Darwin: An Approach for Debugging Evolving Programs

Dawei Qi, Abhik Roychoudhury, Zhenkai Liang
National University of Singapore
and
Kapil Vaswani
Microsoft Research India

Bugs in programs are often introduced when programs evolve from a stable version to a new version. In this paper, we propose an 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 flow 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 these applications contain several thousands lines of code, *Darwin* could usually narrow down the potential root causes to less than 10 lines. In addition, we find that the inputs synthesized by *Darwin* provide additional value by revealing other undiscovered errors or suggesting fixes to buggy inputs.

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

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

An initial version of this paper was published as [Qi et al. 2009] in ESEC-FSE 2009. The conference paper is available from `http://www.comp.nus.edu.sg/~abhik/pdf/fse09.pdf`. Authors' e-mails: `{dawei,abhik,liangzk}@comp.nus.edu.sg, kapilv@microsoft.com` Address of the first three authors: School of Computing, National University of Singapore, Computing 1, 13 Computing Drive, Singapore 117417. Address of the fourth author: Microsoft Research India, 196/36, 2nd Main, Sadashivnagar, Bangalore 560080, India.

and defects are fixed. Often changes are made concurrently by a number of developers. It is during such changes that subtle defects are often introduced. As a result, ensuring that the system continues to meet its requirements in the presence of 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 the cost of development. The cost of maintaining a software 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 extensively been studied. 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 (*e.g.*, see [Chen et al. 1994]). However, even among the tests which are tested in both old and 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 significant 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 i.e. 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.*, see [Horowitz 1990]) have been proposed as a way for 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, 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.*, see [Zeller 1999]). While this approach is very 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 dereferenced is indicative of such a situation. The mistake may only be observed after a change which introduces a pointer dereference. 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, the trace produced by a failed input is compared with the trace produced by a passing input. 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 that is very "similar" to the failing input. However, while regression testing reveals failing inputs, it is often hard to find a similar passing input. In our problem setting, we have a stable program representing expected behavior, which we use to find such passing inputs.

Our approach. In this paper, we propose an approach (called *Darwin*) for automatically 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. 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 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 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 formulae.

An interesting aspect of *Darwin* is that we can diagnose errors even when P and P' are two completely different implementations, rather than being two versions of the same program. We only require that for the set of inputs which are common to P and P', the behavior of P, P' are expected to be "equivalent" i.e. P and P' are two implementations of the same specification. This aspect of our method is illustrated by our experiments on real world web-servers.

Contributions. We propose *Darwin*, an automated and scalable solution to a problem of locating causes of regression bugs. We demonstrate the efficacy of our approach using several 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.

