of `e` at node $n1$ does not satisfy the safety projection condition $\mathtt{e} \leq 5$, then the assertion at node n_4

[1] In our abstract interpreter, the abstract state for a node n is computed as follows: $\text{absState}'[n] = \text{absState}[n] \sqcup_A (\bigsqcup_{p \in Pred(n)} \mu^\sharp(p \rightarrow n, \text{absState[p]}))$. Consequently, the abstract state for a node n always increases (with respect to lattice order $\subseteq_A$). Therefore, we have monotonicity even if a non-monotonic widening operator is used.

fails. Therefore, the safety projection of a property provides a constraint on the set of permissible values for the variables at every node in the CFG, which is a partial specification of the intended behavior of the program. To improve the quality of the repair solutions provided by the error localization algorithm, we compute the safety projections of the properties that are proved by the static analyzer onto every node in the CFG, and use the safety projections as constraints on the values of the result variables in the repair program.

For the program in Fig. 4, abstract interpretation based on the interval domain [5] gives the following constraints for the non-deterministic result variables: $0 \leq \texttt{ndRes1} \leq 10$, $-1 \leq \texttt{ndRes3} \leq 10$, $-1 \leq \texttt{ndRes5} \leq 10$, ..., $-1 \leq \texttt{ndRes22} \leq 10$. These constraints prevent the constraint solver from picking -1 for variable `ndRes1`. Consequently, the constraint solver does not provide a solution that violates the safety property.

Mining Relevant Invariants Because there may be an infinite number of paths in the CFG of a program, computing the exact safety projection is not computationally feasible. Therefore, we compute an over-approximation to the safety projection of $\langle n_S, \varphi \rangle$ at each node using abstract interpretation. First, a new CFG $G' = \langle N, E', V, \mu', n_S, \neg\varphi \rangle$ is created from the CFG $G = \langle N, E, V, \mu, n_0, \varphi_0 \rangle$ of the program. The set of edges in G' is such that $(n, m) \in E'$ if $(m, n) \in E$, i.e., the edges in G are reversed in G'. Every edge $(n, m) \in E'$ is labeled with the weakest precondition operator for the update or condition $\mu(m, n) \in G$. The initial node for G' is n_S, and the initial condition for G' is the safety condition φ. The abstract value $\chi_n^\sharp$ computed at a node $n \in N$ by performing abstract interpretation on G' represents an *over-approximation* for the safety projection χ_n. We use $\chi_n^\sharp$ as a constraint on the values of the result variable at node n.

Because $\chi_n^\sharp$ is an over-approximation to the actual safety projection χ_n, $\chi_n^\sharp$ may include states at n that lead to the violation of φ at n_S. Therefore, it is not guaranteed that constraint solver will never provide a solution that violates the property. However, the constraints on the values of non-deterministic result values obtained as outlined above works well in practice. On the set of benchmarks described in §6, the number of error sites reported by the error localization algorithm is substantially reduced.

Mining Trusted Code For more restrictive repair solutions, in addition to determining constraints for the result variables, we identify a subset of the program statements that are relevant for the proofs obtained from static analysis, and mark them as "trusted". The idea is not to modify the trusted statements in repair program, and restrict the repair solutions to untrusted program statements.

Definition 4.2 ***(Error Projection)*** *Let $n \in N$, $\langle n_S, \varphi \rangle$ be a safety property, and P be the set of paths from n to n_S in the CFG. The error projection of a safety property $\langle n_S, \varphi \rangle$ onto a node n, denoted by ψ_n is the disjunction of the weakest preconditions of $\neg\varphi$ with respect to every path in P: $\psi_n = \bigvee_{p \in P} WP(p, \neg\varphi)$, where $WP(p, \neg\varphi)$ is the weakest precondition of $\neg\varphi$ with respect to the path p.*

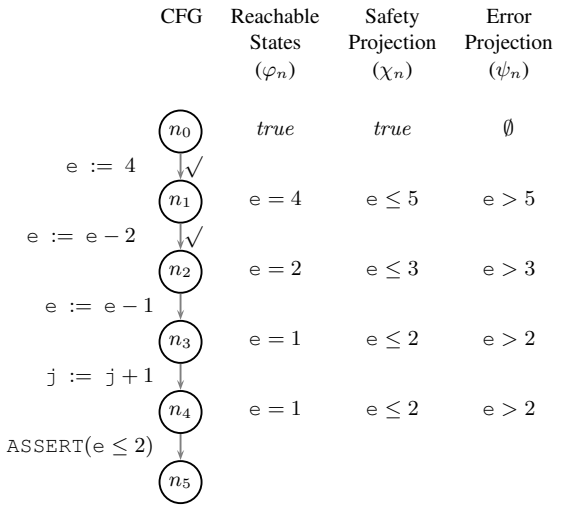

Figure 6. Reachable states, error projection, and safety projection for the property $\langle n_4, \texttt{e} \leq 2 \rangle$. $\surd$ refers to the relevant edges.

