# DARWIN: An Approach to Debugging Evolving Programs

DAWEI QI, ABHIK ROYCHOUDHURY, and ZHENKAI LIANG, National University of Singapore
KAPIL VASWANI, Microsoft Research India

Bugs in programs are often introduced when programs evolve from a stable version to a new version. In this article, we propose a new approach called *DARWIN* for automatically finding potential root causes of such bugs. Given two programs—a reference program and a modified program—and an input that fails on the modified program, our approach uses symbolic execution to automatically synthesize a new input that (a) is very similar to the failing input and (b) does not fail. We find the potential cause(s) of failure by comparing control-flow behavior of the passing and failing inputs and identifying code fragments where the control flows diverge.

A notable feature of our approach is that it handles hard-to-explain bugs, like code missing errors, by pointing to code in the reference program. We have implemented this approach and conducted experiments using several real-world applications, such as the Apache Web server, libPNG (a library for manipulating PNG images), and TCPflow (a program for displaying data sent through TCP connections). In each of these applications, *DARWIN* was able to localize bugs with high accuracy. Even though these applications contain several thousands of lines of code, *DARWIN* could usually narrow down the potential root cause(s) to less than ten lines. In addition, we find that the inputs synthesized by *DARWIN* provide additional value by revealing other undiscovered errors.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids, symbolic execution*; D.3.4 [**Programming Languages**]: Processors—*Debuggers*

General Terms: Experimentation, Reliability

Additional Key Words and Phrases: Software debugging, software evolution, symbolic execution

**ACM Reference Format:**
Qi, D., Roychoudhury, A., Liang, Z., and Vaswani, K. 2012. DARWIN: An approach to debugging evolving programs. ACM Trans. Softw. Eng. Methodol. 21, 3, Article 19 (June 2012), 29 pages.
DOI = 10.1145/2211616.2211622 http://doi.acm.org/10.1145/2211616.2211622



## 1. INTRODUCTION

The development of any large-scale software system is a gradual process. Starting from an initial design, the system evolves as new features are introduced, the system is optimized, and defects are fixed. Often, changes are made concurrently by a number of developers. It is during such changes that subtle defects are introduced. As a result, ensuring that the system continues to meet its requirements in the presence of such

An initial version of this article [Qi et al. 2009] was published in the *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE).*
This work was partially supported by a Ministry of Education research grant MOE2010-T2-2-073 (R-252-000-456-112 and R-252-100-456-112) from Singapore.
Authors' addresses: D. Qi and A. Roychoudhury (was on sabbatical leave to Microsoft Research India during part of this work), and Z. Liang, School of Computing, National University of Singapore, Computing 1, 13 Computing Drive, Singapore 117417; email: {dawei, abhik, liangzk}@comp.nus.edu.sg; K. Vaswani, Microsoft Research India, 196/36, 2nd Main, Sadashivnagar, Bangalore 560080, India; email: kapilv@microsoft.com.

DOI 10.1145/2211616.2211622 http://doi.acm.org/10.1145/2211616.2211622

changes is a huge problem. The effort spent in validating software as it evolves accounts for a large fraction of the overall maintenance costs, which often ends up being much larger than the cost of development. The cost of maintaining a software system and managing its evolution is said to account for more than 90% of the total cost, prompting authors to call it the "legacy crisis" [Seacord et al. 2003].

To tackle the ever-growing problem of software evolution and maintenance, software testing methodologies have been studied extensively. Regression testing is a well-known concept employed in most software development projects. In its simplest form, it involves re-testing a test suite as a program changes from one version to another. In the past, the problem of detecting which tests in a given test suite do not need to be re-tested has been thoroughly studied [Chen et al. 1994]. However, even among the tests which are tested in both the old and the new program versions, how do we find the root cause of a failed test input? For any large software development project, finding root causes of these regression bugs is a challenging problem.

*Problem Statement.* The problem we address can be summarized as follows. Consider a program $P$ accompanied by a test suite $T$, such that the observable output of $P$ for all the tests in $T$ is as expected by the programmer, that is, all the tests pass. We call $P$ the *stable* or *reference* program. Suppose $P$ changes to a new program $P'$ and certain tests in $T$ now fail. Let $t \in T$ be such a test. Our goal is to identify code fragments that potentially explain why $t$ fails in $P'$, while passing in $P$. Of course, we would like to identify as few code fragments as possible, while still localizing the cause of failures with high accuracy.

*Existing Solutions.* To motivate our solution, we first discuss the difficulties in using existing approaches to solve this problem.

—*Differencing Methods.* Program differencing methods (e.g., Horowitz [1990]) have been proposed as a way of identifying semantic differences between program versions by comparing their program dependence graphs. Since we are investigating the behavior of a specific test case in two program versions, we cannot directly use these methods. Interestingly, our conversations with development teams revealed that they often perform differencing of traces (not programs) for finding root causes of regression bugs. Given a test $t$ which passes in program $P$ and fails in program $P'$, one may compare the path traced by $t$ in $P$ vis-a-vis the path traced by $t$ in $P'$. However, a structural comparison of paths of two different programs $P$ and $P'$ is likely to be ineffective, because it does not explicitly consider the semantics of the changes between $P$ and $P'$.

—*Change Inspection.* If we assume that defects are often introduced as part of changes, one way for finding the root cause is to find the specific change that induces failure (e.g., Zeller [1999]). While this approach is appealing, it is ineffective for a class of bugs known as *unmasking regressions*. These are bugs that already existed in the reference version of the program but are exposed by the change. For example, a pointer which is mistakenly set to null in the reference version but never de-referenced is indicative of such a situation. The mistake may only be observed after a change which introduces a pointer de-reference. An accurate root-cause analysis tool should isolate the location where the pointer is mistakenly set to null, not the location where it is de-referenced. Moreover, a search for failure-inducing changes will not work if $P$ and $P'$ are wildly different implementations (say two Web server implementations both implementing the HTTP protocol), since then the set of program changes from $P$ to $P'$ is hard to enumerate.

—*Trace Comparison.* In the last decade, trace comparison methods have been successfully used for localizing error causes in programs. Given a buggy program $P'$, the

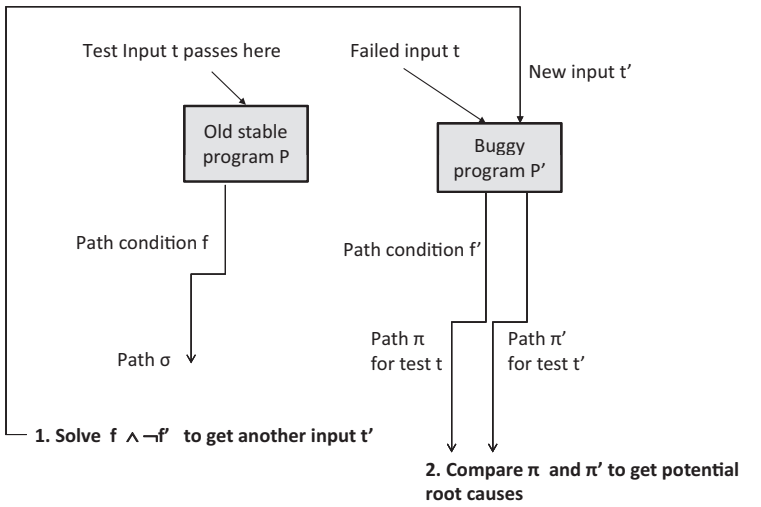


Fig. 1. Rough description of the debugging method.

trace produced by a failed input $t$ is compared to the trace produced by a passing input $t'$. Techniques have been developed to determine (a) which passing input to use ( e.g., Guo et al. [2006]), and (b) how to compare and report the differences between two program executions (e.g., Zeller [2002]). The effectiveness of these approaches depends critically on the availability of a passing input $t'$ that is very similar to the failing input $t$.

*Our Approach.* In this article, we propose an approach (called *DARWIN*) for automatically finding root-causes regression failures. A pictorial description of our approach appears in Figure 1. In the sequel, we use the term *test* and *input* interchangeably. Given a reference program $P$, a buggy program $P'$, and a input $t$ which passes in $P$ and fails in $P'$, we first synthesize a new input $t'$ satisfying the following properties: (i) $t'$ and $t$ follow the same program path in $P$, and (ii) $t'$ and $t$ follow different program paths in $P'$. Such an input $t'$ can be found using a combination of concolic execution [Godefroid et al. 2005] and constraint solving [Brummayer and Biere 2009; de Moura and Bjorner 2008; Barrett and Tinelli 2007; Bruttomesso et al. 2008; Ganesh and Dill 2007]. We then compare the trace produced by $t$ in $P'$ with the trace produced by $t'$ in $P'$. Since $t'$ and $t$ follow the same program path in $P$, we say that $t$ and $t'$ are similar (with respect to the reference program $P$). However, since $t$ and $t'$ follow different program paths in $P'$, their behavior differs in $P'$ (the buggy new version). The key insight of our approach is that the difference in behavior of $t$ and $t'$ in the buggy program often indicates the potential cause(s) of failure.

However, as we will describe later, because of the way we generate the alternative input, trace comparison is not strictly necessary. From the input generation phase itself, our method will know where the traces of $t$ and $t'$ will differ, and these differences can constitute the potential root causes without going through trace comparison. The main advantage of such an approach is that we avoid any heuristics in the trace comparison. Our method is thus based completely on construction and solving of quantifier-free first-order logic formulas.

*Contributions.* We propose *DARWIN*, an automated and scalable solution to the problem of locating causes of regression bugs. The method uses symbolic execution (on

```
1  int x, y;// x and y are both input variables
2  int o; // o is the output variable
3  scanf("%d",&x);
4  scanf("%d",&y);
5  if (x > 0){
6    y = y +1;
7    if (y > 0){
8      o = 10;
9    }else{
10     o = 20;
11   }
12 }else{
13   o = 30;
14 }
```

Fig. 2. An example program which is used to illustrate path condition computation.

both failing/passing program versions) to produce alternate inputs, and compares the behavior of such alternate inputs with the failing input in the buggy program. We demonstrate the efficacy of our approach using four real-world applications (libPNG, TCPflow, miniWeb, and Apache). Further, we find that the alternate inputs generated by our method can be used for purposes other than localizing a given observable error. These alternate inputs can point to new undiscovered errors, as demonstrated by our experiments.

## 2. BACKGROUND

Path condition [Godefroid et al. 2005] serves as the basis of our approach. The computation of path condition is critical to understanding many aspects of our approach. In the following, we give background on path conditions and their calculation.

### 2.1. Path Condition

Consider a program $P$, a program input $t$, and the path $\pi_t$ executed by $t$ in $P$. The path condition of $t$ in program $P$ is a quantifier-free first-order logic formula $\varphi_t$ over input variables of $P$, such that any input satisfying $\varphi_t$ follows the path $\pi_t$ in program $P$, and any input not satisfying $\varphi_t$ does not follow the path $\pi_t$ in program $P$.

Thus, the path condition of a given path serves as the precise logical characterization of the set of inputs tracing the path. Needless to say, if a path is infeasible (no input can trace it), its path condition is the logical formula $false$. Similarly, if all program inputs trace the same path, its path condition is the logical formula $true$.

### 2.2. Example of Path Condition Calculation

Next, we use an example to illustrate the process of computing a path condition. The example program is shown in Figure 2. We use input $\langle x == 1, y == 1\rangle$ as an example to show how a path condition is computed. We use $x^s$ and $y^s$ to denote the symbolic inputs of this program.

The computation is shown in Table I. After executing each line, we show the concrete stores and the symbolic stores of the variables. In the last column, we show the path condition gathered up to the corresponding line. If a conditional branch is executed, the generated branch constraint is accumulated into the path condition, as shown in the last column.