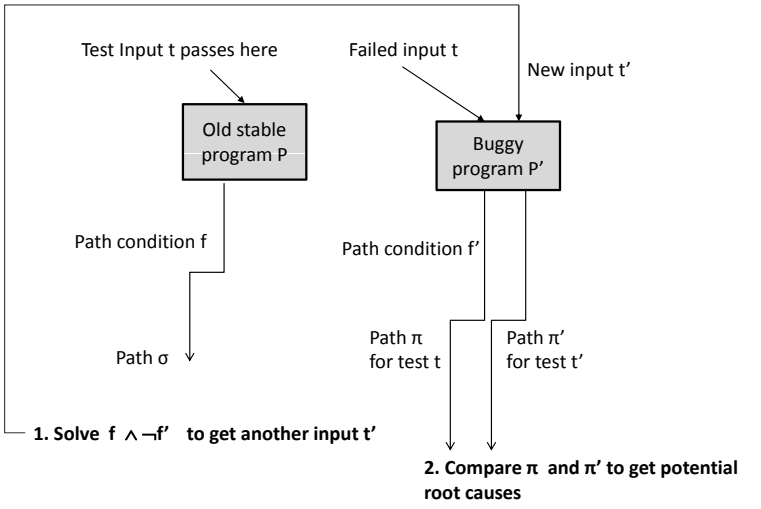

Fig. 1. Rough description of debugging method

Execution trace of $\langle x == 1, y == 1\rangle$	Concrete stores	Symbolic stores	Path condition
3 $scanf("\%d", \&x);$	$\{x \to 1, y \to undef\}$	$\{x \to x^s, y \to undef\}$	$true$
4 $scanf("\%d", \&y);$	$\{x \to 1, y \to 1\}$	$\{x \to x^s, y \to y^s\}$	$true$
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 $o = 10;$	$\{x \to 1, y \to 2\}$	$\{x \to x^s, y \to y^s + 1\}$	$(x^s > 0) \wedge (y^s + 1 > 0)$

Table I. Process of computing path condition for the program in Figure 2

2. BACKGROUND

Path condition[Godefroid et al. 2005] serves as the basis of our approach. The computation of path condition is critical to understand many aspects of our approach. When executing program P with input t, the path condition is a formula over inputs variables of P such that any inputs satisfying the path condition will follow the same path as the path of t in P.

The path condition is computed through symbolic execution. During symbolic execution, we interpret each statement and update the symbolic state to represent the effects of the statement on program variables. At every conditional branch, we compute a *branch constraint*, which is a formula over the program's input variables which must be satisfied for the branch to be evaluated in the same direction as the concrete execution. The result of symbolic execution is a path condition, which is a conjunction of constraints corresponding to all branches along the path. Any input that satisfies the path condition generated by executing an input t is guaranteed to follow the same path as t.