Intuitively, the error projection of a property $\langle n_S, \varphi \rangle$ onto a node n represents the set of states at n that reach an error state at node n_S. For instance, consider the error projection of property $\langle n_4, \texttt{e} \leq 2 \rangle$ onto $n1$ shown in Fig. 6. If the value of `e` satisfies the error projection condition $\texttt{e} > 5$, then the assertion $\texttt{e} \leq 2$ at node $n4$ fails. Analogous to safety projections, we use abstract interpretation to compute an over-approximation $\psi_n^\sharp$ for the concrete error projection ψ_n at each node n in the program. In the following, we first define relevant edge, and then discuss how we use error projections to find trusted statements or conditions in the program.

Definition 4.3 ***(Relevant Edge)*** *Let $\langle n_S, \varphi \rangle$ be a safety property that is proved by static analysis, φ_n be the invariant at node n computed by static analysis on G and ψ_n be the error projection of $\langle n_S, \varphi \rangle$ onto node n. An edge $m \rightarrow n \in E$ is* relevant *if the following holds:*

$$(\varphi_n \wedge \psi_n = \emptyset) \wedge (\varphi_m \wedge \psi_n \neq \emptyset) \tag{1}$$

The conjunct $(\varphi_n \wedge \psi_n = \emptyset)$ encodes the fact that error state is *not* reachable from the node n. The conjunct $(\varphi_m \wedge \psi_n \neq \emptyset)$ encodes that the error state is possibly reachable from m if the transition $m \rightarrow n$ is not present in the program. That

is, if Eqn. (1) is true then the program would reach an error state when the transition $m \rightarrow n$ is removed. Therefore, the edge $m \rightarrow n$ is important for proving the safety property $\langle n_S, \varphi \rangle$. For the CFG in Fig. 6, edge $n_1 \rightarrow n_2$ is relevant because Eqn. (1) holds. For $n_1 \rightarrow n_2$, we have $\varphi_{n_1} : \mathtt{e} = 4$, $\varphi_{n_2} : \mathtt{e} = 2$, $\psi_{n_1} : \mathtt{e} > 4$, and $\psi_{n_2} : \mathtt{e} > 3$. Consequently, $\varphi_{n_2} \wedge \psi_{n_2} = \emptyset$ and $\varphi_{n_1} \wedge \psi_{n_2} \neq \emptyset$. In Fig. 6, only edges $n_0 \rightarrow n_1$ and $n_1 \rightarrow n_2$ are relevant.

Defn. 4.3 provides a simple and efficient way to identify the transitions that are important for the static analyzer to prove a given safety property. We provide a theoretical basis for such an observation as follows. Let $m \rightarrow n$ represent the relevant edge with the abstract transformer $\mu(m \rightarrow n)$ in a CFG G, with the safety checker $\langle n_S, \varphi \rangle$.

Lemma 4.1 $\varphi_n \subsetneq_A \varphi_m$.

Proof: Follows from Defn. 4.3. □

Lemma 4.2 *If we replace $\mu(m \rightarrow n)$ with the identity transformer in a modified CFG G', then $\varphi_n \subsetneq_A \varphi'_n = \varphi'_m$, where φ'_m and φ'_n refer to the invariants computed at node m and n, respectively, in G'. Further, $\varphi_{n_S} \subseteq_A \varphi'_{n_S}$.*

Proof: Follows from Lemma 4.1 and monotonicity of the abstract transformers. □

Observation 4.1 *For a given G if we replace all $\mu(m \rightarrow n)$ with identify transformers to obtain a modified CFG G', the static analyzer may no longer find the proof for the safety checker, i.e., $\varphi'_{n_S} \wedge \neg\varphi = \emptyset$ may not hold.*

As φ'_{n_S} gets larger (Lemma 4.2), it may likely contain the error state, and therefore, a static proof may not hold in G'. On the other hand, a static proof may still hold in G' if an edge that is not relevant in G has become relevant in the modified G'. This can happen when for some edge $a \rightarrow b$, the following condition holds (*inadequacy condition*): $\psi_b = \emptyset$ in G, but $\psi'_b \neq \emptyset$ in G'.

