DARWIN: An Approach for Debugging Evolving Programs

Dawei Qi[†], Abhik Roychoudhury[†], Zhenkai Liang[†], Kapil Vaswani[§]
[†]National University of Singapore [§]Microsoft Research India
{dawei,abhik,liangzk}@comp.nus.edu.sg kapilv@microsoft.com

ABSTRACT

Debugging refers to the laborious process of finding causes of program failures. Often, such failures are introduced when a program undergoes changes and evolves from a stable version to a new, modified version. In this paper, we propose an automated approach for debugging evolving programs. Given two programs (a reference, stable program and a new, modified program) and an input that fails on the modified program, our approach uses concrete as well as symbolic execution to synthesize new inputs that differ marginally from the failing input in their control flow behavior. A comparison of the execution traces of the failing input and the new inputs provides critical clues to the root-cause of the failure. A notable feature of our approach is that it handles hard-to-explain bugs like code missing errors by pointing to the relevant code in the reference program. We have implemented our approach in a tool called DARWIN. We have conducted experiments with several real-life case studies, including real-world web servers and the libPNG library for manipulating PNG images. Our experience from these experiments points to the efficacy of DARWIN in pinpointing bugs. Moreover, while localizing a given observable error, the new inputs synthesized by DARWIN can reveal other undiscovered errors.

Categories and Subject Descriptors

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

General Terms

Experimentation, Reliability

Keywords

Software Evolution, Debugging, Symbolic Execution

1. INTRODUCTION

Programmers do not write programs entirely from scratch. Rather, over time, a program gradually evolves. In industrial software development projects, this complexity of software evolution is explicitly managed via check-in of program versions. Validation of such evolving programs (say, to address possible bugs introduced via program changes) remains a huge problem in terms of software development. This adds to the cost for software maintenance, which is much larger than the initial software development cost. The cost of maintaining a software and managing its evolution is said to account for more than 90% of the total cost of a software project, prompting authors to call it the "*legacy crisis*" [26].

To tackle the ever-growing problem of software evolution and maintenance, software testing methodologies have long been studied. Regression testing is a well-known concept which is currently employed in any software development project. 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 [10]). 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 major headache! In this paper, we employ dynamic analysis to address this issue.

Problem statement. The problem we tackle can be summarized as follows. Consider a program P accompanied by a test-suite T, such that P passes all the tests in T. We call P as the old program or the stable program since it passes all the tests, that is, the observable output of P for all the tests in T is as expected by the programmer. Suppose P changes to a new program P' and certain tests in T now fail, that is, their output does not meet the programmer's expectations. Let $t \in T$ be such a test. Our goal is to produce a bug report $Rep(t, P, P')$ such that Rep contains the explanation of why t fails in P' while passing in P. Of course, we want to construct a bug report which is as concise as possible, while pinpointing the error cause statements to the programmer. Program debugging methods often highlight a fragment of the program being debugged as *bug report*. So, we can expect $Rep(t, P, P')$ to be a fragment of the buggy program P' and/or stable program P. For our proposed solution to work, P and P' need not even be versions of the same program. They might simply be *two completely different implementations* — P being a reference implementation and P' being an optimized implementation. We only require the following — for the set of inputs which are common to P and P', the behavior of P, P' are expected to be "equivalent", that is, P and P' are two implementations of the same specification.

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

ESEC-FSE'09, August 23–28, 2009, Amsterdam, The Netherlands.

Change inspection. Since by changing a stable program P to a new program P' we cause certain tests to fail, one possibility is to find the failure inducing changes (*e.g.*, see [31]). However, the actual bug may be in P, but it could be manifested by the change from P to P'. A pointer which is mistakenly set to null in P but never dereferenced is indicative of such a situation. The mistake may only be observed in P' where the pointer is dereferenced. A good bug report should pinpoint the control location where the pointer is mistakenly set to null, not the control location where it is dereferenced. Moreover, a search for failure inducing changes will not work if P and P' are two completely 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 test input (whose behavior is unexpected) is compared with the trace produced by a successful test input (whose behavior is as expected). Techniques have been developed to determine (a) which successful test input to use (*e.g.*, [17]), and (b) how to compare and report the differences between two program executions (*e.g.*, [32]). In these approaches, the successful test input is supposed to capture "bug-free" program behavior. In our problem setting, we have a stable program representing bug-free behavior, which should be used and exploited within the debugging method.