For example, after line 6 is executed, the accumulated path condition is $(x^s > 0)$. Since line 7 is a conditional branch, the branch constraint $(y^s + 1 > 0)$ is generated and added into the path condition. So after executing line 7, the path condition becomes

Table I. Process of Computing Path Condition for the Program in Figure 2

| Execution trace of $\langle x == 1, y == 1 \rangle$ | Concrete stores | Symbolic stores | Path condition |
|---|---|---|---|
| 3 $scanf(\text{“}\%d\text{”}, \&x);$ | $\{x \to 1, y \to undef\}$ | $\{x \to x^s, y \to undef\}$ | $true$ |
| 4 $scanf(\text{“}\%d\text{”}, \&y);$ | $\{x \to 1, y \to 1\}$ | $\{x \to x^s, y \to y^s\}$ | $true$ |
| 5 $if(x > 0)\{$ | $\{x \to 1, y \to 1\}$ | $\{x \to x^s, y \to y^s\}$ | $(x^s > 0)$ |
| 6 $y = y + 1;$ | $\{x \to 1, y \to 2\}$ | $\{x \to x^s, y \to y^s + 1\}$ | $(x^s > 0)$ |
| 7 $if(y > 0)\{$ | $\{x \to 1, y \to 2\}$ | $\{x \to x^s, y \to y^s + 1\}$ | $(x^s > 0) \wedge (y^s + 1 > 0)$ |
| 8 $o = 10;$ | $\{x \to 1, y \to 2\}$ | $\{x \to x^s, y \to y^s + 1\}$ | $(x^s > 0) \wedge (y^s + 1 > 0)$ |

$(x^s > 0) \wedge (y^s + 1 > 0)$. The final path condition is simply the conjunction of all the branch constraints. In this example, two branch constraints $(x^s > 0)$ and $(y^s + 1 > 0)$ are generated from lines 5 and line 7, respectively. Taking the conjunction of the two branch constraints, the final path condition is simply $pc = (x^s > 0) \wedge (y^s + 1 > 0)$. The path condition computed in this way only contains input variables. The path condition can guarantee that any input satisfying the path condition will follow the same path as $\langle x == 1, y == 1 \rangle$.

Although path condition is a conjunction of branch constraints, the assignments executed in the trace are also taken into consideration in the path condition. As we can see in the example, the assignment in line 6 is considered when computing path condition. The assignment in line 6 first affects the symbolic store of $y$. When $y$ is used in the condition in line 7, the symbolic store of $y$ is used to compose the branch constraints. If there is an error in line 6, the error can affect the branch constraint generated in line 7 and, therefore, affect the path condition.

### 2.3. Mechanism for Computing Path Conditions

Overall, the path condition is computed through symbolic execution. During symbolic execution, we interpret each statement and update the symbolic store to represent the effects of the statement on program variables.

Note that a path is given by a sequence of program statements in the source code. First, we transform the source code such that every statement is either an assignment or a branch statement. Then, we traverse forward along the sequence of statements in the given path, starting with a null formula and gradually building it up.

At any point during the traversal of the path, we maintain a set of symbolic expressions for the symbolic store and a logical formula for the path condition. During the forward traversal, we update the symbolic store and path condition as follows.

—For every assignment encountered, we only update the symbolic assignment store.
—For every branch statement encountered, we conjoin the branch condition with the path formula. While doing so, we use the symbolic assignment store for every variable appearing in the branch condition.

The formula thus obtained upon reaching the end of the path is the path condition.

## 3. OVERALL APPROACH

In this section, we first present an overview of our approach using an illustrative example. Consider a program fragment $P$ (Figure 3) with an integer input variable `inp`. We assume that $P$ is the stable reference program where all test cases pass. Note that `g` and `h` are functions invoked from $P$. The code for `g` and `h` is not essential to understanding the example.

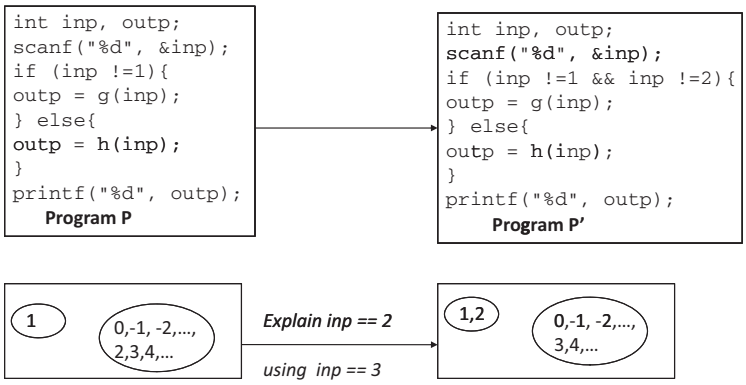


Fig. 3. Two example programs $P$ and $P'$ and their input-space partitioning. The behavior of input 2 changes during the change $P \rightarrow P'$. We choose an input 3 to explain the behavior of the failing input 2, since 2 and 3 are in the same partition in $P$ but in different partitions in $P'$.

Suppose the program $P$ is changed to the program $P'$, shown in Figure 3, thereby introducing a bug. Due to this bug, certain inputs which passed in $P$ may fail in $P'$. One such input is `inp == 2`, whose behavior is changed from $P$ to $P'$. Let us assume this input is found during regression testing, and we now want to localize the cause for failure. Our approach works as follows.

—We symbolically execute the program $P$ for test input `inp == 2` and derive a *path condition* $f$, a formula representing the set of inputs which exercise the same path as `inp == 2` in program $P$. In our example, path condition $f$ is $inp \neq 1$.
—We symbolically execute the program $P'$ for input `inp == 2` and calculate the *path condition* $f'$, a formula representing the set of inputs which exercise the same path as `inp == 2` in program $P'$. In our example, path condition $f'$ is $(inp \neq 1 \wedge inp == 2)$.
—We solve the formula $f \wedge \neg f'$. By construction, any satisfying instance of the formula is an input which follows the same path as the failing input `inp == 2` in the reference program $P$ but follows a different path than failing input in the new program $P'$. In our example, $f \wedge \neg f'$ is

$$(inp \neq 1 \wedge \neg(inp \neq 1 \wedge inp == 2)) \equiv (inp \neq 1 \wedge inp \neq 2).$$

A solution to this formula is any value of `inp` other than 1 or 2, say `inp == 3`.
—We could compare the trace of `inp == 3` in buggy program $P'$ with the trace of the failing input `inp == 2` in $P'$. Instead, we show how trace comparison could be avoided and could be replaced by formula manipulation instead. During the process of finding a satisfying instance for the formula $f \wedge \neg f'$, we find that $\neg f'$ is equivalent to $\neg(inp \neq 1 \wedge inp == 2)$ that is, $inp == 1 \vee inp \neq 2$. These are the possible *deviations* from $f'$. The first deviation, when conjoined with $f$, produces $inp \neq 1 \wedge inp == 1$, which is unsatisfiable. The second deviation, when conjoined with $f$, produces $inp \neq 1 \wedge inp \neq 2$. Hence we highlight the source-code location corresponding to this constraint as the potential reason for the input `inp == 2` failing in program $P'$.

In general, in trying to derive the potential cause by solving $f \wedge \neg f'$, we note that $f'$ is a conjunction of primitive constraints, say $f' \equiv \psi_1 \wedge \psi_2 \wedge \cdots \wedge \psi_m$. We then enumerate all possible deviations of $f'$, namely $\neg\psi_1$, $\psi_1 \wedge \neg\psi_2$, ..., $\psi_1 \wedge \psi_2 \wedge \cdots \wedge \psi_{m-1} \wedge \neg\psi_m$. We then conjoin each of these deviations with $f$, producing $m$ formulas (where $m$ is the number of primitive constraints in $f'$). For each of the $m$ formulas that are satisfiable, we consider the corresponding branch as a potential root cause. In other words, if

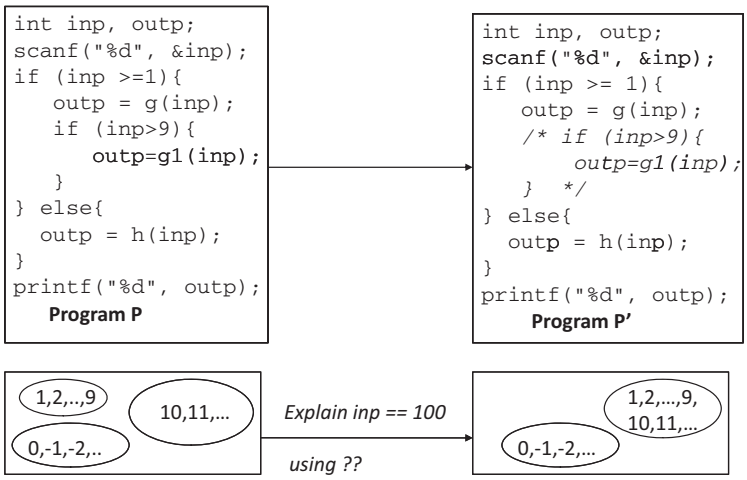


Fig. 4. Two example programs $P$ and $P'$ and their input-space partitioning. The behavior of input 100 changes during the change $P \rightarrow P'$. How to find an input to explain its behavior?

$f \wedge \psi_1 \wedge \psi_2 \wedge \ldots \psi_{i-1} \wedge \neg\psi_i$ is satisfiable, we consider the program branch contributing to the constraint $\psi_i$ as a potential root cause.

The example in Figure 3 clarifies the intuition behind our method. For the inputs common to $P$ and $P'$ (in this example, the two programs have exactly the same input space), we consider the partitioning of program inputs based on paths—-two inputs are in the same partition if and only if they follow the same path. Then, as $P$ changes to $P'$, certain inputs migrate from one partition to another. Figure 3 illustrates this partitioning and partition migration. The behavior of the failing input `inp == 2` is explained by `inp == 3`. The two inputs are in the same partition in the old program $P$ but in different input partitions in $P'$.

Sometimes, given two program versions $P$ and $P'$ and a failing input $t$, we may not find any alternate input by solving $f \wedge \neg f'$. Consider the example programs in Figure 4 and their associated input-space partitioning. In this case, we have a *code-missing error*; the code

```
if (inp > 9) {outp = g1(inp);},
```

is left out by mistake. Suppose we have the task of explaining the behavior of `inp == 100`.

The path condition $f$ of `inp == 100` in $P$ is $(inp \geq 1 \wedge inp > 9)$, that is, $inp > 9$. The path condition $f'$ of `inp == 100` in $P'$ is $inp \geq 1$. So, in this case,

$$f \wedge \neg f' \equiv (inp > 9 \wedge \neg(inp \geq 1 \wedge inp > 9)) \equiv (inp > 9 \wedge \neg(inp \geq 1)),$$

which is unsatisfiable! The reason is simple. All inputs sharing the same partition as that of `inp == 100` in the old program, also share the same partition with `inp == 100` in the new program.

The solution to this dilemma lies in focusing our debugging effort on the reference program. If we find that $f \wedge \neg f'$ is unsatisfiable, we can solve $f' \wedge \neg f$. This yields an input $t'$ which takes a different path than that of the failing input $t$ in the reference program.

In our example Figure 4, we have

$$f' \wedge \neg f \equiv (inp \geq 1 \wedge \neg(inp \geq 1 \wedge inp > 9)),$$

that is, $1 \leq inp \leq 9$. The solutions to this formula are the values $1, 2, \ldots, 9$ for the variable `inp`.

Once again, while solving $f' \wedge \neg f$, we enumerate the deviations from $f$ first. Since $f \equiv (inp \geq 1 \wedge inp > 9)$, the deviations from $f$ the following.