Based on the Observation 4.1, one can obtain a set of adequate relevant edges, by identifying all the relevant edges in a CFG and replacing the corresponding abstract transformers with the identity transformers in the modified CFG, and iterating the process on the modified CFG until *inadequacy condition* does not hold. However, for error diagnosis, we do not need adequate set of edges, though such a set would give a more precise result. For efficiency reasons, we choose a single iteration to obtain the relevant statements from a given CFG, and mark the relevant statements as "trusted". For trusted statements, the associated selector variables are set to $false$. Consequently, the trusted statements are not modified in the repair program. (Note that, while safety projections are used to obtain constraints on the values of the result variables in the repair program, error projections are used to obtain constraints on the values of the selector variables in the repair program.)

For the repair program in Fig. 3, the assignment "`i = 0`" at step 1 is marked as relevant (using the proof of `P1`). Therefore, the error localization algorithm does not report the statement "`i = 0`" as an error site. Note, that this is an improvement over the previous case where we only specify constraints on the result variables. In the previous case, the error localization algorithm may report "`i = 0`" as a possible error site.

4.2. Annotation Library

In the program shown in Fig. 4, the constraint solver may choose $true$ for the result variable at any branch in the execution of the program. That is, the error is avoided trivially by cutting the execution of the program at any arbitrary branch. Such behavior causes the error diagnosis algorithm to report useless repair solutions.

To avoid this problem, we use an annotation library. Instead of blindly replacing the expressions with new nondeterministic variables, we rely on a library of possible replacements for the operators and expressions in the program. The intuition is that, for a particular kind of error, programmers typically make the same kind of mistakes. For instance, a majority of the array out-of-bound violations are typically caused by one of the following kinds of errors: (1) using the $\leq$ operator instead of $<$ operator, i.e., off-by-one errors, (2) using an incorrect variable as the upper bound for an index, such as using $i < m$ instead of $i < n$, and (3) errors caused by copy-paste operations. An example of the annotation library is as follows:

Operator	Alternatives (weight)
$\leq$	$<$ (30), $\geq$ (20), $>$ (10)
$<$	$\leq$ (30), $>$ (20), $\geq$ (10)
$>$	$\geq$ (30), $<$ (20), $\leq$ (10)
$\geq$	$>$ (30), $\leq$ (20), $<$ (10)

The numbers in the parenthesis are weights that refer to the relative preference among the alternatives. The operator with higher weight is preferred over another operator with lower weight. For instance, $<$ is the most preferred alternative for the $\leq$ operator. Suppose that annotation library given above is used, the condition `i <= n` at step k of Fig. 3 would be replaced with the condition

```
 (ndSel_k == 3)?(i < n):
((ndSel_k == 2)?(i >= n):
((ndSel_k == 1)?(i > n):(i <= n)))
```

instead of `ndSel_k?(ndRes_k):(i <= n)`.

In our current system, an expert manually populates the annotation library based on the knowledge of the problem domain. It is also possible to create this library automatically by using machine learning or data mining techniques on the information from CVS logs or fixes made by the programmer for other bugs.

Using the weighted max-sat algorithm When an annotation library is provided, we use the weighted max-sat algorithm (implemented in SMT solver as such [7]) to determine if $M \wedge P \wedge D$ is satisfiable at step 9 of the error localization algorithm shown in Fig. 5. The weights provided in the annotation library are used as weights in the max-sat algorithm for the constraints that choose an alternative. For instance, using the annotation library shown earlier, weight 30 is assigned to the constraint `ndSel_k` $= 3$, weight 20 is assigned to the constraint `ndSel_k` ≥ 2, and weight 10 is assigned to the constraint `ndSel_k` > 3. With these weights, the SMT solver is more likely to find a solution that satisfies `ndSel_k` $= 3$. Therefore, an operator is replaced with its most preferred alternative.