Basic idea behind our approach. Given a stable program P, a buggy program P' and a test input t which passes in P and fails in P', we compare the trace produced by t in P' with the trace produced by *another* test-case t' in P'. We automatically generate a test 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 a test t' can be found by computing the path conditions of t in P and P'. Since t' and t follow the same program path in P – the behavior of t, t' are supposed to be "similar" in P (the stable program version). However, since t, t' follow different program paths in P' — their behaviors "differ" in P' (the buggy new version). By computing and highlighting the differences in their behavior, we highlight the possible causes of the error exposed by test-case t. A pictorial description of the debugging method appears in Figure 1. As we will see in the next section, this is only the core method and needs extensions. Our solution can also be used to debug errors in the situation where P, P' are two completely different implementations (of the same specification), rather than being two versions of the same program. This feature of our method is shown by our experiments on web-servers.

Thus, our approach works in two phases. In the first phase, we collect and suitably compose the path conditions of the failed test input in the two programs to generate alternate test inputs. In the second phase, we compare the traces of the alternate test inputs with the trace of the failed test input to produce a bug report.

Contributions. We provide an automated and scalable solution to a problem faced by any development team — locating causes of regression bugs. Efficacy of our debugging method is demonstrated on (i) a case study involving `libPNG` — a widely used open-source library for the Portable Network Graphics (PNG) image format, and (ii) case studies involving real-life webservers such as `miniweb`, `savant` and `apache`. Further, the alternate test 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.

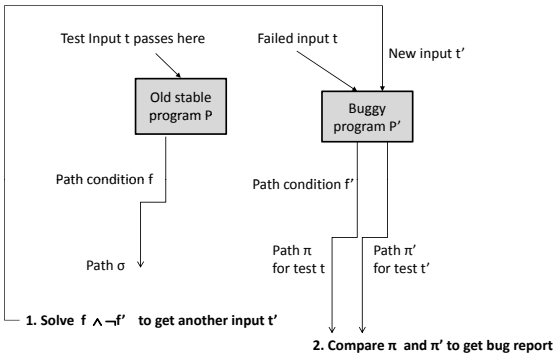

Figure 1: Pictorial description of basic debugging method

2. OVERALL APPROACH

To start with, consider a program fragment with an integer input variable `inp` – the program P in Figure 2. This is the old program version. Note that `g`, `h` are functions invoked from P. The code for `g`, `h` is not essential to understanding the example, and hence is not given. Suppose the program P is slightly changed to the program P' in Figure 2, thereby introducing a "bug". Program P' is the new program version. As a result of the above bug, certain test inputs which passed in P may fail in P'. One such test input is `inp == 2` whose behavior is changed from P to P'. Now suppose the programmer faces this failing test input and wants to find out the reason for failure. Our core method works as follows.

- We run program P for test input `inp == 2`, and calculate the resultant *path condition* f, a formula representing set of inputs which exercise the same path as that of `inp == 2` in program P. In our example, the path condition f is $inp \neq 1$.
- We also run program P' for test input `inp == 2`, and calculate the resultant *path condition* f', a formula representing set of inputs which exercise the same path as that of `inp == 2` in program P'. In our example, the path condition f' is $\neg(inp \neq 1 \wedge inp \neq 2)$.
- We solve the formula $f \wedge \neg f'$. Any solution to the formula is a test input which follows the same path as that of the test input `inp == 2` in the old program P, but follows a different path than that of the test input `inp == 2` in the new program P'. In our example $f \wedge \neg f'$ is

  $$inp \neq 1 \wedge (inp \neq 1 \wedge inp \neq 2)$$

  A solution to this formula is any value of `inp` other than 1,2 — say `inp == 3`.
- Finally, we compare the trace of the test input being debugged (`inp == 2`) in program P', with the trace of the test input that was generated by solving path conditions (here `inp == 3`). By comparing the trace of `inp == 2` with the trace of `inp == 3` in program P' we find that they differ in the evaluation of the branch `inp !=1 && inp !=2`. Hence this branch is highlighted as the *bug report* — the reason for the test input `inp == 2` failing in program P'.