—$\neg inp \geq 1$ that is, $inp < 1$.
—$inp \geq 1 \wedge \neg inp > 9$ that is, $1 \leq inp < 9$.

The first deviation, when conjoined with $f'$, produces $inp \geq 1 \wedge inp < 1$, which is unsatisfiable. The second deviation, when conjoined with $f'$, is satisfiable. So, we consider the corresponding branch, namely $inp > 9$, as a potential root cause. Indeed, this branch is the check which was missing in the buggy program $P'$, points us the code-missing error in this example.

The reader may consider the preceding situation as odd. When a test fails in a buggy program, we may point to a fragment of the reference program as a potential root cause! But, indeed this is a key feature of our approach. Code fragments in the reference program often help the programmer comprehend the change from the reference program to the buggy program, thereby helping him/her comprehend the source of the failure.

In summary, the outline of our method is as follows. Given a reference program version $P$, a new, buggy program $P'$, and a test input $t$ which passes in $P$ and fails in $P'$, our method proceeds as follows.

(1) Compute $f$, the path condition of $t$ in $P$.
(2) Compute $f'$, the path condition of $t$ in $P'$.
(3) Check whether $f \wedge \neg f'$ is satisfiable. If yes, it yields a test input $t'$ as well as a constraint $\psi_i'$ in $f'$. The constraint $\psi_i'$ is the reason why $f'$ is not satisfied and is considered as a potential root cause. As we have explained, the constraint $\psi_i'$ are obtained by enumerating the deviations of $f'$ and conjoining them with $f$. Details of the procedure for obtaining $\psi_i'$ are describe in Section 4.
Since we perform approximations while computing the path conditions, we also check that the solution $t'$ indeed follows the same path as that of $t$ in $P$ and a different path from that of $t$ in $P'$. This is done by concrete execution of input $t'$.
(4) If $f \wedge \neg f'$ is unsatisfiable, find a solution to $f' \wedge \neg f$. This again produces an input $t'$ and a constraint $\psi_i$. The code to $\psi_i$ is considered a potential root cause.
We also check $t'$ against $f' \wedge \neg f$ (i.e., it follows the same path as that of test $t$ in program $P'$ and follows a different path in program $P$). This is done by concrete execution of test input $t'$.

In the event $(f \wedge \neg f') \vee (f' \wedge \neg f)$ is unsatisfiable, we fail to find a potential root cause. This situation is unlikely, since this would mean that $f \Leftrightarrow f'$ is valid, which means that the input partition of the failing input remains unchanged while going from the stable program to the buggy program. We also never encountered this situation in our experiments.

## 4. DETAILED METHODOLOGY

In this section, we elaborate on different aspects of our approach, that is, input generation, formula simplification, input validation, and finally bug reporting.

### 4.1. Generating Alternate Inputs

In this phase, we execute the failing input $t$ in both the program versions. We first concretely execute $t$ for each program binary, record a trace, and then perform symbolic execution on the recorded trace. Our symbolic execution engine models each byte of the program's input as a symbolic variable. For each program variable, the engine also

stores a symbolic formula over the input variables that represents the set of values that can be assigned to this variable in the concrete execution. This mapping between program variables and expressions represents the symbolic state.

We compute the path condition using the method explained in Section 2. Two path condition formulas $f$ and $f'$ are computed for input $t$ in $P$ and $P'$, respectively.

A key component of the symbolic execution engine is the constraint solver. The precision of symbolic execution depends, to a large extent, on the ability of the constraint solver to symbolically reason about computations in the program. For example, for a program branch `if (x * y > 0)`, we need to add the constraint `x * y > 0` to the path condition. This may be problematic if our constraint solver is a linear programming solver and does not reason about operations, such as multiplication. An approach commonly used by most symbolic execution engines [Godefroid et al. 2005] to overcome limitations of the constraint solver is to *under-approximate* the path condition. Usually, such an under-approximation is achieved by instantiating some of the variables in the actual path condition. For example, to keep the path condition as a linear formula, we may under-approximate the condition `x * y > 0` by instantiating either `x` or `y` with its value from concrete program execution.

Recall that we need to solve the formula $f \wedge \neg f'$ for getting an alternate programs input, where $f$ and $f'$ are the path conditions of the input $t$ being examined in the reference and buggy programs respectively. Let $f_{computed}$, $f'_{computed}$ be the computed path conditions in the reference and buggy programs, respectively. In general, the computed $f$ and $f'$ will be an under-approximation of the actual path conditions. Thus, $f_{computed} \Rightarrow f$, and $f'_{computed} \Rightarrow f'$. However, due to the negation, $f_{computed} \wedge \neg f'_{computed}$ is not guaranteed to be an under approximation of $f \wedge \neg f'$. Consequently, a solution to $f_{computed} \wedge \neg f'_{computed}$ may not satisfy the required properties, namely, $t$ and $t'$ follow the same program path in the reference program and follow different paths in the buggy program. Hence, after solving $f_{computed} \wedge \neg f'_{computed}$, if we find a solution $t'$, we *validate* $t'$. Such a validation can be performed by concretely executing the test inputs $t$ and $t'$ in the old and new program versions and checking if our criteria are satisfied. Similarly, if we need to solve the formula $f' \wedge \neg f$, we validate the test input obtained by solving $f' \wedge \neg f$.

*Choosing Alternate Inputs.* Note that since $f$ and $f'$ are path conditions, they are conjunctions of primitive constraints, that is, $f' = (\psi_1 \wedge \psi_2 \wedge \cdots \wedge \psi_m)$, where $\psi_i$ are primitive constraints. Thus, instead of solving $f \wedge \neg f'$, we solve the following $m$ formulas $\{\varphi_i \mid 0 \leq i < m\}$, where

$$\varphi_i \stackrel{def}{=} f \wedge \psi_1 \wedge \ldots \psi_i \wedge \neg\psi_{i+1}.$$

Each $\varphi_i$ is a conjunction. A solution to any $\varphi_i$ is a solution for $f \wedge \neg f'$. We solve each $\varphi_i$ separately and obtain any one solution of $\varphi_i$ (if one exists). Thus, we obtain at most $m$ solutions to the formula $f \wedge \neg f'$. Each of these are inputs which now undergo validation, that is, we check via concrete execution whether they follow the same path as that of $t$ in $P$ and follow a different path from that of $t$ in $P'$. The reader may note our choice of $\varphi_i$, the formulas dispatched to the solver. Each $\varphi_i$ denotes a deviation from the path condition $f'$ in exactly the $i^{th}$ branch condition of $f'$. Thus, any alternate input we get by solving $\varphi_i$ can be expected to produce a trace which differs from the trace of the buggy input in exactly the $i^{th}$ branch position. Moreover, by solving the different $\varphi_i$, we consider all possible ways of deviating from the path denoted by path condition $f'$. Thus, our alternate inputs are witnesses to deviations from the path denoted by path condition $f'$—one alternate input for each possible deviation point in the path. Finally, note that if $f \wedge \neg f'$ is unsatisfiable, we solve $f' \wedge \neg f$ similarly. Thus, if $f$ is

a conjunction of $k$ primitive constraints $\theta_i$, say $f = (\theta_1 \wedge \theta_2 \wedge \cdots \wedge \theta_k)$, we solve the $k$ formulas $f' \wedge \theta_1 \wedge \ldots \theta_i \wedge \neg\theta_{i+1}$, where $0 \leq i < k$.

### 4.2. Formula Simplification

A crucial component of our debugging method is the generation of alternate test inputs. This is achieved via checking satisfiability using satisfiability modulo theory (SMT) solvers. Thus, the scalability of our method depends on the scalability of formula solving. We propose several techniques for improving the efficiency of formula solving specific to our problem domain.

*Checking for Unsatisfiable Subformula.* Recall that we are trying to solve formulas of the form $f \wedge \neg f'$, where $f$ and $f'$ are the path conditions collected from two program versions for a given test input $t$. Assuming $f' \stackrel{def}{=} (\psi_1 \wedge \psi_2 \ldots \wedge \psi_m)$, we solve the $m$ formulas $\varphi_i \stackrel{def}{=} f \wedge \psi_1 \wedge \ldots \psi_i \wedge \neg\psi_{i+1}$. The key problem we face now is that the SMT solver may take substantial time to solve each of the $\varphi_i$ formula. We note that common programming practices may make $\psi_1 \wedge \ldots \psi_i \wedge \neg\psi_{i+1}$ unsatisfiable. For example, consider a check $c$ being repeated many times in a program code. Clearly, if $\psi_j$ (for some $j \leq i$) and $\psi_{i+1}$ are both $c$, an SMT solver would very quickly conclude that $\psi_1 \wedge \ldots \psi_i \wedge \neg\psi_{i+1}$ is unsatisfiable. In such a situation, we do not need to solve the larger formula $f \wedge \psi_1 \wedge \ldots \psi_i \wedge \neg\psi_{i+1}$. Overall, instead of directly dispatching $\varphi_i$ to the SMT solver (to check the satisfiability of $\varphi_i$), we first dispatch $\psi_1 \wedge \ldots \psi_i \wedge \neg\psi_{i+1}$ to the SMT solver and try to see whether the SMT solver declares it to be unsatisfiable within a short time bound. Our experience indicates that this is often the case, and in such a situation, we do not need to solve the bigger formula $\varphi_i \equiv f \wedge \psi_1 \wedge \ldots \psi_i \wedge \neg\psi_{i+1}$.

*Slicing out Unrelated Symbolic Variables.* Second, using dynamic slicing, we could find the subset of symbolic input bytes that could affect the only branch (contributing to $\psi_{i+1}$) that we want to execute differently in both program versions. For unrelated symbolic input bytes, we use their value from the concrete execution, which guarantees that we are making minimal changes to the failing input. This is useful in practice, since for structured program inputs, the processing of two different portions of the input is often independent. Using concrete values for certain portions of our input greatly simplifies the formulas we need to solve and reduces the burden on the SMT solver.

*Formula Simplification Steps.* We now describe the steps we employ to reduce the amount of time taken in checking the satisfiability of $\varphi_i$.

(1) We impose a short time bound (say ten seconds), and within this time bound, we let the solver check whether $\psi_1 \wedge \ldots \psi_i \wedge \neg\psi_{i+1}$ is satisfiable. If the solver says that $\psi_1 \wedge \ldots \psi_i \wedge \neg\psi_{i+1}$ is unsatisfiable, clearly $\varphi_i$ is not satisfiable. If the solver does not terminate within the time bound or says that $\psi_1 \wedge \ldots \psi_i \wedge \neg\psi_{i+1}$ is satisfiable, we continue with the following steps.
(2) We perform slicing on the (assembly-level) execution trace $\pi'$ corresponding to path condition $f'$ to find out the set of input bytes that $\psi_{i+1}$ is dependent on. This is done as follows. Note that $\psi_{i+1}$ is a primitive constraint corresponding to some branch instance $b$ in the execution trace. Due to traceability links between subformula in the path condition and branches contributing to these formulas, we can find the branch $b$ contributing to $\psi_{i+1}$. Let $l$ be the control location corresponding to $b$ and $Vars$ be the variables appearing in the constraint $\psi_{i+1}$. We perform dynamic slicing [Korel and Laski 1988; Agrawal and Horgan 1990; Wang and Roychoudhury 2004] with respect to the slicing criterion $(l, Vars)$ on the assembly-level execution trace $\pi'$ corresponding to path condition $f'$. During the traversal of the execution trace,

the dynamic slicing algorithm maintains (i) a set of instruction instances (the slice), and (ii) a set of variables $\delta$ whose values need to be explained. At the end of the slicing, we inspect the set of input fields (or bytes) which appear in $\delta$. These are the input bytes on which $\psi_{i+1}$ depends in the trace for $f'$. Let this set of input bytes be $In_{i+1}$.