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 path condition is computed. We use x^s and y^s to denote the symbolic inputs of

```
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

this program. The computation process 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 $(x^s > 0) \wedge (y^s + 1 > 0)$. 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 line 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 at line 6 is considered when computing path condition. The assignment at line 6 first affects the symbolic store of y. When y is used in the condition at 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 at line 7 and therefore affect 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`. Assume this is the stable reference version. Note that `g`, `h` are functions invoked from P. The code for `g`, `h` is not essential to understanding the example. 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*

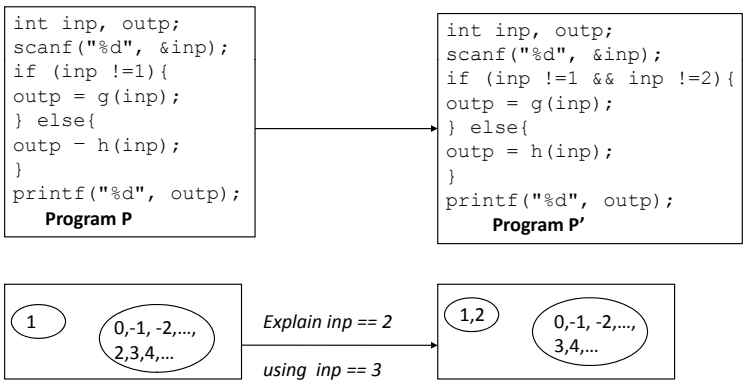

Fig. 3. Two example programs P, P' and their input space partitioning. The behavior of the input 2 changes during the change $P \rightarrow P'$. We choose an input 3 to explain the behavior of the failing input 2 since $2, 3$ are in the same partition in P, but different partitions in P'.

condition f, a formula representing 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 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 1,2. Say `inp` == 3.

—During this process of finding a satisfying instance for the formula, we find that $\neg f'$ is equivalent to $\neg(inp \neq 1 \wedge inp == 2)$ i.e. $inp == 1 \vee inp \neq 2$. 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 $inp \neq 1 \wedge inp \neq 2$. This constraint can be thought of as the reason why two similar inputs (`inp == 2` and `inp == 3`) behave differently in the buggy program. 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 $f' \equiv \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_m$. We then enumerate all possible deviations of f' namely $\neg\psi_1$, $\psi_1 \wedge \neg\psi_2$, …, $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{m-1} \wedge \neg\psi_m$. We then conjoin each of these deviations with f, producing m formulae (where m is the number of primitive constraints in f'). For each of the m formulae that are satisfiable, we consider the corresponding branch as a potential root cause. In other words, if $f \wedge \psi_1 \wedge \psi_2 \wedge \ldots \psi_{i-1} \wedge \neg\psi_i$ is satisfiable, we consider the program branch contributing to the constraint ψ_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, 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 the above 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 are

—$\neg inp \geq 1$ i.e. $inp < 1$

—$inp \geq 1 \wedge \neg inp > 9$ i.e. $1 \leq inp < 9$

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', and points us the code-missing error in this example.

The reader may think the above 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.

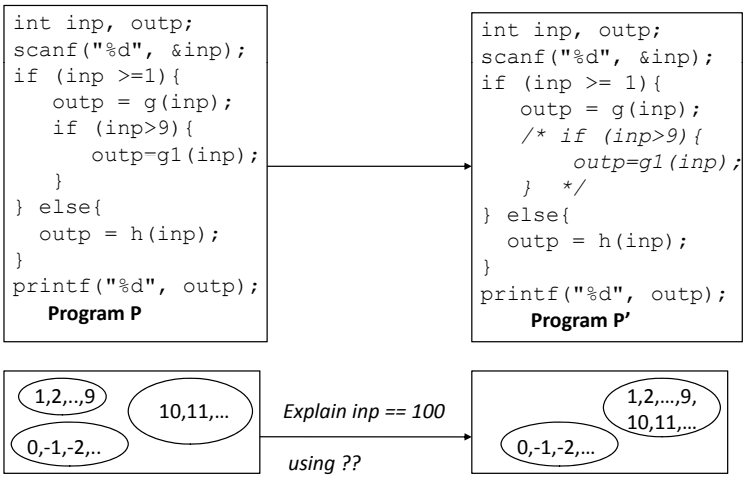

Fig. 4. Two example programs P, P' and their input space partitioning. The behavior of the input 100 changes during the change $P \rightarrow P'$. How to find an input to explain its behavior?

In summary, the outline of our method is as follows. Given a reference program version P, a new, buggy program P', 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 ψ_i' in f'. The constraint ψ_i' is the reason why f' is not satisfied, and is considered as a potential root cause. As we explained, the constraint ψ_i' is obtained by enumerating the deviations of f' and conjoining them with f. Details of the procedure for obtaining ψ_i' is 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 ψ_i. The code to ψ_i is considered as 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'.

(5) In the event $(f \wedge \neg f') \vee (f' \wedge \neg f)$ is unsatisfiable, we fail to find a potential root cause.

4. DETAILED METHODOLOGY

In this section, we elaborate on different aspects of our approach i.e. input generation, formulation 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 formulae 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.

A key property of under-approximation is that any input which satisfies the under-approximation is still guaranteed to follow the same path. However, this property does not hold in *Darwin*. Recall that we need to solve the formula $f \wedge \neg f'$ for getting an alternate program input, where f, f' are the path conditions of the input t being examined in the reference and buggy program respectively. Let $f_{computed}$, $f'_{computed}$ be the computed path conditions in the reference and buggy program 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 simply concretely executing the test inputs t, 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, f' are path conditions, they are conjunctions of primitive constraints, that is, $f' = (\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_m)$ where ψ_i are primitive constraints. Thus instead of solving $f \wedge \neg f'$ we solve the following m formulae $\{\varphi_i \mid 0 \leq i < m\}$ where