The above example clarifies the idea 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

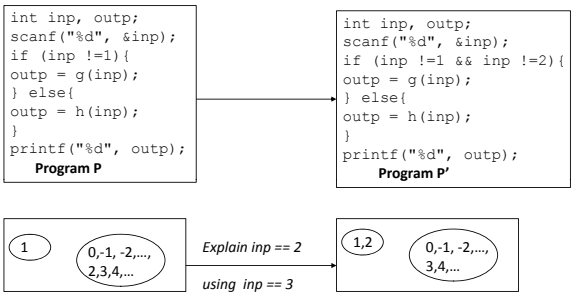

Figure 2: Two example programs P, P' and their input space partitioning. The behavior of the input 2 changes during the change $P \rightarrow P'$. We choose an input 3 to explain the behavior of the failing input 2 — since $2, 3$ are in the same partition in P, but different partitions in P'.

2 illustrates this partitioning and partition migration. The behavior of the failing input `inp == 2` is explained by comparing its trace with the trace of `inp == 3`, an input in a different partition in the new program P'. Furthermore, `inp == 3` and `inp == 2` lie in the same partition in the old program P.

Sometimes, given two program versions P, P' and a test input t which passes in P and fails in P' — we may not find any alternate input by solving $f \wedge \neg f'$. Consider the example programs in Figure 3 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

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

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 conducting our debugging in the old 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 old program version. We can now compare the traces of t and t' in the old program version to find the error root cause.

In our example Figure 3, we have

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

that is, $1 \leq inp \leq 9$. The solutions to this formula are the values $1, 2, \ldots, 9$ for the variable `inp`. These can serve as alternate inputs. Comparing the trace of any of these alternate inputs with the trace of `inp == 100` in the old program P, points us to the branch `inp > 9`. 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 new program, we may return a fragment of the old program as bug report! But, indeed this is our thesis — the bug report returned by our debugging method will help the application programmer comprehend the *change* from the old program to the new program, thereby helping him/her comprehend the new program.

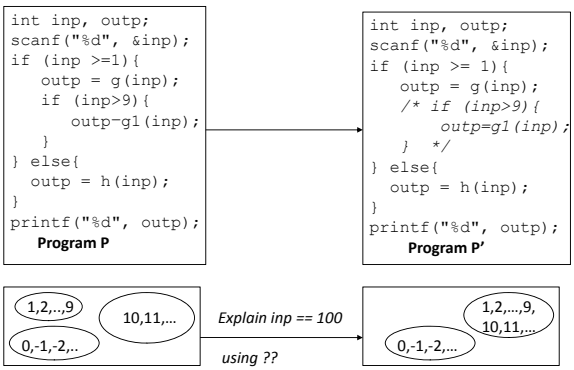

Figure 3: Two example programs P, P' and their input space partitioning. The behavior of the input 100 changes during the change $P \rightarrow P'$. How to find an input to explain its behavior?

In summary, the outline of our method is as follows. Given an old program version P, a new program version 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'. Compare the trace of t' in P' with the trace of t in P'. Return bug report.
4. If $f \wedge \neg f'$ is unsatisfiable, find a solution to $f' \wedge \neg f$. This produces a test input t'. Compare the trace of t' in P with the trace of t in P. Return bug report.

It is noteworthy that $(f \wedge \neg f') \vee (f' \wedge \neg f)$ should be satisfiable, and hence we should get an alternate input from the steps given in the preceding. If $(f \wedge \neg f') \vee (f' \wedge \neg f)$ is unsatisfiable, the formula $(f \Leftrightarrow f')$ is valid — which means that the input space partition containing the failed test input t remains unchanged while going from old program to new program.