(3) We assign all input bytes not appearing in $In_{i+1}$ to the actual values used in the concrete execution of the test input $t$ being debugged. We also use forward constant propagation along the execution trace $\pi'$ to propagate these concrete values to other program variables (which do not correspond to program input). This greatly simplifies $f$ as well as $\psi_1 \wedge \ldots \psi_i \wedge \neg\psi_{i+1}$, since many of the variables in the formulas get instantiated to concrete values. Let the simplified formulas be called $f_{simplified}$ and $(\psi_1 \wedge \ldots \psi_i \wedge \neg\psi_{i+1})_{simplified}$.

(4) Finally, we solve the simplified formula $f_{simplified} \wedge (\psi_1 \wedge \ldots \psi_i \wedge \neg\psi_{i+1})_{simplified}$ using an SMT solver. After concretizing the input bytes other than those in $In_{i+1}$ and propagating constants, the formulas to be solved are greatly simplified owing to instantiation. This leads to a greatly reduced solution time.

### 4.3. Backward Traceability and Input Validation

Recall that to solve $f \wedge \neg f'$, we solve $m$ formulas

$$\varphi_i = f \wedge \psi_1 \wedge \ldots \psi_i \wedge \neg\psi_{i+1} \quad 0 \leq i < m,$$

where

$$f' \equiv \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_m.$$

Since $f'$ is a path condition, it is a conjunction of primitive constraints. In other words, each $\psi_i$ appearing in $f'$ is a primitive constraint contributed by a branch in the program $P'$.

Suppose the branch corresponding to $\psi_{i+1}$ is $b_{i+1}$, and the execution path of input $t$ in program $P$ is $\pi(P, t)$. If we can get a solution $t'$ of $\varphi_i$, $\pi(P, t')$ and $\pi(P, t)$ are expected to be the same. The execution paths $\pi(P', t')$ and $\pi(P', t)$ are expected to be the same before $b_{i+1}$ and differ at $b_{i+1}$. Instead of comparing the execution traces $\pi(P', t')$ and $\pi(P', t)$ to get $b_{i+1}$, we can straightaway report $b_{i+1}$ as a potential root cause, provided we can guarantee that $t'$ and $t$ follow different paths in $P'$, differing at branch $b_{i+1}$, and that $t'$ and $t$ follow same path in $P$. This can be validated by concrete execution of tests $t, t'$ in programs $P, P'$.

Note that this validation is necessary, because the computed path conditions are approximations of the exact path conditions. If the input $t'$ is successfully validated, we can directly report $b_{i+1}$ as a potential root cause.

### 4.4. Putting It All Together

Given an input $t$ and two program versions $P$ and $P'$, we compute the path conditions $f$, $f'$ of input $t$ in programs $P, P'$, respectively. First, we try to solve $f \wedge \neg f'$. Instead of directly solving the formula (which may have many solutions), we choose the solutions as follows. Let $f' = \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_m$ where $\psi_i$ are primitive constraints. We solve the $m$ formulas $\{\varphi_i \mid 0 \leq i < m\}$.

$$\varphi_i \stackrel{def}{=} f \wedge \psi_1 \wedge \ldots \psi_i \wedge \neg\psi_{i+1}.$$

For all $0 \leq i < m$, if $\varphi_i$ is satisfiable, we use backwards traceability links to find the branch $b_{i+1}$ contributing to the primitive constraint $\psi_{i+1}$. We report $b_{i+1}$ as a potential root cause if the solution for $\varphi_i$ is successfully validated. This means that the input produced by solving $\varphi_i$ should follow the same path as that of $t$ in $P$ and follow a

different path as compared to $t$ in $P'$. For checking the satisfiability of each $\varphi_i$, we use the four optimization steps given earlier in Section 4.2.

On the other hand, if $f \wedge \neg f'$ is unsatisfiable, we replicate the preceding steps for solving $f' \wedge \neg f$. Again, we do not solve $f' \wedge \neg f$ directly but, instead, solve $k$ formulas $(f' \wedge \theta_1 \wedge \ldots \wedge \theta_i \wedge \neg\theta_{i+1})$, where $f = (\theta_1 \wedge \ldots \wedge \theta_k)$, and $\theta_i$ are primitive constraints. Again, we get a set of at most $k$-validated alternate inputs.

*Working with Different Implementations.* An interesting characteristic of our approach is that we do not require the two programs $P$ and $P'$ to be similar (i.e., versions of the same application). The programs could be two completely different implementations. We only require that the programs operate on the same input space and implement the same specification for all common inputs. As long as these conditions are satisfied, the path conditions we compute will be formulas over the same input variables, and hence, all solutions to $f_{computed} \wedge \neg f'_{computed}$ are valid inputs for both programs. Also, note that although we use two programs to generate new inputs, we always compare inputs on the same program version. Thus, our approach for finding code fragments where two inputs diverge is completely oblivious to the amount of change between programs. This is unlike other approaches [Zeller 1999] that require two reasonably similar program versions, such that a correspondence between parts of the programs could be established. We refer the reader to our case study using Apache and miniWeb Web servers (Section 7) for a more detailed description of this aspect of our approach.

## 5. DEBUGGING COMMON PROGRAMMING ERRORS

We now explain the suitability of our debugging methodology for different common kinds of programming errors: branch errors, assignment errors, and code-missing errors.

*Branch Errors.* We believe that our methodology is naturally suited for localizing errors in branch conditions, because our method finds the difference between two path conditions which consists of branch conditions. So, if the error is in the condition of a branch $b$, typically $b$ will be evaluated differently (from the erroneous trace) in the trace without the observable error. The examples given in Section 3 illustrate this point. Since our approach for synthesizing and comparing tests is based on control flow, our approach is ideally suited to bugs that cause a change in the control flow. Branch condition errors cause a change in control flow and, hence, are easily root-caused using our approach.

*Errors that do no Affect Control Flow.* Since our approach relies on comparing control flow, errors that do not cause any change in the control flow cannot be directly root-caused using our approach. We now describe a strategy that can translate such bugs into those that influence control flow. Inspired by ideas in statistical debugging [Liblit et al. 2005; Liblit 2005], we instrument the program with a predefined family of predicates. These predicates are instrumented as branch conditions at various points in the program. The predicates we instrument are as follows.

—Checks for null and the sign of return values at each function return site.
—Checks for equality of two program variables of the same type. Before each statement that modifies a program variable `x`, we add predicates of the form `x == y` for all variables `y` which are (i) of the same type as `x` and (ii) are live at the statement.

These predicates provide our *DARWIN* with additional opportunities to find new tests that reveal the difference between the actual and the expected control flow of the failing test. On the flip side, the instrumentation can increase the cost of tracing and

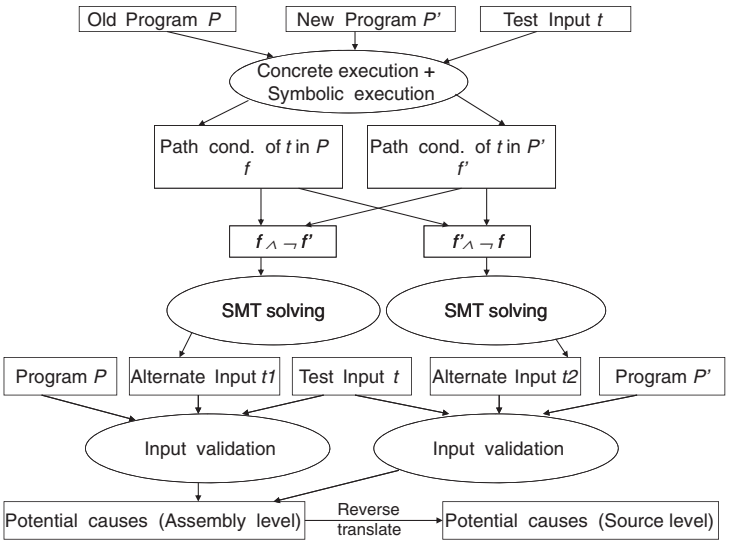


Fig. 5. Architecture of our *DARWIN* toolkit. It takes an old program $P$, a new program $P'$, and a test input $t$ which passes in $P$ but fails in $P'$. The output is a report explaining the behavior of test $t$. The entire flow is automated.

the complexity of constraint solving. In our experiments, we measured the overheads from instrumentation and found it to be less than 20% for our subject programs (see Section 7.7).

*Code-Missing Errors.* Code-missing errors correspond to portions of code left out during the change of a program. Such code would be missing in the buggy program but would be present in the reference program. Whether the missing code chunk contains assignments (which, if they were present, would have affected control flow via instrumented branches) or branches (which directly affect control flow), the reference program $P$ could be expected to have more paths than the buggy program $P'$. Given a failing test input $t$ and $f$, $f'$ being the path conditions of $t$ in $P$, $P'$, respectively, we can thus expect $f' \wedge \neg f$ to yield a solution. This would be an input $t'$ following the path of $t$ in $P'$ but following a different path than $t$ in $P$ (the code missing in $P'$ is present in $P$, leading to more branches and more paths). Thus, the traces of $t'$ and $t$ in $P$ would be compared to yield potential root causes. No extension is needed in our methodology to handle code-missing errors.

## 6. IMPLEMENTATION

We now describe our implementation setup. The overall architecture of *DARWIN* is summarized in Figure 5. We built *DARWIN* on top of the BitBlaze platform [Song et al. 2008]. Most of the modules used by *DARWIN* are contained in the recent open-source release of BitBlaze. However, BitBlaze does not have the modules for formula manipulation and optimization. We built these modules for *DARWIN* on our own.

### 6.1. Generating Alternate Inputs

*DARWIN* uses a symbolic execution engine for computing the path condition of a given program execution. Our execution engine is a part of the BitBlaze platform [Song et al. 2008], which works on $\times 86$ binaries. Given an input, the platform concretely executes the program on the specific input and records the trace. It then performs symbolic execution to compute the path condition of the concrete trace recorded. The

path condition represents a constraint denoting the set of inputs which execute the concrete trace.

The concrete execution is carried out by TEMU, a whole-system emulator based on QEMU [2009]. TEMU can run Windows and Linux as its guest operating system, enabling us to analyze both Windows and Linux binaries. After the concrete execution, TEMU generates a trace of instructions executed by the program. The trace is also annotated with input dependence information, for example, whether the operand of an instruction is dependent on input (an operand is dependent on the input if there is a data dependence chain from the operand to an input). TEMU allows users to specify several types of inputs, such as network inputs, files, and keyboard inputs.

The path condition calculation is performed by the VINE component of BitBlaze. It first defines the bytes in the program input as symbolic variables: each byte in the input is a distinct variable. Then, it makes a forward pass through the trace recorded by TEMU, considering only *tainted* instructions, that is, instructions whose operands are (directly or transitively) dependent on the program input (via data dependencies). Note that such dependency information is present as annotations in the trace recorded by TEMU. For each tainted instruction in the trace, VINE translates the instruction to a sequence of statements in its own intermediate language, where the semantics of each instruction is preserved [Brumley et al. 2007]. This translation helps the VINE tool deal with the complexity of the x86 instruction set. Finally, VINE performs a traversal of the trace in the intermediate language to compute the path condition.

Two points need to be noted about the BitBlaze execution engine and its interplay with our debugging framework. First, the concrete and symbolic execution engines work on $\times 86$ binaries. Our path conditions are also computed at the level of binaries rather than source code, thereby capturing the precise semantics of the program execution. On the other hand, the bug report is computed at the level of source code for ease of understanding (by the programmer). Second, the variables appearing in the path condition correspond to the different bytes of the program input.