4.3. Exploiting Correlation

Repeating Statements A repair program is generated from an unrolled error trace, which has multiple copies of the statements in a loop. Therefore, a repair program may have multiple copies of a statement that occurs in a loop. In the scheme described in §3, a different non-deterministic selector variable is used for every statement. Therefore, the error localization algorithm may provide repair solutions that make changes to the such statements inconsistently. For instance, the algorithm provides a solution in which the statement `i = i + 1` is changed in one step of the error trace, but not in another step. Such repair solutions are not useful because a change to the semantics of a statement in a loop has to be applied consistently across all executions steps of the statement in the loop. Therefore, we add constraints so that the SMT solver chooses a consistent value for the non-deterministic selector variables associated with the statements that are repeated in the trace.

For the repair program in Fig. 4, we add the following constraints:

$$(\texttt{ndSel2} = \texttt{ndSel4} \ldots = \texttt{ndSel22}) \wedge$$
$$(\texttt{ndSel3} = \texttt{ndSel5} \ldots = \texttt{ndSel21})$$

(Note, in our implementation, we actually simplify and propagate the equalities to reduce the formula size.) However, we cannot add similar constraints for the result variables of the statements in a loop, because the result of the computation at each loop step can be different.

Use-Def Chains Consider the following repair program (comments show the use-def chain in the original program):

```
x = ndSel1?ndRes1:e; // x = e;
...
y = ndSel2?ndRes2:x; // y = x;
...
```

As a repair solution `ndSel1`=*true* and `ndSel2`=*true*, does not propagate the repair effect of `x` to `y`, we add the constraint `ndSel1`=*true* $\Rightarrow$ `ndSel2`=*false* to avoid such redundant repair solution.

4.4. Other Improvements

Multiple error traces for a single error We use multiple error traces to improve error localization. A bug is typically manifested as multiple error traces for violations of one or more safety checkers. Consider the following simple C program fragment:

```
N1:  x = 0;
if(...) {N2:  x = x + 2;}
else    {N3:  x = x + 3;}
N4:if(x > 1) ERR();
```

The above program reaches an error state, because the condition `x` > 1 at `N4` is always satisfied. The model checker provides two error traces: (1) `N1` $\rightarrow$ `N2` $\rightarrow$ `N4` and (2) `N1` $\rightarrow$ `N3` $\rightarrow$ `N4`. If error trace (1) is examined in isolation, the error localization algorithm provides a solution that suggests that either `N1` or `N2` or both need to be fixed. Similarly, if error trace (2) is examined in isolation, the error localization algorithm provides a solution that suggests that either `N1` or `N3` or both need to be fixed. In either case, changing `N1` is the best fix because it fixes both the error traces. But, it is not possible to arrive at this conclusion by examining the error traces in isolation. Taking the intersection of the fixes suggested by the error localization algorithm for the different error traces gives `N1` as the only fix. Therefore, if we have multiple error traces, we just take the intersection of the fixes for each error trace.

Limiting the number of changes Typically, it would only require a few changes to the program to fix the error. We leverage this observation when finding repair solutions. We add constraints that bound the number of selector variables that can be assigned *true* by the solver. By adding such constraints, the error localization algorithm may be directed to find solutions that only require a minimal number of changes to the original program.

5. Ranking the Repair Solutions

The error localization algorithm provides several repair solutions to avoid the error. However, all the fixes provided by the tool may not be relevant to the error. Therefore, we use a simple ranking mechanism to prioritize the repair solutions.

First, the repair solutions are sorted by the number of steps in the execution of the repair program using the assignments from the repair solutions. Recall that the error localization algorithm provides repair solutions that skip to the `END` statement at any of the branches in the repair program. This criterion gives preference to the repair solutions that do not skip large parts of the repair program.

After sorting on the number of steps, the repair solutions are sorted by the number of fixes suggested by error local-

ization algorithm. The intuition behind this criterion is that the programmer would prefer to look at lesser number of error sites when debugging.

Finally, the repair solutions are sorted by the number of fixes to non-loop statements. That is, repair solutions that have more fixes in non-loop statements are given higher preference. The idea behind this criterion is that the fixes to non-loop statements change the semantics of only fewer steps in the program. However, the fixes to loop statements change the semantics of several steps in the program.

These ranking criteria were obtained based on our experience with using PED on a set of publicly available benchmarks. While the criteria are good enough for our set of benchmarks, it may not necessarily be good for other applications. The ranking scheme can be generalized by adapting ranking methods that are based on statistical analysis and user feedback [18].