3. DETAILED METHODOLOGY

In this section, we elaborate on the different steps of our method — alternate input generation, trace comparison, and finally bug report construction.

3.1 Generating Alternate Inputs

In this phase, we need to execute the test input under examination t in both the program versions. We first concretely execute t, record the trace, and then perform symbolic execution on the recorded trace. During the symbolic execution of t along a program path π, we accumulate a symbolic formula characterizing the set of inputs that exercise the path π. This symbolic formula is the path condition of path π, the condition under which path π is executed. Each byte of the program input is a symbolic variable in the formula. It is worth mentioning that our path conditions are calculated on the *program binary*, rather than the source code.

One issue that arises in the accumulation of path conditions is their solvability by constraint solvers. In general, we have to assume that the path condition calculated for a path π is an *under-approximation* of the actual path condition. Usually such an under-approximation is achieved by instantiating some of the variables in the actual path condition. Recall that, we need to solve the formula $f \wedge \neg f'$ for getting an alternate program input, where

f, f' are the path condition of the test input t being examined in the old and new program version respectively. Let $f_{computed}$, $f'_{computed}$ be the computed path conditions in the old and new program versions respectively. In general, the computed f, f' will be an under-approximation of the actual path conditions. Thus $f_{computed} \Rightarrow f$ and $f'_{computed} \Rightarrow f'$. As a result, we cannot ensure that $f_{computed} \wedge \neg f'_{computed}$ is an under approximation of $f \wedge \neg f'$. Hence, after solving $f_{computed} \wedge \neg f'_{computed}$ if we find a solution t' we also perform a *validation* on t'. The validation will ensure our required properties, namely: t, t' follow the same program path in the old program version, and follow different paths in the new program version. Such a validation can be performed simply by concrete execution of test inputs t, t' in the old and new program versions. 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"; 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 ith 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 ith 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 $f' \wedge \neg f$ also in a similar way. 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 $0 \leq i < k$.

Reducing formula size. To reduce the size of the formulae φ_i dispatched to the solver, we adopt a few optimization techniques. First of all, during symbolic execution we only consider instructions which *"depend" on the program input*. These dynamic dependencies are computed during the concrete execution, prior to the symbolic execution. Further, while solving φ_i, we do not submit the entire formula φ_i to the constraint solver. Recall that f, being a path condition, is a conjunction of primitive constraints. We first find the variables Var_i appearing in $\psi_1, \ldots, \psi_{i+1}$. We then perform a least fixed-point computation to find those variables appearing in f which are (directly/indirectly) "affected" by Var_i — they appear with Var_i, or some variable affected by Var_i in a primitive constraint. Of course all variables in Var_i are affected by Var_i by default. Now, only the primitive constraints of f which contain variables "affected" by Var_i need to be considered while solving φ_i. As an example suppose f is $x > y \wedge y > 10 \wedge z > w \wedge w > 0$ and ψ_1 is $x > 100$. While solving $\varphi_1 = f \wedge \neg\psi_1$ we can then only solve for $x > y \wedge y > 10 \wedge \neg(x > 100)$ to get the new solutions for x, y from the solver. The solutions for the other variables (in this case z, w) are unchanged — these are obtained from the failing test input which generated the path condition f in the first place. The optimization mentioned here substantially cuts down the size of formulae submitted to the solver, and the solution space that needs to be explored by solver.

Why Trace Comparison is Necessary. If our symbolic execution engine maintained traceability links between the path condition sub-formulae and the program fragments contributing such sub-formulae, we could simply construct the bug report as follows — (i) solve the m formulae $\{\varphi_i \mid 0 \leq i < m\}$, (ii) for all $0 \leq i < m$ if φ_i is satisfiable — put the branch contributing to the ith primitive constraint of path condition f' into the bug report. However, we cannot relate the sub-formulae of a path condition to a branch in the program. Hence we need to construct alternate inputs and align their traces with the trace of the buggy input.

3.2 Comparing Traces

In the second phase of our method, we compare the traces of two program inputs. The two test inputs are (a) the test input under examination t, and (b) an alternate test input t' generated in the first phase.