Given program versions $P$ and $P'$ and a test input $t$ which passes in $P$ and fails in $P'$, we compute the path conditions $f$, $f'$ of input $t$ in programs $P$, $P'$, respectively. In fact, the symbolic execution engine in BitBlaze constructs these path conditions as formulas in the well-known SMT-LIB [Ranise and Tinelli 2003] format. The SMT-LIB format is supported by all the solvers that participated in the SMT annual competition. Thus, expressing the path conditions in the SMT-LIB format allows us to leverage a lot of state-of-the-art SMT solvers. It also allows us to benefit from the ongoing improvement in the solving ability of the existing solvers—we can use whichever solver is currently the fastest. The solver we are currently using is Boolector [Brummayer and Biere 2009], the winner of the SMT competition in 2009 for quantifier-free formulas with bitvectors, arrays, and uninterpreted functions (the QF_AUFBV category). Indeed, this is suitable for us, since our formulas do not have universal quantification, and any variable is implicitly existentially quantified.

### 6.2. Reporting Root Causes

Given the solutions of $f \wedge \neg f'$, we first validate them. In case we find $f \wedge \neg f'$ to be unsatisfiable or none of the solutions of $f \wedge \neg f'$ can be validated, we solve $f' \wedge \neg f$ in a similar fashion. By following the steps mentioned in the previous section (solving either $f \wedge \neg f'$ or $f' \wedge \neg f$), we obtain a set of branches at the assembly level as potential root causes of the bug. Using standard compiler-level debug information, these can be reverse-translated back to lines in source code.

*Accuracy of Our Reports.* We now discuss some low-level issues which make a substantial difference to the accuracy of our results. Given the path conditions $f$ and $f'$, let

```
if (!(png_ptr->mode & PNG_HAVE_PLTE))
{
   png_warning(png_ptr, "Missing PLTE before tRNS");
}
else if (length > (png_uint_32)png_ptr->num_palette)
{
   png_warning(png_ptr, "Incorrect tRNS chunk length");
   png_crc_finish(png_ptr, length);
   return;
}
```

Fig. 6. Buggy code fragment from `libPNG`.

$f' = (\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_m)$, where $\psi_i$ are primitive constraints. As mentioned in the previous section, we solve the $m$ formulas $\{\varphi_i \mid 0 \leq i < m\}$, where $\varphi_i \stackrel{def}{=} f \wedge \psi_1 \wedge \ldots \psi_i \wedge \neg\psi_{i+1}$. The VINE symbolic execution engine ensures that the path conditions contain only constraints from branches which are dependent on the program input. In practice, this greatly cuts down on the number of $\psi_i$ constraints and, hence, the number of $\varphi_i$ formulas that need to be dispatched to the SMT solver. Since each $\varphi_i$ formula contributes at most one statement in our report, we get a smaller-sized report by reducing the number of $\varphi_i$. If the number of root causes is still high (due to a large number of alternate inputs), we prioritize statements obtained from successful alternate inputs over other statements, since these are more likely to reveal the real root cause.

## 7. DEBUGGING EXPERIENCE

We report our experience in using *DARWIN* to locate error causes in real-life case studies.

### 7.1. Experience with `libPNG`

We first describe our experience in debugging the `libPNG` open source library [LibPNG 2009], a library for reading and writing PNG images. We used a previous version of the library (1.0.7) as the buggy version. This version contains a known security vulnerability, which was subsequently identified and fixed in later releases. A PNG image that exploits this vulnerability is also available online. As the reference implementation, or stable version, we used the version in which the vulnerability was fixed (1.2.21). Assuming this vulnerability was a regression bug, we used our tool to see if the vulnerability could be accurately localized.

The bug we localized is a remotely exploitable stack-based buffer overrun error in `libPNG`. Under certain situations, the `libPNG` code misses a length check on PNG data prior to filling a buffer on the stack using the PNG data. Since the length check is missing, a buffer overrun may occur. What is worse, such a bug may be remotely exploited by emailing a bad PNG file to another user who uses a graphical email client for decoding PNGs with a vulnerable `libPNG`. In Figure 6, we show a code fragment of `libPNG` showing the error in question. If the first condition `!(png_ptr->mode & PNG_HAVE_PLTE)` is true, the length check is missed, leading to a buffer overrun error. A fix to the error is to convert the `else if` in Figure 6 to an `if`. In other words, whenever the length check succeeds, the control should return.

We now explain some of the issues we face in localizing such a bug using approaches other than ours. Suppose we have the buggy `libPNG` program and a bad PNG image which causes a crash due to the preceding error. If we want to perform program-differencing methods (such as source code "diff") to localize the bug, there are 1,589 differences in 28 files. Manually inspecting these differences requires a lot of effort. *Semantic diff* [Jackson and Ladd 1994; Ren et al. 2004; Horowitz 1990; Apiwattanapong

et al. 2004] could only provide limited help to the manual inspection. Because of the very large number of source-code differences, the number of semantic differences would still be large. Moreover, given a coarse-grained semantic difference, such as *method change* [Ren et al. 2004], one still needs to inspect more details to tell whether this change indeed causes the bug.

If we want to localize the error by an analysis of the erroneous execution trace starting from the observable error, it is very hard to even define the observable error. Even if the buffer being overrun is somehow defined as the observable error, tracking program dependencies from the observable error could be problematic for the following reason. The `libPNG` library is used by a client which inputs an image, performs computation, and outputs to a buffer (the one that is overrun due to an error inside `libPNG`). In this case, we are debugging the sum total of the client, along with the `libPNG` library. Since almost all statements in the client program and many statements in `libPNG` involve manipulation of the buffer being overrun itself, a dynamic slicing approach seems to highlight almost the entire client program, as well as large parts of the `libPNG` library.

If we want to employ statistical bug isolation methods (which instrument predicates and correlate failed executions with predicate outcomes), the key is to instrument the right predicate. In this case, the predicates in question (such as `!(png_ptr->mode & PNG_HAVE_PLTE)` ) contain pointers and fields. Hence, they would be hard to guess using current statistical debugging methods which usually consider predicates involving return values and scalar variables.

If we want to perform debugging by trace comparison, we must compare the trace of the bad PNG image (which exposes the error) with the trace of a good PNG image (which does not show the error). The question then is how do we get the good PNG image? Even if we have a pool of good PNG images from which we choose one, making the right choice becomes critical to the accuracy of root cause analysis. Moreover, the method is sensitive to the pool of PNG images available at hand.

We now describe the working of our *DARWIN* method in terms of root-causing the libPNG bug. Given the bad PNG image[1], *DARWIN* synthesizes an alternate PNG image via semantic analysis of the execution traces of the bad PNG image in the two program versions. This image is a minimal modification of the bad PNG image. Our analysis only minimally changes the bad PNG image to get a good image as alternate program input.

Specifically, *DARWIN* first computes the path conditions of the bad PNG image on the two libPNG versions 1.0.7 and 1.2.21. Let these be $f_{buggy}$ and $f_{fixed}$, respectively. We find that $f_{fixed} \wedge \neg f_{buggy}$ is unsatisfiable, so we solve for $f_{buggy} \wedge \neg f_{fixed}$. By solving this formula, we get nine alternate inputs from the Boolector solver. These nine alternate inputs are essentially nine PNG images. All these nine inputs passed validation, hence we report nine statements as potential root causes.

We prioritize these nine statements as follows. Among the nine alternate inputs, we find out which of them are successful, that is, the program output for a successful input should be the same in both the program versions. Only one of our nine alternate inputs is found to be successful. The branch instruction contributed (to the result) by this input corresponds to the branch `length > (png_uint_32)png_ptr->num_palette`, thereby pointing directly to the cause of failure. This branch is (mistakenly) not executed in the buggy libPNG version 1.0.7.

*Discovering New Errors.* Interestingly, in the process of this debugging, we found other potential problems in `libPNG`. As mentioned earlier, *DARWIN* obtained nine alternate inputs, only one of which exhibits bug-free behavior, and pointed us to the

[1]The bad PNG image is from http://scary.beasts.org/security, with reference number CESA-2004-001.

error. Interestingly, the other branch instructions point us to other deviations between the two versions of libPNG. For example, by following one of these eight instructions, we find that the two versions of libPNG use different functions to retrieve the length field of a chunk from the input. In version 1.0.7, we have

```
length = png_get_uint_32(chunk_length);
```

while in version 1.2.21, we have

```
length = png_get_uint_31(chunk_length).
```

In particular, the code for png_get_uint_31 is as follows.

```
png_get_uint_31(png_structp png_ptr, png_bytep buf)
{
   png_uint_32 i = png_get_uint_32(buf);
   if (i > PNG_UINT_31_MAX)
     png_error(png_ptr, "PNG unsigned integer
                         out of range.");
   return (i);
}
```

Thus, png_get_uint_31 first uses png_get_uint_32 and then performs a length check. If png_get_uint_32 is directly used to find the length of a chunk, a length check with respect to the constant PNG_UINT_31_MAX is missing. We also report the branch instruction containing this missing length check, thereby pointing to another potential error in libPNG.

### 7.2. Experience with miniweb-apache

In our second case study, we study the Web server miniweb [Huang 2009], an optimized HTTP server implementation which focuses on low resource consumption. The input query whose behavior we debugged was a simple HTTP GET request for a file, the specific query being "GET x." Ideally, we would expect miniweb to report an error, as x is not a valid request URI (a valid request URI should start with '/'). However, miniweb does not report any errors and returns the file index.html. We then attempt to localize the root cause of this observable error.

We found that even the latest version of miniweb contains the error. Therefore, we could not choose another version of miniweb as the reference implementation. We chose another HTTP server apache [Apache 2009] as the reference implementation. apache is a well-known open-source secure HTTP server for Unix and Windows. Since both apache and miniweb implement the HTTP protocol, they should behave similarly for any input accepted by both implementations. Further, apache does not exhibit the bug we are trying to fix. It reports an error on encountering the input query "GET x."

We generate the path conditions of "GET x" in both apache and miniweb. Let these be $f_{apache}$ and $f_{miniweb}$, respectively. We find $f_{apache} \wedge \neg f_{miniweb}$ to be unsatisfiable. However, by solving $f_{miniweb} \wedge \neg f_{apache}$, we can get alternate input queries. By following our methodology described in Section 4.1, we get exactly five alternate inputs and, hence, five potential root causes.

GET /, GET \, GET *, GET . and GET %.

Based on the first of these five branches, we were able to localize the bug immediately. miniweb does not check for '/' in GET queries and treats the query "GET x" similarly to "GET /", thereby returning the file index.html.

*Discovering New Errors.* Only one of our five alternate inputs was successful, exhibiting the same output in both program versions. The branch instruction corresponding to this input pointed us to the missing check for '/'. The other statements pointed us

to other missing checks in `miniweb`. Indeed, we can locate that `apache` contains checks for each of these five characters, while `miniweb` misses the check for all five of them, leading to potential errors.

*In a Broader Perspective.* Our experiments with `apache` and `miniweb` also give us a broader perspective on the applicability of our method. Even if all versions of a program exhibit a given error (as was the case with `miniweb`), we can still use *DARWIN* to localize the error. We only need a reference program which is intended to behave similarly to the program being debugged, and does not exhibit the bug being localized. In our experiments, the `apache` Web server was the reference program. Thus, the applicability of our method is broader than delta debugging methods, like Zeller [1999], which search for bugs within the changes across program versions.

### 7.3. Experience with `savant-apache`

`Savant` [Savant 2009] is a full-featured open-source Web server for Windows. We notice that `savant` does not report any errors when faced with an input query of the form "`GOT /index.html`," a typo from the valid HTTP GET request "`GET /index.html`." We cannot choose another version of `savant` as the reference program because the latest version of `savant` also exhibits this error. As a reference program, we choose the `apache` Web server, which reports an error for the query "`GOT /index.html`." Both `savant` and `apache` implement the HTTP protocol and are expected to behave similarly.