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

Each φ_i is a conjunction. A solution to any φ_i is a solution for $f \wedge \neg f'$. We solve each φ_i separately, and obtain *any one* solution of φ_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 i.e. we check via concrete execution whether they follow same path as that of t in P, and different path from that of t in P'. The reader may note our choice of φ_i, the formulae dispatched to the solver. Each φ_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 φ_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 φ_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 θ_i, say $f = (\theta_1 \wedge \theta_2 \wedge \ldots \wedge \theta_k)$ we solve the k formulae $f' \wedge \theta_1 \wedge \ldots \theta_i \wedge \neg\theta_{i+1}$ 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 to improve the efficiency of formula solving specific to our problem domain.

Checking for unsatisfiable sub-formula. First, we identify some unsatisfiable formulae using very low cost. Recall that we are trying to solve formulae 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 formulae $\varphi_i \stackrel{def}{=} f \wedge \psi_1 \wedge \ldots \psi_i \wedge \neg\psi_{i+1}$. The key problem we face now is that the SMT solver may take substantial time to solve each of the φ_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 ψ_j (for some $j \leq i$) and ψ_{i+1} are both c, an SMT solver will very quickly conclude that $\psi_1 \wedge \ldots \psi_i \wedge \neg\psi_{i+1}$ is unsatisfiable. In such 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 φ_i to the SMT solver (to check the satisfiability of φ_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. Secondly, using dynamic slicing, we can find the subset of symbolic input bytes that can affect the only branch (contributing to ψ_{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 input (for structured program inputs, the processing of two different portions of the input is usually independent). Using concrete values for certain portions of our input greatly simplifies the formulae we need to solve and reduces the burden on the SMT solver.

We now describe the steps we employ to reduce the amount of time taken in checking satisfiability of φ_i.

(1) We impose a short time bound (say 10 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 φ_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 π' corresponding to path condition f' to find out the set of input bytes that ψ_{i+1} is dependent on. This is done as follows. Note that ψ_{i+1} is a primitive constraint corresponding to some branch instance b in the execution trace. Due to traceability links between sub-formula in the path condition and branches contributing to these formulae we can find the branch b contributing to ψ_{i+1}. Let l be the control location corresponding to b and $Vars$ be the variables appearing in the constraint ψ_{i+1}. We perform dynamic slicing [Korel and Laski 1988; Agrawal and Horgan 1990; Wang and Roychoudhury 2004] w.r.t the slicing criterion $(l, Vars)$ on the assembly level execution trace π' 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), (ii) a set of variables δ whose values need to be explained. At the end of the slicing, we inspect the set of input fields (or bytes) which appear in δ. These are the input bytes on which ψ_{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 π' 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 formulae get instantiated to concrete values. Let the simplified formulae be called $f_{simplified}$ and $(\psi_1 \wedge \ldots \psi_i \wedge \neg\psi_{i+1})_{simplified}$.

(4) We check the satisfiability of $(\psi_1 \wedge \ldots \psi_i \wedge \neg\psi_{i+1})_{simplified}$. If it is unsatisfiable, we can stop. Otherwise, we go to the next (and final) step.

(5) Finally we solve the simplified formula $f_{simplified} \wedge (\psi_1 \wedge \ldots \psi_i \wedge \neg\psi_{i+1})_{simplified}$ using a SMT solver.

After concretizing the input bytes other than those in In_{i+1} and propagating constants, the formulae to be solved are greatly simplified owing to instantiation. This greatly reduces the solution time.

4.3 Backward Traceability and Input Validation

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

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