Comparison of program traces has been widely studied in software debugging, and various distance metrics have been proposed. Usually, these metrics choose an important characteristic, compute this characteristic for the two traces and report their difference as the bug report. Commonly studied characteristics (for purposes of debugging via trace comparison) include set of executed statements in a trace, set of executed basic blocks in a trace, sequence of executed branches in a trace, and so on. A sequence-based difference metric (which captures sequence of event occurrences in an execution trace) may distinguish execution traces with relatively greater accuracy. In our work, we adopt a difference metric focusing on sequence of executed branches in a trace, but apply it for traces at the instruction level. After collecting and comparing the traces at the instruction level, we can report back the instructions appearing in the "difference" between the two traces at the source-code level for the convenience of the programmer.

We represent each trace as a string of instructions executed. In practice, we need not record every instruction executed; storing the branch instruction instances (and their outcomes as captured by the immediate next instruction) suffices. Given test inputs t and t', a comparison of the traces for these two inputs amounts to finding branches which are executed with similar history in both the traces, but are evaluated differently. In order to find branches with similar history in both the traces, we employ string alignment algorithms employed on DNA sequences in computational biology (*e.g.*, see [9]). These methods produce an alignment between two strings by computing their "minimum edit distance".

To illustrate the details of our trace comparison method, consider the program fragment in Figure 4. This program is taken from a faulty version of the `replace` program from Software-artifact infrastructure repository (SIR) [13], simplified here for illustration. This piece of code changes all substrings s_1 in string `lin` matching a pattern to another substring s_2. Here variable `i` represents the index to the first un-processed character in string `lin`, variable `m` represents the index to the end of a matched substring s_1 in string `lin`, and variable `lastm` records variable `m` in the last loop iteration. The bug in the code lies in the fact that the branch condition in line 3 should be `if (m >= 0) && (lastm != m)`. At the ith iteration, if variable `m` is not changed at line 2, line 3 is wrongly evaluated to true, and substring s_2 is wrongly returned as output, deemed by the programmer as an observable error.

```
1.   while (lin[i] != ENDSTR) {
2.     m= ...
3.      if (m >= 0) {
4.        ...
5.        lastm = m;
6.      }
7.      if ((m == -1) || (m == i)){
8.        ...
9.        i = i + 1;
10.     }
11.     else{
12.       i = m;
13.     }
14.  ...
15.  }
```

Figure 4: An example program fragment from SIR suite.

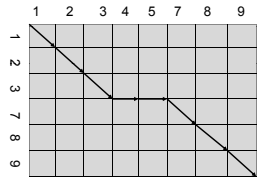

Figure 5: Conceptual view of aligning two execution traces. The traces are taken from the program fragment in Figure 4.

An execution trace exhibiting the above-mentioned observable error will execute $\langle 1, 2, 3, 4, 5, 7, 8, 9\rangle$ in the ith loop iteration. An execution trace not exhibiting the error will execute $\langle 1, 2, 3, 7, 8, 9\rangle$ in the ith loop iteration. Now, let us consider the alignment of these two execution traces — for simplicity we only show the alignment of their ith loop iterations.

We compute the smallest edit distance between the two traces — the minimum cost edits with which one string can be transformed to another. The edit operations are insert/delete/change of one symbol, and the cost of each of these operations need to be suitably defined. Conceptually this is achieved by constructing a two-dimensional grid. The rows of the grid are symbols in the first execution trace, and the columns of the grid are the symbols in the second execution trace. Finding the best alignment between the traces now involves finding the lowest cost path from the top-left corner of the grid to the bottom right corner of the grid. In each cell of the grid, we have choice of taking a horizontal, vertical or diagonal path. Horizontal path means insertion of a symbol in the first execution trace, vertical path means deletion of symbol from the first execution trace, and a diagonal path means comparing the corresponding symbols in the two traces. If we have to insert/delete a symbol we incur some penalty (say $\alpha > 0$). Moreover, if we compare two symbols of the two traces and record a mismatch we also incur some penalty (say β, where typically $\beta > \alpha$). Thus, $0 < \alpha < \beta$. Of course, if we compare two symbols of the two traces and record a match, zero penalty is incurred. A least-cost alignment then corresponds to finding the path with minimum penalty from the top left corner to bottom right corner of the grid. Figure 5 shows the grid for the two traces $\langle 1, 2, 3, 7, 8, 9\rangle$ and $\langle 1, 2, 3, 4, 5, 7, 8, 9\rangle$ taken from the program in Figure 4.[1] A least-cost alignment found for these two traces (assuming $\alpha = 1, \beta = 2$) is shown in Figure 5 via arrows. This corresponds to the following (expected) alignment.