In this case study, *DARWIN* found 46 alternate inputs. Out of these, only one is successful, that is, produces the same output in both `savant` and `apache`. This is the input "`GET /index.html`." Using the branch instruction corresponding to this alternate input, *DARWIN* pinpointed the error to missing checks in `savant`. The `savant` program does not check for all three letters 'G', 'E', 'T' in HTTP GET requests for HTTP protocol version HTTP/0.9 (which is the default assumed, since we do not explicitly specify an HTTP protocol version in the query "`GOT /index.html`"). Indeed, we found that `savant` reports an error if we provide "`GOT /index.html HTTP/1.0`" as input. In HTTP/0.9 there is only one command, namely GET. The error lies in the fact that `savant` does not check for the string "`GET`", and assumes any given string to be the GET command.

*Discussion.* Our experiments with `savant` also illustrate another additional feature of *DARWIN*—the ability to rectify program inputs. The process of alternate input generation in *DARWIN* can help correct errors in an almost correct program input, such as the input "`GOT /index.html`." In this case, the input fix was easy and could have been done manually as well. In the future, we plan to conduct experiments with programs like Web browsers to see if an almost correct HTML file (where the incorrectness in the file is hard-to-see) can get rectified through alternate input generation.

### 7.4. Experience with `TCPflow`

We use two versions of the `TCPflow` program, namely `TCPflow_0.21.ds1-2` and the same version with the patch `10_extra-opts.diff`, which is supposed to provide the user with some extra options. TCP is the most popular transport layer protocol, and `TCPflow` is a program which captures and displays data sent through TCP connections. The statistics about the `TCPflow` program are given in Table II.

What is the intended functionality of the `TCPflow` program? If we capture the raw TCP packets transmitted over the network, there is a TCP header inside each TCP packet. Inside each raw packet, we also have the header for the network layer protocol (usually the IP protocol). Thus, it is nontrivial to manually distinguish which parts in a raw packet correspond to the real data being transmitted. Moreover, there can be multiple active TCP connections at the same time. As a result, it is hard to determine which packets are from the same connection. `TCPflow` is a program which solves these

```
47 45 54 20 2F 69 6E  64 65 78 24 68 74 6D 20 0D  |GET /index.htm .|
0A 0D 0A                                          |...              |
```

Output from the unpatched version of TCPflow

```
00 47 45 54 20 2F 69 6E  64 65 78 24 68 74 6D 20  |.GET /index.htm |
0D 0A 0D 0A                                       |....             |
```

Output from the patched version of TCPflow

Fig. 7. Output from the TCPflow program.

```
// unpatched version of the TCPflow
handle_tcp (packet_t packet) {
   if( this packet has no data) {
        return;
   }
   if ((state = find_flow_state(current_flow)) == NULL)
        state = create_flow_state(flow, seq);
   offset = seq - state->ins;
   write data from offset;
}
// patched version of the TCPflow
handle_tcp (packet_t packet) {
   if( this packet has no data) {
        if ((state = find_flow_state(current_flow)) == NULL)
            state = create_flow_state(flow, seq);
        return;
   }
   offset = seq - state->ins;
   write data from offset;
}
```

Fig. 8. Schematic code fragment from TCPflow.

problems. It analyzes the raw data (TCP packets) from TCP connections and outputs the actual data being transmitted over the network. A TCP connection is associated with source IP address, destination IP address, source port, and destination port. The output from TCPflow is also classified by the connections.

TCPflow can read input both from network and file. If the input is from network, then it captures the data that is being transmitted and analyzes the data. In our experiment, the input is from a file which is generated by tcpdump.

The bug we investigate is introduced by the patch 10_extra-opts.diff. We provided two packets from the same connection to TCPflow: an SYN packet to setup the connection and a simple HTTP request packet. Figure 7 shows the output from both versions of the TCPflow program, where only the HTTP request payload is shown, and the headers from TCP layer and IP layer are excluded.

The two versions of TCPflow we use are TCPflow_0.21.ds1-2 and the same version with the patch 10_extra-opts.diff. Although the patch is supposed to provide some extra options to the user, it actually introduces a bug into the code. Figure 8 is a simplified code pattern from TCPflow. For each TCP connection, a struct named flow_state_t is used in the program to maintain some data associated with the connection. The program processes the packets one by one from the start to the end. So, for our program input, the SYN packet is processed before the data packet. The bug appears, because the manner in which empty packets are handled is changed by the patch.

In the unpatched version of the program, if we see an empty packet and no other packets from the same connection have been seen before, the packet is simply ignored (the struct flow_state_t for the connection is not created at all). However in the patched

version, empty packets are not ignored (the struct `flow_state_t` for the TCP connection is still created). Note that in TCP connections, each transmitted packet has a sequence number which is used by the sliding window protocol to make sure the packet is transmitted to the destination. In our case, the sequence number of the data packet is just the sequence number of the SYN packet increased by one. Given a TCP connection, the corresponding struct `flow_state_t` has one critical member field named `ins` which is used to store the initial sequence number the program has seen for this connection. When a `flow_state_t` is created, `ins` is assigned with the sequence number of the current packet being handled.

Since the SYN packet has no data inside, and the manner of handling such packets are different in the two program versions, the `flow_state_t` are created with different `ins` values in two program versions. In the unpatched version, because the SYN packet is ignored, the `flow_state_t` is only created when the data packet is seen, so the `ins` field is equal to the sequence number of the data packet. In the patched version, the `flow_state_t` for this connection is created when the SYN is seen, so the `ins` field is equal to the sequence number of the SYN packet. Note that the `ins` field is later used to calculate the offset in the output file when the data is written out. The offset is calculated via the statement

```
offset = seq - state->ins;
```

where `seq` is the sequence number of the packet being written. So, while writing the data packet in the unpatched version, the value of `seq` is equal to the value of `state->ins`; they are both set to the sequence number of the data packet. However, in the patched version, the `seq` is the sequence of the data packet, and the `state->ins` is the sequence of the SYN packet. So the offset is 1 in the patched version, making the program write from the second byte in the output file. As a result, there is an additional $0\times00$ (in the first byte of the buggy output), as shown in Figure 7.

Once again, we emphasize that the bug we previously described (and detected using *DARWIN*) is a real-life bug appearing in a patch of the `TCPflow` program. The bug happens because the authors of `TCPflow` forgot to modify the update of `state->ins` field after the manner of handling empty packets was changed. In fact, this bug is only observed when the input to `TCPflow` contains at least one empty packet. When we attempted to localize the root cause of this bug using *DARWIN*, the root causes we reported were extremely accurate. Only six statements are reported as potential root causes, and one of them points to a branch condition which checks for empty packets.

Over and beyond the accuracy, making *DARWIN* work on the `TCPflow` program presented us with a substantial challenge in terms of scalability. Although the `TCPflow` program contains only 1,000 lines of code, its path condition size was the largest among all our four case studies. Part of the reason for this comes from the frequent usage of libraries during the execution of `TCPflow`. The execution of the libraries bloats up the trace size and creates substantial time overheads for symbolic execution. Recall that we are trying to solve formula of the form $f_{unpatched} \wedge \neg f_{patched}$, where $f_{unpatched}$, $f_{patched}$ are the path conditions of our chosen program input on the unpatched and patched versions of `TCPflow`, respectively. Assuming $f_{patched} \equiv \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_m$, we actually solve $m$ formulas $\{\varphi_i \mid 0 \leq i < m\}$ where,

$$\varphi_i \stackrel{def}{=} f_{unpatched} \wedge \psi_i \wedge \psi_2 \ldots \wedge \psi_i \wedge \neg\psi_{i+1}.$$

Without the optimizations mentioned in Section 4.2, solving each $\varphi_i$ takes up to 30 minutes, and there are around 2,000 $\varphi_i$ formulas to solve!! Clearly, such a time overhead would be unacceptable.

```
if((state->ins != seq) && !(IS_SET(flags, TH_ACK))){
      return;   /* ERROR here: should be printf("Warning: xxxxxx\n"); */
}
```

Fig. 9. Injected bug in `TCPflow`.

Let us now examine the impact of the different optimizations mentioned in Section 4.2. In the experiment with `Tcpflow`, we use one additional optimization technique to further shorten the formula solving time. We only solve those $\varphi_i$ formulas where $\psi_{i+1}$ corresponds to a branch in the source code (not a branch within a library). The effect of this technique is discussed next. By considering only $\varphi_i$ formulas from the source code, there are still 86 formulas left to solve. The estimated time for solving these formulas comes to two days (since the solving of each $\varphi_i$ formulas in the `TCPflow` program seems to take about 30 minutes). However, recall that in the first step of our formula simplification (see Section 4.2), we check whether $\psi_1 \wedge \ldots \psi_i \wedge \neg\psi_{i+1}$ is satisfiable in a time-bounded fashion. In other words, we set a time limit (ten seconds for our experiments), and see how many of the $\varphi_i$ formulas can be proved to be unsatisfiable within this time limit. Clearly, if $\psi_1 \wedge \ldots \psi_i \wedge \neg\psi_{i+1}$ is unsatisfiable, $\varphi_i$ cannot be satisfiable! We find that 64 out of the 86 formulas are proved to be unsatisfiable in this fashion. Thus, we are left with (86–64), that is, 22 formulas to solve. The estimated time for solving these formulas without any further optimization comes to around 12 hours. As mentioned in Section 4.2, we further employ dynamic slicing and constant propagation to reduce the burden of the SMT solver. By using all of the formula simplification steps mentioned in Section 4.2, the total time taken by the SMT solver is reduced to only ten minutes. The total debugging time (which includes tracing as well) comes to 33 minutes. The final result from *DARWIN* contains only six statements, including the line containing the error cause.

### 7.5. Experiment with Latent Bug

In this section, we report our experience with a latent injected bug to show a special feature of our debugging method. We want to demonstrate the scenario in which the actual bug exists in the old stable program; however, it only gets manifested in the new changed program. Note that in such scenarios, change-analysis-based debugging methods, such as that of Zeller [1999], will not work, since they seek to report a subset of the changes (between the old and new programs) as the cause of error. However, our method, being based on semantic analysis of the old and new programs, can still locate the error cause.

We use the unpatched and patched versions of the `TCPflow` program, as described in Section 7.4. The injected bug is shown in Figure 9. The code in Figure 9 is injected in both versions of `TCPflow`. In the unpatched version of `TCPflow`, whenever the code is executed, `state->ins` is always equal to `seq`, the second condition `!(IS_SET(flags, TH_ACK))` is never evaluated, and the `return` statement is never executed. However, in the patched version, because of other code modifications, `state->ins` can be not equal to `seq`. As a result, the `return` statement is executed, manifesting the error.

Although we have the same buggy code in both versions, the injected code is actually executed differently in the two versions. This difference is caused by other modifications in the patched version. Change-analysis-based delta debugging [Zeller 1999] cannot expose such error causes, since the error is in a line which was not changed across versions.