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 ψ_i appearing in f' is a primitive constraint contributed by a branch in the program P'.

Suppose the branch corresponding to ψ_{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 φ_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}

—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 the above 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 program 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 ψ_i are primitive constraints. We solve the m formulae $\{\varphi_i \mid 0 \leq i < m\}$

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

For all $0 \leq i < m$ if φ_i is satisfiable, we use backwards traceability links to find the branch b_{i+1} contributing to the primitive constraint ψ_{i+1}. We report b_{i+1} as a potential root cause if the solution for φ_i is successfully validated. For checking the satisfiability of each φ_i, we use the five optimization steps given earlier in subsection 4.2.

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

Finally, if we still obtain a large number of alternate inputs (and hence a large number of potential root causes), we prioritize them as follows. We choose the *alternate inputs which are successful*, that is, produce same outputs in both the program versions. Since such successful inputs exhibit bug-free behavior (in terms of program output), by comparing their traces with the buggy input's trace we hope to localize the error cause. The branch instruction contributed by each successful alternate input is thus prioritized over other branches.

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 formulae 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 can 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. COMMON PROGRAMMING ERRORS

We now explain the suitability of our debugging methodology for different common kind 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. This is 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 causes 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 pre-defined 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 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 being left out during the change of a program. Such code will be missing in the buggy program, but is 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 can be expected to have more paths than the buggy program P'. Given a failing test input t, and f, f' being the path condition of t in P, P', we can thus expect $f' \wedge \neg f$ to yield a solution. This will be a 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 will 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* based on 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 our 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 x86 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 [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 i.e. 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 works on x86 binaries, so our path conditions are computed at the level of binaries, rather than source code. Second, the variables appearing in the path condition correspond to the different bytes of the program input.

Given program versions P, 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'. In fact, the symbolic execution engine in BitBlaze constructs these path conditions as formulae 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 on 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

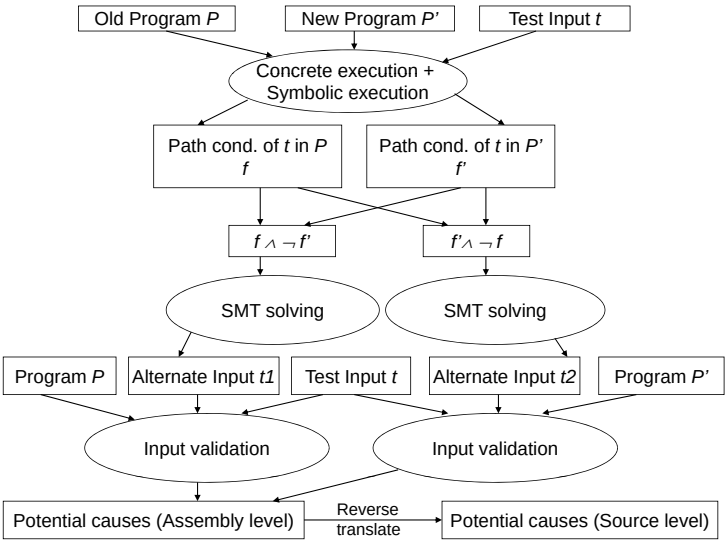

Fig. 5. Architecture of our *Darwin* toolkit. It takes an old program P, a new program P' and a test input t which passes in P but fails in P'. The output is a report explaining the behavior of test t. The entire flow is automated.

we are currently using is Boolector [Brummayer and Biere 2009], the winner of the SMT competition in 2009 for quantifier free formulae with bitvectors, arrays and uninterpreted functions (the QF_AUFBV category). Indeed this is suitable for us, since our formulae 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 $f' = (\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_m)$ where ψ_i are primitive constraints. As mentioned in the previous section, we solve the m formulae $\{\varphi_i \mid 0 \leq i < m\}$ 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 ψ_i constraints, and hence the number of φ_i formulae that need to be dispatched to the SMT solver. Since each φ_i formula contributes at most one statement in our report, we get a smaller sized report by reducing the number of φ_i. If the number of root causes is still high (due to 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 on our experience in using *Darwin* for locating 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 e-mail 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.