6. Experiments

To evaluate the effectiveness of the error localization algorithm, we ran the algorithm on a collection of programs from the Verisec benchmark suite [19]. The Verisec benchmark suite is a collection of programs that include functions extracted from popular open source programs with known buffer-overrun vulnerabilities. The statements in the program that cause the buffer overflow are also known. We ran the error localization algorithm with different settings on the benchmark programs to evaluate the usefulness of the improvements described in the paper. For the experiments, the maximum number of fixes reported by the tool was set to 250, and an annotation library was also used. We modified the SMT-based BMC framework [9] to generate an SMT formula from the repair program, and used the YICES SMT solver [7] (version 1.0.10) as the backend constraint solver in our PED tool. We used a workstation with Intel QuadCore 2.4GHz, 8GB of RAM running Linux.

Tab. 1 shows the results of the experiments. The column labeled "Default" refers to the basic error localization algorithm described in §3. The column labeled "No Inv" refers to the algorithm with only the improvements from §4.3 and §4.4. The column labeled "With Inv" refers to the algorithm with the invariants extracted using the results of static analysis as described in §4.1 along with the improvements used for "No Inv". The column labeled "Relv. Stmnt." refers to the run of the algorithm with the information about the relevant statements obtained from static analysis as described in §4.1 along with the improvements used for "With Inv". The column "#F" represents the number of fixes reported by the tool. The column labeled "#U" represents the number of fixes that included the known error site. Effectively, the "#U" column provides a way to evaluate the usefulness of the reported fix because if a given fixes includes the known error site then the programmer will find it useful when debugging.

The results are encouraging. When the improvements described in §4.3 and §4.4 are used, the number of fixes reported by the tool is substantially reduced. The number of fixes reported by the tool *without* the improvements is *2* to 3 *times* the number of fixes reported with the improvements. The advantage of having a lesser number of fixes is that the user of the tool only has to look a smaller number of fixes to identify the root cause of the bug.

Similarly, the number of fixes reported by the tool is substantially reduced if the relevant invariants extracted from static analysis is used to add constraints on the non-deterministic result variables. Also, when the information about relevant statements is used, the number of fixes is reduced to less than one-third the number of fixes reported without the information about relevant statements. (The examples for which "Relv. Stmnt." shows improvements over "With Inv" is highlighted in bold in Tab. 1.)

We see a similar trend in the number of useful fixes reported by the tool. When the improvements are not used, the useful fixes reported by the tool include the error site as well as other statements that are not a root cause for the error, thereby, making it hard for the programmer to use the reported fix. However, when the improvements described in §4.1 through §4.4 are used, the error localization algorithm reports the most relevant fixes that only contain the error site, thereby, making it more useful for the programmer in debugging. Further, we found that the ranking scheme described in §5 is effective for our examples. When we examined the fixes provided by the tool for the configuration "Relv. Stmnt.", we found that the buggy statement was reported in one of the first five fixes.

The column labeled "#T(s)" represents the time taken (in seconds) by the error localization algorithm to find the fixes for a given error trace. We do not include the time necessary for static analysis and model checking in time reported under "#T" because any error diagnosis tool would incur the cost of static analysis and model checking to find the error traces. Similarly, the time to compute the safety projections and error projections is not included because they are computed during static analysis in our F-SOFT framework [1]. (For our examples, the time required for static analysis is in the order of a few seconds.) The time required by the solver to find the fixes is the only additional time required for error diagnosis. Further, the time to perform error localization is only dependent on the length of the error trace, which is a major advantage of our method as opposed to previous methods [2, 11, 12] that perform additional model checking on the whole program to find the root causes of an error. Therefore, we expect our technique to scale to larger programs.

Table 1. Results of error localization on Verisec benchmarks. #L denotes the length of the error trace. "LOC" refers to the number of lines of code. #F represents the number of fixes. #U represents the number of fixes that included the error site. #T denotes the time taken in seconds.