Using our technique, the difference in program executions is captured in the path conditions $f_{unpatched}$ and $f_{patched}$ of the unpatched/patched program versions. The branch `!(IS_SET(flags, TH_ACK))` appears in $f_{patched}$ but not in $f_{unpatched}$. So our technique is

Table II. Properties of the Subject Programs

| Programs | LOC | Trace size (# instructions) | # Branches in trace | # Tainted instructions |
|---|---|---|---|---|
| libPNG v1.0.7 | 31,164 | 87,336 | 13,635 | 2,999 |
| libPNG v1.2.21 | 36,776 | 108,769 | 15,472 | 2,592 |
| Miniweb | 2,838 | 270,856 | 26,201 | 331 |
| Savant | 8,730 | 121,714 | 16,212 | 1,613 |
| Apache | 358,379 | 60,380 (miniweb) | 5,388 (miniweb) | 264 (miniweb) |
| | | 74,002 (savant) | 9,672 (savant) | 6,889 (savant) |
| TCPflow (unpatched) | 895 | 56,838 | 7,210 | 7,753 |
| TCPflow (patched) | 934 | 58,079 | 7,375 | 7,860 |

Table III. Performance of *DARWIN*'s Extended Debugging Method

| Programs | Time in step 1 | Time in step 2 | Time in step 3 | Time in step 4 | Total Time |
|---|---|---|---|---|---|
| libPNG(v1.0.7-v1.2.21) | 3m 57s | 1m 49s | 7m 44s | 4s | 13m 34s |
| Miniweb-Apache | 2m 4s | 1m 1s | 2m 42s | 1s | 5m 48s |
| Savant-Apache | 2m 27s | 1m 11s | 5m 2s | 10s | 8m 50s |
| TCPflow(unpatched-patched) | 7m 9s | 57s | 20m 12s | 3m 32s | 31m 50s |

*Note*: m = minutes, s = seconds.

able to construct an alternate input that satisfies $f_{unpatched} \wedge \neg f_{patched}$ by negating the branch `!(IS_SET(flags, TH_ACK))`. Thus, one of our $\varphi_i$ formulas corresponds to a deviation in the branch `!(IS_SET(flags, TH_ACK))`, since this is a branch recorded in the path condition. This deviation results in `!(IS_SET(flags, TH_ACK))` being selected as a potential root cause. On the whole, we identify ten potential root causes. Clearly, the inclusion of the branch `!(IS_SET(flags, TH_ACK))` as a potential root cause helps the programmer diagnose the issue.

### 7.6. Performance of Our Debugging Method

In this section, we evaluate the performance of our debugging method. The properties of our subject programs in terms of trace size and other statistics appear in Table II.

Recall from Section 4 that our debugging method involves four steps. The steps are (i) constructing and checking the satisfiability of the $\psi_1 \wedge \ldots \psi_i \wedge \neg\psi_{i+1}$; (ii) slicing on the $f'$; (iii) concretizing all the inputs that are not in the slicing result and performing constant propagation; and (iv) solving the simplified formula $f \wedge \psi_1 \wedge \ldots \psi_i \wedge \neg\psi_{i+1}$.

Table III summarizes the time taken in these steps by *DARWIN* for all programs including `TCPflow`. The input validation only compares whether two execution traces are the same or different; no formula generation is needed. It takes hardly any time to validate the inputs in all our case studies.

In the first step of our method, we construct the path conditions in the two program versions and then construct several formulas $\varphi_i$. We also use a very short time to check the satisfiability of $\psi_1 \wedge \ldots \psi_i \wedge \neg\psi_{i+1}$. We count the time taken to generate the traces and raw path conditions into this step. The total time taken in this step was less than seven minutes in all the case studies. In the second step, we use dynamic slicing to find out the relevant input bytes for each formula. The time taken was less then two minutes in all the case studies. In the third step, we concretize all the irrelevant input bytes and perform constant propagation to simplify the formulas. The time taken by this step was less than 21 minutes in all our case studies. In the fourth step, we solve the whole formula $f \wedge \psi_1 \wedge \ldots \psi_i \wedge \neg\psi_{i+1}$ (which has been greatly simplified by now, due to constant propagation). The time taken by this step was less than four minutes in all the case studies.

Table IV. Overhead of Predicate Instrumentation

| Programs | Additional branches (%) | Additional Instructions (%) |
|---|---|---|
| TCPflow | 17.78% | 16.48% |
| Miniweb | 4.06% | 3.83% |

Overall, *DARWIN* took less than 32 minutes in all the case studies. We consider this time to be very tolerable, considering that programmers often take hours and days to find the root causes of errors in large code bases.

### 7.7. Additional Overheads Due to Predicate Instrumentation

Our debugging method is most suited for debugging branch errors (errors in program branches) and code-missing errors. For errors in assignments, our technique needs to be augmented with predicate instrumentation, as discussed in Section 5. Our predicate instrumentation is geared to expose assignment errors, as mentioned in Section 5. We introduce branches with branch conditions, checking that the following conditions hold.

—There are function return values at each function return site.
—There are binary constraints describing the equality of a program variable `x` with other variables of the same type, at each assignment to `x`. Thus, if `x`, `y` are of the same type, we introduce branches to check `x == y`.

Table IV shows the overhead for our predicate instrumentation. The additional branches and instructions are introduced because of our predicate instrumentation. We only show the numbers for `TCPflow` (a program with high instrumentation overhead) and `miniweb` (a program with low instrumentation overhead). The overhead, in terms of number of additional branches and instructions, is less than 20%. The instrumentation is done at source-code level, and hence, library code is not instrumented. This also prevents the instrumentation overhead from blowing up.

## 8. RELATED WORK

Validation of evolving programs is an important problem, since any software system moves from one version to another. Among the established efforts in this direction is the work on regression testing, which focuses on which tests need to be executed for a changed program. Even though regression testing, in general, refers to any testing process intended to detect software regressions (where a program functionality stops working after some change), often regression testing amounts to re-testing of tests from an existing test suite. In the past, there have been several research directions which go beyond re-testing all of the tests of an existing test suite. One stream of work has espoused test selection [Chen et al. 1994; Rothermel and Harrold 1997], selecting a subset of tests from an existing test suite (before program modification) for running on the modified program. Another stream of works propose test prioritization [Elbaum et al. 2000; Srivastava and Thiagarajan 2002], that is, ordering tests in existing test suites to better meet testing objectives of the changed program. Finally, Santelices et al. [2008] has studied test-suite augmentation, that is, developing certain criteria for new tests so that they are likely to stress the effect of the program changes. Our technique is complementary to regression testing, in that regression testing detects or uncovers software regressions, whereas we explain (already detected) software regressions.

Using path conditions to partition input space has been explored in concolic testing works [Godefroid et al. 2005; Sen et al. 2005]. However, the problem tackled by us is entirely different from concolic testing. The main focus of concolic testing is exploring the input space of one program to find test cases, whereas our technique performs simultaneous analysis of two program versions for debugging a given test.

The issues in comprehending program changes for an evolving code base have been articulated by Sillito et al. [2006]. Program differencing methods [Horowitz 1990; Apiwattanapong et al. 2004; Ren et al. 2004] try to identify changes across two program versions. Indeed, this can be the first step towards detecting errors introduced due to program changes—identifying the changes themselves. The works on change impact analysis are often built on such program differencing methods (e.g., Ren et al. [2004], where the analysis identifies not only the changes, but also which tests are affected by which changes). Person et al. [2008] recent work uses symbolic execution to accurately capture behavioral differences between program versions. Overall, the works on program differencing try to identify (via static analysis) possible software regressions, rather than finding the root cause of a given software regression. Dynamic-analysis-based change detection methods have also been studied (e.g., Giroux and Robillard [2006], which analyzes the change in dependencies between parts of a program via regression testing). These works focus on qualitative code measures and the possible impact of program changes. Instead, we focus on the issue of root causing a bug that has surfaced due to program changes.

In the area of computer security, deviation detection of various protocol implementations have been studied (e.g., Brumley et al. [2007]). This problem involves finding corner test inputs, in which two implementations of the same protocol may deviate in program output. We note that finding such deviating program inputs bears similarities with uncovering software regressions, whereas our work is focused on explaining already uncovered software regressions. Even though Brumley et al. [2007] appear to employ techniques similar to ours, their goal is to generate a deviating program input which can demonstrate the behavior differences between two programs, while the goal of our work is to explain such a behavior difference. Thus, the deviating program input generated by Brumley et al. [2007] can be fed to our debugging method.

Turning now to works on software debugging, the last decade has seen a spurt of research activity in this area. Some of the works are based on static analysis to locate common bug patterns in code (e.g., Hovemeyer and Pugh [2004]), while others espouse a combination of static and dynamic analysis to find test inputs, which expose errors (e.g., Csallner and Smaragdakis [2006]). Another area of works addresses the problem of software fault localization (typically via dynamic analysis), that is, given a program and an observable error for a given failing program input, these works try to find the root cause of the observable error. Our work solves this problem of fault localization, albeit for evolving programs. Next, we discuss the works on fault localization.

The works on software fault localization proceed by either (a) dynamic dependence analysis of the failing program execution (e.g., [Sridharan et al. 2007; Zhang et al. 2006, 2007]), or (b) comparison of the failing program execution with the set of all correct executions (e.g., Ball et al. [2003]), or (c) comparison of the failing program execution with one chosen program execution which does not manifest the observable error in question (e.g., [Zeller 2002; Renieris and Reiss 2003; Guo et al. 2006]). Our work bears some resemblance to works which proceed by comparing the failing program execution with one chosen program execution. Our approach tries to construct an alternate input with whose trace we compare the failing program execution. However, the main novelty in our approach lies in its ability to consider two different programs in the debugging methodology. We do so by a separation of concerns: the two different program versions are used to generate alternate program input (apart from the failing program input), while the executions of the alternate input and failing input in the modified program version are compared.

Comparing with delta debugging [Zeller and Hildebrandt 2002], we find that it cannot be used in general to construct alternate inputs for evolving program debugging. Consider a test input $t$ showing a regression bug (failing in one program version,

passing in another). Delta debugging generates alternate inputs by deleting certain fields of $t$ which are irrelevant to the bug. However, it cannot generate new test inputs by modifying certain fields of $t$; this is done in our method. For example, in our `libPNG` case study, the bad PNG image contains a *chunk* (a PNG file is divided into chunks) with an incorrect length field. To make the bug disappear, we need to correct the length field, rather than delete fields in the PNG input. Moreover, arbitrary deletion in the PNG input will create illegal PNG inputs, since the checksum will not match. In contrast, the semantic analysis supported by our path conditions (where the relationship between the checksum and the other fields is captured in the path condition) ensures that we generate an alternate test input which is a legal PNG image and avoids the bug in question.

Zeller [1999] studies debugging of evolving programs and proposes identifying failure-inducing changes. However, this is restricted to only reporting the changes as error causes. Errors present in the old version, which get manifested due to changes, cannot be explained using such an approach. Moreover, suppose during program evolution we encounter a bug for the first time (a test input which was ignored during the testing of the past versions). Such bugs are not regression bugs. Our approach can still be applied, provided a reference implementation is available; this is demonstrated in our experiments with Web servers. In such a situation, searching among changes across implementations is unlikely to work, since the reference implementation is a completely different program, often with different algorithms/data structures.

In summary, existing works on program-analysis-based software debugging have not studied the debugging of evolving programs. In particular, the possibility of exploiting stable implementations (which were thoroughly tested) for finding the root cause of an observable error in a buggy implementation has not been studied. This, indeed, is the key observation behind our approach. Moreover, existing works on evolving software testing/analysis primarily focus on finding tests which show differences in the behavior of different program versions. These works do not prescribe any method for explaining or debugging a failed test, an issue that we study here.

## 9. THREATS TO VALIDITY

In this section, we discuss certain threats to validity of the results presented in this article. This also clarifies any implicit assumptions on which our debugging method may be built.