```
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`

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 above error. If we want to perform program differencing methods (such as source code "diff") to localize the bug, there are 1589 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 can 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 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.

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 compute 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 9 alternate inputs from the Boolector solver. These 9 alternate inputs are essentially 9 PNG images. All these 9 inputs passed validation, hence we report 9 statements as potential root causes.

We prioritize these 9 statements as follows. Among the 9 alternate inputs, we find out which of them are successful i.e., the program output for a successful input should be the same in both the program versions. Only one of our 9 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 9 alternate inputs, only one of which exhibits bug-free behavior, and pointed us to the error. Interestingly, the other branch instructions point us to other deviations between the two versions of `libPNG`.

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

For example, by following one of these 8 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 w.r.t. 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 cannot 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 5 alternate inputs and hence 5 potential root causes:

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

Based on the first of these 5 branches, we were able to localize the bug immediately. miniweb does not check for '/' in GET queries and treats the query "GET x" similar to "GET /" thereby returning the file index.html.

Discovering New Errors. Only one of our 5 alternate inputs was successful, exhibiting 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 5 characters while `miniweb` misses the check for all 5 of them, leading to potential errors.

In a Broader Perspective. Our experiments with `apache-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.

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 reference program, we choose the `apache` webserver, 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 the 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 a 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. In-

```
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

side each raw packet, we also have the header for the network layer protocol (usually the IP protocol). Thus, it is non-trivial 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 tell which packets are from the same connection manually. `TCPflow` is a program which solves these 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.

```
// 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`

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 an 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 un-patched 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, 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 0x00 (in the first byte of the buggy output) as shown in Figure 7.

Once again, we emphasize the bug we described above (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 6 statements are reported as potential root causes from and one of them points to a branch condition which checks for empty packets.

Over and above 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 1000 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 un-patched and patched versions of `TCPflow`. Assuming $f_{patched} \equiv \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_m$, we actually solve m formulae $\{\varphi_i \mid 0 \leq i < m\}$ 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 φ_i takes up to 30 minutes, and there are around 2000 φ_i formulae to solve!!

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 φ_i formulae where ψ_{i+1} corresponds to a branch in the source code. The effect of this technique is discussed in the next paragraph. By considering only φ_i formulae from the source code, there are still 86 formulae left to solve. The estimated time to solve these formulae comes to 2 days (since the solving of each φ_i formulae 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 (10 seconds for our experiments), and see how many of the φ_i formulae 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, φ_i cannot be satisfiable! We find that 64 out of the 86 formulae are proved to be unsatisfiable in this fashion. Thus, we are left with $(86 - 64)$, that is, 22 formulae to solve. The time to solve these formulae 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 10 minutes. The total debugging time (which includes tracing as well) comes to 33 minutes. The final result from *Darwin* contains only 6 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 where 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 [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 Fig. 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

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

Fig. 9. Injected bug in `TCPflow`

Programs	LOC	Trace size (# instructions)	# Branches in trace	# Tainted instructions
libPNG v1.0.7	31,164	87,336	13,635	2999
libPNG v1.2.21	36,776	108,769	15,472	2592
Miniweb	2,838	270,856	26,201	331
Savant	8,730	121,714	16,212	1613
Apache	358,379	60,380 (miniweb)	5,388 (miniweb)	264 (miniweb)
		74,002 (savant)	9,672 (savant)	6889 (savant)
TCPflow (unpatched)	895	56838	7210	7753
TCPflow (patched)	934	58079	7375	7860

Table II. Properties of the subject programs

Programs	Time in step 1	Time in step 2	Time in step 3	Time in steps 4&5	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	20m12s	3m32s	31m 50s

Table III. Performance of *Darwin*ś extended debugging method (m=minutes, s=seconds)

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 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 φ_i formulae 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 10 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 Time Taken by 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 five 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) concretize all the inputs that are not in the slicing result and perform constant propagation, (iv) check the satisfiability of the simplified formula $\psi_1 \wedge \ldots \psi_i \wedge \neg\psi_{i+1}$ after constant propagation (v) solving the simplified formula $f \wedge \psi_1 \wedge \ldots \psi_i \wedge \neg\psi_{i+1}$ (if the simplified $\psi_1 \wedge \ldots \psi_i \wedge \neg\psi_{i+1}$ is found to be satisfiable in step iv).

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 formulae φ_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 7 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 is less then 2 minutes in all the case studies. In the third step, we concretize all the irrelevant input bytes and perform constant propagation to simplify the formulae. The time taken by this step was less than 21 minutes in all our case studies. In the last two steps, we first check the satisfiability of the simplified formula $\psi_1 \wedge \ldots \psi_i \wedge \neg\psi_{i+1}$ after constant propagation. If it is satisfiable, we solve the whole formula $f \wedge \psi_1 \wedge \ldots \psi_i \wedge \neg\psi_{i+1}$ (which also has been greatly simplified by now due to constant propagation). The time taken by this step was less than 4 minutes in all the case studies.