Program				Default			No Inv			With Inv			Relv. Stmnt.		
		LOC	Error (#L)	#F	#U	#T (s)	#F	#U	#T (s)	#F	#U	#T (s)	#F	#U	#T(s)
NetBSD-libc	Ex0	167	Err0(246)	219	114	282.16	64	20	237.38	46	21	224.69	**15**	**4**	232.18
(CVE-2006-6652)			Err1 (83)	232	124	72.56	59	21	405.22	17	5	92.88	**17**	**4**	93.07
			Err2 (83)	232	124	42.70	36	17	65.20	17	6	63.04	17	6	63.13
			Err3 (72)	210	110	7.94	34	17	4.17	21	8	2.39	**20**	**4**	3.30
	Ex1	159	Err0(59)	69	37	3.30	29	12	4.18	25	9	1.93	**17**	**7**	2.35
			Err1 (73)	97	52	5.47	33	12	5.72	30	11	2.88	**19**	**7**	3.54
			Err2 (200)	129	66	209.62	21	7	196.21	21	7	10.33	**11**	**3**	12.95
	Ex2	49	Err0 (26)	4	2	<1	4	2	<1	4	2	<1	4	2	<1
Apache	Ex0	111	Err0 (54)	85	48	4.06	67	33	5.00	43	21	33.67	43	21	33.67
(CVE-2004-0940)			Err1 (50)	85	48	3.79	67	33	5.20	43	21	33.39	43	21	33.39
			Err2 (51)	107	61	4.43	78	38	5.45	43	21	33.00	43	21	33.66
	Ex1	105	Err0 (47)	15	8	32.93	15	8	34.02	14	8	3.10	14	8	3.13
			Err1 (50)	21	12	3.27	21	12	5.09	20	12	3.38	20	12	3.42
	Ex2	117	Err0 (54)	85	48	4.89	67	33	5.82	40	18	34.06	40	18	34.10
			Err1 (60)	107	76	5.59	45	22	6.16	40	18	34.25	40	18	34.00
	Ex3	109	Err0 (47)	15	8	33.65	15	8	35.13	14	8	3.81	14	8	3.72
			Err1 (56)	21	12	3.82	21	12	5.39	20	12	3.92	20	12	3.75
Sendmail	Ex0	40	Err0 (21)	7	7	1.17	7	7	1.17	7	7	1.19	7	7	1.23
(CVE-1999-0047)	Ex1	33	Err0 (17)	3	3	<1.00	3	3	<1.00	3	3	<1.00	3	3	<1.00
Sendmail	Ex0	59	Err0 (30)	22	13	2.69	16	9	3.48	7	2	32.52	7	2	32.48
(CVE-1999-0206)			Err1 (35)	24	14	3.27	16	8	3.76	7	2	33.08	7	2	33.08
	Ex1	69	Err0 (33)	18	10	3.07	12	5	4.15	5	1	32.88	5	1	32.86
			Err1 (25)	21	15	2.14	18	12	2.53	12	8	2.09	12	8	2.07
			Err2 (38)	36	26	3.62	10	4	4.77	7	2	33.12	7	2	33.18
SpamAssasin	Ex0	85	Err0 (49)	102	102	7.3	48	10	7.43	36	36	5.66	**27**	**27**	6.09

7. Related Work

We briefly review the most related work. In model-checking based methods [2, 12, 11], the correct traces are obtained by re-executing the model checker with additional constraints. Similarities and differences in correct traces (also referred to as positives) and erroneous traces (negatives) are analyzed transition by transition to obtain the root causes. These methods are in general limited by the scalability of the model checker. Further, the differences between positive and negative traces do not always provide a good explanation of the failure.

In program repair approaches and error correction [10, 16, 24], fault localization is achieved by introducing non-deterministic repair solutions in a modified system, and using a model checker to obtain a set of possible causes for the symptoms. Such an approach has been successful for hardware. However, they fail to pinpoint the real causes in software due to the presence of a large number of repair solutions.

In another work based on static analysis [26], path-based syntactic-level weakest pre-condition computation is used to obtain the minimum proof of infeasibility for the given error trace. This method does not require correct trace, and does not use expensive model checking. This causal analysis provides an infection chain of the defect (i.e., relevant statements through which the defect in the code propagates), and not necessarily the root cause of the error.

Test-based error diagnosis methods [17, 22] rely on availability of good test-suite with large successful executions. The error traces are compared with the correct traces to pin-point the possible causes of the failure.

Delta debugging [4, 28] is an automatic experimental method to isolate failure causes. It requires two programs runs, one run where the failure occurs, and another where it does not. The subset of differences between the two is applied on the non-erroneous run to obtain the failure run. Such differences are then classified as causes of the problem. This method is purely empirical, and is different from formal or static analysis. Also, it may require a large number of tests to find a difference that pinpoints the error-site.

In game-theoretic based approaches [15], error trace is partitioned into two segments "fated" and "free": "fated" controlled by the environment forcing the system to error, and "free" controlled by system to avoid the error. Fated segments manifest unavoidable progress towards the error while free segments contain choices that, if avoided, can prevent the error. This approach is significantly more costly than a standard model checking.