—One key assumption of our approach is that the program requirements vis-a-vis the buggy input do not change. The program requirements for the buggy input define the supposed behavior of the program execution with the buggy input. In reality, what commonly happens is that the program requirements vis-a-vis existing features do not change (although new features may be added). In such a case, our assumption is guaranteed to be satisfied. In fact, a typical scenario where *DARWIN* is applicable may be described as follows. A program version $P$ evolves to a new program version $P'$ because the customers want some new features to be added. However, in trying to program the new features, the code for the old features mistakenly gets affected. Thus, a test case $t$, which used to pass in program $P$, fails in the new program $P'$. In other words, in going from program $P$ to program $P'$, there is code evolution but no evolution of requirements. The requirements for the old features (those supported by both $P$ and $P'$) remain unchanged. *DARWIN* is most suited to explain and root-cause such errors resulting from *code evolution*.

Note that the preceding assumption does not conflict with our claim that *DARWIN* works with two different implementation of the same specification. Suppose $P$ and $P'$ are two different implementations, such as `miniweb` and `apache`. As long as the

behavior of the buggy input is supposed to be the same in both $P$ and $P'$, we can use $P$ as a reference implementation to debug $P'$.

To illustrate the issue with a more concrete example, consider a banking system $P$ supporting some basic features like "login", "logout", "view balance", and so on. Suppose now the customers of the banking system demand a new feature for transferring funds between accounts. In trying to implement this system and produce a new banking system $P'$, the programmer may make mistakes and incorrectly modify the account balance. As a result, the "view balance" functionality, which used to work correctly earlier, may not work correctly any more, leading to an observable error. *DARWIN* is most suited for explaining the root cause of such observable errors. Consider an alternate scenario in which the requirements of the banking system itself are being changed. Suppose the "view balance" functionality used earlier to view of the account balance is now changed to display the account balance for current accounts and the account balance minus $50 (the minimum deposit) for a savings accounts. In this situation, the requirements of the "view balance" feature itself have changed. *DARWIN*'s approach is not suited to explain any errors resulting from such evolution of software requirements.

—Path conditions serve as the basis of our debugging technique. In particular, the approach hinges on the observation that the path conditions $f$ and $f'$ of the test input $t$ being debugged are different in the two program versions. What if $f$ and $f'$ are logically equivalent? This means that the effect of the error being debugged is not observable by a difference in control flow. Our DARWIN approach is not inherently suited to explain such errors. Thus, the approach is most suited for explaining errors that manifest as changes in control flow. In Section 5, we proposed some methods for introducing more control-flow paths to handle assignment errors that do not affect control flow. Even with heavy instrumentation, our solution cannot guarantee that all such errors will be correctly diagnosed.

Apart from the assignments discussed in Section 5, some other program elements, such as function pointers, cannot affect the path conditions either. Some ideas similar to those in Section 5 could be used to introduce more branches. We could also control the compilation process to avoid optimizations that remove branches. For example, switch cases should be compiled into conditional jumps instead of direct jumps using jump tables.

—Regarding the scalability of our technique, the size of the generated SMT formula largely depends on the number of tainted instructions in the execution trace. This is because only the tainted instructions are analyzed in the path condition generation and all subsequent steps of our tool. From our experience in the experiments, we found that the number of tainted instructions depends on the input size, as well as the size of the program. Since SMT solving is extensively used in our approach, the scalability of our approach is also tied to the scalability of the SMT solvers.

—Finally, there are some limitations regarding our experiments. Long program executions with large input size would produce large SMT formulas. We did not perform experiments on programs of this kind. For errors in assignment, one may need to follow dependency links to find the root cause if our instrumentation technique in Section 5 is not used. Some manual code inspection is needed in this case. We did not perform any case studies to evaluate this manual effort. However, as suggested by the results in Section 7.7, instrumentation overhead is affordable. Therefore, users could employ the instrumentation technique to expose errors in assignment.

## 10. CONCLUDING REMARKS

In this article, we presented *DARWIN*, a debugging methodology and tool for evolving programs. *DARWIN* takes in two programs and explains the behavior of a test input

which passes in the stable program, while failing in the buggy program. The stable program and buggy program can be two completely different implementations of the same specification. *DARWIN* handles hard-to-explain code-missing errors inherently, by pointing to code in the stable program. We have conducted experiments using four real-world applications, such as the Apache Web server, libPNG (a library for manipulating PNG images), and TCPflow (a program for displaying data sent through TCP connections). Our experience with real-life case studies demonstrates the utility of our method for localizing real bugs.

Developers are often faced with hard-to-locate bugs when a large software system changes from one version to another. As long as the program requirements vis-a-vis existing features do not change, *DARWIN* can truly be a useful automatic debugging assistant for developers.

The alternate inputs generated by our method can also help detect new errors, apart from localizing a given observable error. This can also help test-suite augmentation of evolving programs, that is, when a program changes, we can find out potentially new test cases to be tested for stressing the change.

## REFERENCES


AGRAWAL, H. AND HORGAN, J. R. 1990. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'90)*. ACM Press, New York, NY, 246–256.

APACHE. 2009. Apache Web server. http://httpd.apache.org/.

APIWATTANAPONG, T., ORSO, A., AND HARROLD, M. 2004. A differencing algorithm for object-oriented programs. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, Los Alamitos, CA.

BALL, T., NAIK, M., AND RAJAMANI, S. 2003. From symptom to cause: Localizing errors in counterexample traces. In *Proceedings of the International Symposium on Principles of Programming Languages (POPL)*. ACM Press, New York, NY.

BARRETT, C. AND TINELLI, C. 2007. CVC3. In *Proceedings of the 19th International Conference on Computer-Aided Verification*. 298–302.

BRUMLEY, D., CABALLERO, J., LIANG, Z., NEWSOME, J., AND SONG, D. 2007. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In *Proceedings of the USENIX Security Conference.* USENIX Association, Berkeley, CA.

BRUMMAYER, R. AND BIERE, A. 2009. Boolector: An efficient smt solver for bit-vectors and arrays. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09)*. 174–177.

BRUTTOMESSO, R., CIMATTI, A., FRANZÉN, A., GRIGGIO, A., AND SEBASTIANI, R. 2008. The MathSAT 4 SMT Solver. In *Proceedings of the International Conference on Computer Aided Verification*. 299–303.

CHEN, Y., ROSENBLUM, D., AND VO, K. 1994. Testtube: A system for selective regression testing. In *Proceedings of the International Conference on Software Engineering*. IEEE Computer Society Press, Los Alamitos, CA.

CSALLNER, C. AND SMARAGDAKIS, Y. 2006. DSD-Crasher: A hybrid analysis tool for bug finding. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. ACM Press, New York, NY.

DE MOURA, L. AND BJORNER, N. 2008. Z3: An efficient SMT solver. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.

ELBAUM, S., MALISHEVSKY, A., AND ROTHERMEL, G. 2000. Prioritizing test cases for regression testing. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. ACM Press, New York, NY.

GANESH, V. AND DILL, D. L. 2007. A decision procedure for bit-vectors and arrays. In *Proceedings of the Computer Aided Verification Conference (CAV)*. 524–536. Available online at http://sites.google.com/site/stpfastprover/.

GIROUX, O. AND ROBILLARD, M. P. 2006. Detecting increases in feature coupling using regression tests. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT'06/FSE-14)*. ACM Press, New York, NY, 163–174.

GODEFROID, P., KLARLUND, N., AND SEN, K. 2005. DART: Directed automated random testing. In *Proceedings of the Conference on Programming Languages Design and Implementation (PLDI)*. ACM Press, New York. NY.

GUO, L., ROYCHOUDHURY, A., AND WANG, T. 2006. Accurately choosing execution runs for software fault localization. In *Proceedings of the International Conference on Compiler Construction (CC)*.

HOROWITZ, S. 1990. Identifying the semantic and textual differences between two versions of a program. In *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, New York, NY.

HOVEMEYER, D. AND PUGH, W. 2004. Finding bugs is easy. In *Proceedings of the Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'04)*. ACM Press, New York, NY, 132–136.

HUANG, S. 2009. Miniweb Web server. http://miniweb.sourceforge.net/.

JACKSON, D. AND LADD, D. A. 1994. Semantic diff: A tool for summarizing the effects of modifications. In *Proceedings of the International Conference on Software Maintenance*. 243–252.

KOREL, B. AND LASKI, J. W. 1988. Dynamic program slicing. *Inform. Process. Letters 29,* 3, 155–163.

LIBLIT, B. 2005. Cooperative bug isolation. Ph.D. dissertation, UC Berkeley.

LIBLIT, B., NAIK, M., ZHENG, A., AIKEN, A., AND JORDAN, M. 2005. Scalable statistical bug isolation. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, New York, NY.

LIBPNG. 2009. libPNG library. http://www.libpng.org.

PERSON, S., DWYER, M., ELBAUM, S., AND PASAREANU, C. 2008. Differential symbolic execution. In *Proceedings of the International Conference on Foundations of Software Engineering (FSE)*. ACM Press, New York, NY.

QEMU. 2009. QEMU emulator. http://www.qemu.org.

QI, D., ROYCHOUDHURY, A., LIANG, Z., AND VASWANI, K. 2009. Darwin: An approach for debugging evolving programs. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC-FSE)*. ACM Press, New York, NY, 33–42.

RANISE, S. AND TINELLI, C. 2003. The SMT-LIB format: An initial proposal. In *Proceedings of the Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR)*.

REN, X., SHAH, F., TIP, F., RYDER, B. G., AND CHESLEY, O. 2004. Chianti: A tool for change impact analysis of java programs. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04)*. ACM Press, New York, NY, 432–448.

RENIERIS, M. AND REISS, S. P. 2003. Fault localization with nearest neighbor queries. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, Los Alamitos, CA.

ROTHERMEL, G. AND HARROLD, M. J. 1997. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methodol. 6,* 2, 173–210.

SANTELICES, R., CHITTIMALLI, P., APIWATTANAPONG, T., ORSO, A., AND HARROLD, M. 2008. Test-suite augmentation for evolving software. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, Los Alamitos, CA.

SAVANT. 2009. Savant Web server. http://savant.sourceforge.net/info.html.

SEACORD, R., PLAKOSH, D., AND LEWIS, G. 2003. *Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices*. Addison-Wesley, Boston, MA.

SEN, K., MARINOV, D., AND AGHA, G. 2005. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM Press, New York, NY, 263–272.

SILLITO, J., MURPHY, G., AND DE VOLDER, K. 2006. Questions programmers ask during software evolution tasks. In *Proceedings of the International Conference on Foundations of Software Engineering (FSE)*. ACM Press, New York, NY.

SONG, D., BRUMLEY, D., YIN, H., CABALLERO, J., JAGER, I., KANG, M. G., LIANG, Z., NEWSOME, J., POOSANKAM, P., AND SAXENA, P. 2008. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security*. Keynote invited paper.

SRIDHARAN, M., FINK, S. J., AND BODIK, R. 2007. Thin slicing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*. ACM Press, New York, NY, 112–122.

SRIVASTAVA, A. AND THIAGARAJAN, J. 2002. Effectively prioritizing tests in development environment. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM Press, New York, NY, 97–106.

WANG, T. AND ROYCHOUDHURY, A. 2004. Using compressed bytecode traces for slicing Java programs. In *Proceedings of the 26th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, 512–521.

ZELLER, A. 1999. Yesterday, my program worked. Today, it does not. Why? In *Proceedings of the 7th European Software Engineering Conference held jointly with the ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 253–267.

ZELLER, A. 2002. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*. ACM Press, New York, NY, 1–10.

ZELLER, A. AND HILDEBRANDT, R. 2002. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng. 28,* 2, 183–200.

ZHANG, X., GUPTA, N., AND GUPTA, R. 2006. Pruning dynamic slices with confidence. In *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, New York, NY, 169–180.

ZHANG, X., TALLAM, S., GUPTA, N., AND GUPTA, R. 2007. Towards locating execution omission errors. In *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, New York, NY, 415–424.

Received October 2009; revised May, November 2010, February 2011