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 the following —

—function return values at each function return site, and

—binary constraints describing 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.

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

Table IV. Overhead of Predicate Instrumentation

8. RELATED WORK

Validation of evolving programs is an important problem, since any large software moves from one version to another. Among the established efforts in this direction are the work on regression testing which focus 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 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 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] — ordering tests in existing test suite to better meet testing objectives of the changed program. Finally, [Santelices et al. 2008] has studied test-suite augmentation — 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 — 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 to explore 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 in [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.*, see [Ren et al. 2004] — where the analysis identifies not only the changes, but also which tests are affected by which changes). A recent work [Person et al. 2008] 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 via regression testing the change in dependencies between parts of a program). 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.*, see [Brumley et al. 2007]). This problem involves finding corner test inputs in which two implementations of the same protocol might "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 superficially [Brumley et al. 2007] appears to employ techniques similar to ours — the goal of [Brumley et al. 2007] is to generate a deviating program input which can demonstrate the behavior difference 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 section of works address the problem of software fault localization (typically via dynamic analysis) — 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. We now 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; Zhang et al. 2007]), or (b) comparison of the failing program execution with the set of all "correct" executions (*e.g.*, see [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.

The work of [Zeller 1999] studies debugging of evolving programs and proposes to identify 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 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 paper. This also clarifies any implicit assumptions on which our debugging method may be built.

—One key assumption of our approach is 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 above 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 where the requirements of the banking system itself being changed. Suppose the "view bal-

ance" functionality earlier used to be interpreted as viewing of the account balance, and is now changed to display the account balance for current accounts and displays the account balance minus $50 (the minimum deposit) for savings accounts. In this situation, the requirements of the "view balance" feature itself has changed. *Darwin* 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, 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 to introduce 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 can 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 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 directly tied to the scalability of the SMT solvers. We believe that there are generally two ways to increase the scalability of our approach. First, we can use various methods to reduce our SMT formula size. The high-level idea is to remove something unrelated. In the paper, we have presented a means that concretizes some unrelated input bytes and backward slices out unrelated components. For a particular program, the user may know which modules/functions are trustable. These information can be used to reduce the formula size further. If a program has large structured input, the technique from [Zeller and Hildebrandt 2002] would be useful to simplify the input before applying our tool. Secondly, the scalability of the SMT solvers is increasing all the time. This could also benefit our approach.

—Although our *Darwin* tool is built based on the C binary executables, our technique is generalizable to other languages. As long as the errors can affect program flow and program requirements vis-a-vis the buggy input are the same, our technique should apply.

—Finally, there are some limitations regarding our experiments. Long program execution with large input size would produce large SMT formulae. We did not perform any experiments on programs of this kind. For programs with large structured inputs, we suggested that some input simplification techniques should be adopted. We did not perform any experiments to evaluate the effectiveness of these simplification techniques on *Darwin*. 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 result in subsection 7.7, the instrumentation overhead is affordable. Therefore, users could employ the instrumentation technique to expose errors in assignment.

10. CONCLUDING REMARKS

In this paper, 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 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). 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 an 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 — when a program changes we can find out potentially new test cases to be tested for stressing the change.

ACKNOWLEDGMENTS

This work was partially supported by a Defense Innovative Research Programme (DIRP) grant (R-252-000-393-422) from Defence Research and Technology Office (DRTech). The second author was on sabbatical leave to Microsoft Research India during part of this work.

REFERENCES

AGRAWAL, H. AND HORGAN, J. R. 1990. Dynamic program slicing. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*. ACM, New York, NY, USA, 246–256.

APACHE. 2009. Apache webserver. `http://httpd.apache.org/`.