8. Conclusions

We proposed a repair-based proof-guided error diagnosis framework. Our techniques improve existing approaches

by taking into account information, such as relevant invariants and relevant statements from proofs obtained during static analysis, syntactic closeness of operators, correlation among statements, and multiple error traces to improve error localization. We implemented the technique and integrated it as a plug-in module to the F-SOFT verification framework. Our experiments show that such an approach improves the quality of error diagnosis.

References

[1] G. Balakrishnan, S. Sankaranarayanan, F. Ivancic, O. Wei, and A. Gupta. SLR: Path-sensitive analysis through infeasible-path detection and syntactic-language refinement. In *Proc. Static Analysis Symposium (SAS)*, 2008.

[2] T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: localizing errors in counterexample traces. In *POPL*, 2003.

[3] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[4] H. Cleve and A. Zeller. Locating causes of program failures. In *Int. Conf. on Softw. Eng. (ICSE)*, pages 342–351, 2005.

[5] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *POPL*, 1977.

[6] P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In E. Neuhold, editor, *Formal Descriptions of Programming Concepts, (IFIP WG 2.2, St. Andrews, Canada, August 1977)*, pages 237–277. North-Holland, 1978.

[7] B. Dutertre and L. de Moura. The YICES SMT solver. http://yices.csl.sri.com/tool-paper.pdf.

[8] B. Dutertre and L. de Moura. A fast linear-arithmetic solver for DPLL(T). In *CAV*, 2006.

[9] M. K. Ganai and A. Gupta. Acceleration high-level bounded model checking. In *Int. Conf. on Computer-Aided Design (ICCAD)*, 2006.

[10] A. Griesmayer, R. Bloem, and B. Cook. Repair of boolean programs with an application to C. In *CAV*, 2006.

[11] A. Groce, S. Chaki, D. Kroening, and O. Strichman. Error explanation with distance metrics. *Softw. Tools for Tech. Transfer*, 8(3):229–247, 2005.

[12] A. Groce and W. Visser. What went wrong: Explaining counterexamples. In *Spin Workshop*, pages 121–135, 2003.

[13] N. Halbwachs, Y.-E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, 1997.

[14] F. Ivančić, I. Shlyakhter, A. Gupta, M. K. Ganai, V. Kahlon, C. Wang, and Z. Yang. Model checking C programs using F-SOFT. In *ICCD*, pages 297–308, 2005.

[15] H. Jin, K. Ravi, and F. Somenzi. Fate and free will in error traces. In *TACAS*, pages 445–459, 2002.

[16] B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In *CAV*, pages 226–238, 2005.

[17] J. A. Jones, M. J. Harrold, and J. T. Stasko. Visualization of test information to assist fault localization. In *Int. Conf. on Softw. Eng. (ICSE)*, pages 467–477, 2002.

[18] T. Kremenek and D. Engler. Z-Ranking: Using statistical analysis to counter the impact of static analysis approximations. In *Proc. Static Analysis Symposium (SAS)*, 2003.

[19] K. Ku. The verisec buffer overflow benchmark. http://www.cs.toronto.edu/ kelvin/benchmark/.

[20] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *Trans. on Softw. Eng.*, 32(3):176–192, 2006.

[21] A. Miné. The octagon abstract domain. In *8th Working Conf. on Rev. Eng.*, pages 310–322, 2001.

[22] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *ASE*, pages 30–39, 2003.

[23] S. Sankaranarayanan, F. Ivancic, I. Shlyakhter, and A. Gupta. Static analysis in disjunctive numerical domains. In *SAS*, pages 3–17, 2006.

[24] S. Staber, B. Jobstmann, and R. Bloem. Finding and fixing faults. In *CHARME*, pages 35–49, 2005.

[25] M. Stumptner and F. Wotawa. A survey of intelligent debugging. In *AI Communications*, 1998.

[26] C. Wang, Z. Yang, F. Ivancic, and A. Gupta. Whodunit? causal analysis for counterexamples. In *ATVA*, 2006.

[27] M. Weiser. Program slicing. *Trans. on Softw. Eng.*, SE-10(4):352–357, July 1984.

[28] A. Zeller. Isolating cause-effect chains from computer programs. In *Found. of Softw. Eng.*, pages 1–10, 2002.