APIWATTANAPONG, T., ORSO, A., AND HARROLD, M. 2004. A differencing algorithm for object-oriented programs. In *ASE: International Conference on Automated Software Engineering*. IEEE Computer Society, Washington, DC, USA.

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

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 *USENIX Security Conf.* USENIX Association, Berkeley, CA, USA.

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

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

CSALLNER, C. AND SMARAGDAKIS, Y. 2006. DSD-Crasher: a hybrid analysis tool for bug finding. In *ISSTA: International Symposium on Software Testing and Analysis*. ACM, New York, NY, USA.

ELBAUM, S., MALISHEVSKY, A., AND ROTHERMEL, G. 2000. Prioritizing test cases for regression testing. In *ISSTA: International Symposium on Software Testing and Analysis*. ACM, New York, NY, USA.

GIROUX, O. AND ROBILLARD, M. P. 2006. Detecting increases in feature coupling using regression tests. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, New York, NY, USA, 163–174.

GODEFROID, P., KLARLUND, N., AND SEN, K. 2005. DART: Directed automated random testing. In *PLDI*. ACM, New York, USA.

GUO, L., ROYCHOUDHURY, A., AND WANG, T. 2006. Accurately choosing execution runs for software fault localization. In *CC: International Conference on Compiler Construction*. Springer, Berlin, Heidelberg.

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

HOVEMEYER, D. AND PUGH, W. 2004. Finding bugs is easy. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. ACM, New York, NY, USA, 132–136.

HUANG, S. 2009. Miniweb webserver. `http://miniweb.sourceforge.net/`.

JACKSON, D. AND LADD, D. A. 1994. Semantic diff: a tool for summarizing the effects of modifications. In *Proc. Conf. Int Software Maintenance*. 243–252.

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

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

LIBLIT, B., NAIK, M., ZHENG, A., AIKEN, A., AND JORDAN, M. 2005. Scalable statistical bug isolation. In *PLDI*. ACM, New York, NY, USA.

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

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

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 *ESEC-FSE: Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, New York, 33–42.

RANISE, S. AND TINELLI, C. 2003. The SMT-LIB format: An initial proposal. 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 *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM, New York, NY, USA, 432–448.

RENIERIS, M. AND REISS, S. P. 2003. Fault localization with nearest neighbor queries. In *ASE: International Conference on Automated Software Engineering*. IEEE Computer Society, Washington, DC, USA.

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 *ASE: International Conference on Automated Software Engineering*. IEEE Computer Society, Washington, DC, USA.

SAVANT. 2009. Savant webserver. `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, USA.

SEN, K., MARINOV, D., AND AGHA, G. 2005. Cute: a concolic unit testing engine for c. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, New York, NY, USA, 263–272.

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

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.* Springer-Verlag, Hyderabad, India.

SRIDHARAN, M., FINK, S. J., AND BODIK, R. 2007. Thin slicing. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*. ACM, New York, NY, USA, 112–122.

SRIVASTAVA, A. AND THIAGARAJAN, J. 2002. Effectively prioritizing tests in development environment. In *ISSTA: Proceedings of the ACM SIGSOFT International Symposium on Software testing and analysis*. ACM, New York, NY, USA, 97–106.

WANG, T. AND ROYCHOUDHURY, A. 2004. Using compressed bytecode traces for slicing Java programs. In *ICSE: Proceedings of the 26th International Conference on Software Engineering*. IEEE Computer Society, Washington, DC, USA, 512–521.

ZELLER, A. 1999. Yesterday, my program worked. today, it does not. Why? In *ESEC/FSE-7: 7th European software engineering conference held jointly with the ACM SIGSOFT international symposium on Foundations of software engineering*. Springer-Verlag, London, UK, 253–267.

ZELLER, A. 2002. Isolating cause-effect chains from computer programs. In *SIGSOFT '02/FSE-10: 10th ACM SIGSOFT symposium on Foundations of software engineering*. ACM, New York, NY, USA, 1–10.

ZELLER, A. AND HILDEBRANDT, R. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering 28,* 2, 183–200.

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

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