[1]We are explaining our example by presenting the traces at the level of statements. However, in our implementation they will be captured at the level of instructions.

```
1 2 3 _ _ 7 8 9
1 2 3 4 5 7 8 9
```

Having found the *alignment* between two traces, we simply record the aligned branches in the two traces which have been evaluated differently. The sequence of these branches then constitute the trace "difference". In the preceding example, only the branch `3` will appear in the trace difference.

We have now explained our trace alignment and comparison. The trace alignment can be computed by dynamic programming methods operating on the above-mentioned two-dimensional grid – where for each cell `[i,j]` of the grid we keep track of the lowest cost path from the top left corner (cell `[0,0]`) to cell `[i,j]`. However, a straightforward application of dynamic programming will involve space proportional to the product of the lengths of two traces being compared. In practice, this can lead to huge blow-up. For this reason, we have integrated existing linear-space string alignment methods into our trace comparison. Given two traces of length `m`, `n` respectively, we will never construct the entire two-dimensional grid. Instead, we compute the least cost alignment in a divide-and-conquer fashion by finding least cost paths on smaller sub-grids. The reader is referred to [9] for more details.

3.3 Putting it All Together

Given an input t and two program versions P, 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 $\{\varphi_i \mid 0 \leq i < m\}$

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

We take only one solution (if one exists) for each φ_i. This gives us at most m alternate inputs. These alternate inputs are then validated — we check whether the alternate inputs indeed follow the same path as that of test t in P and a different path as that of t in P'. Thus, we get at most m validated alternate inputs; let this set be $Inputs_{validate}(t, P, P')$. For each $t' \in Inputs_{validate}(t, P, P')$,

- we construct the concrete trace π' of t' in P',
- we align and compare π' with π (the trace of input t in P') to get a sequence of branches capturing their difference $diff(t, t')$, and
- get the first branch $bfirst(t, t')$ from the sequence $diff(t, t')$.

Our bug report is the set of branches

$$\{bfirst(t, t') \mid t' \in Inputs_{validate}(t, P, P')\}$$

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. By aligning and comparing the traces of these inputs in P (the old program) with the trace of the input t in P — we obtain at most k branch sequences. By taking the first branch from these branch sequences, we get a bug report of size at most k.

Finally, if we still obtain a large number of alternate inputs (and hence a large bug report), 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 investigated first, before investigating other branch instructions in the bug report.

4. 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 branch errors — errors in branch conditions. 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.

Assignment errors. Bugs that do not influence the program's control flow, such as errors in assignment statements (say that cause the program to generate wrong output) 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 [21, 20], 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 (i) checks for null and the sign of return values at each function return site and (ii) checks for equality of two program variables of the same type (inserted prior to assignment statements). These predicates provide our DARWIN tool 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.

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 new program version, but is present in the old program version. 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 old program P can be expected to have more paths than the new 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 test 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 aligned and compared to yield a bug report. No extension is needed in our methodology to handle code missing errors.

5. IMPLEMENTATION

We now describe our implementation setup. Our debugging tool is called DARWIN — since it explains bugs introduced by software evolution. The overall architecture of our DARWIN toolkit is summarized in Figure 6.

5.1 Generating Alternate Inputs

Within the DARWIN tool, we use a symbolic execution engine for computing the path condition of a given program execution. Our execution engine is a part of the BitBlaze platform [28], 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.

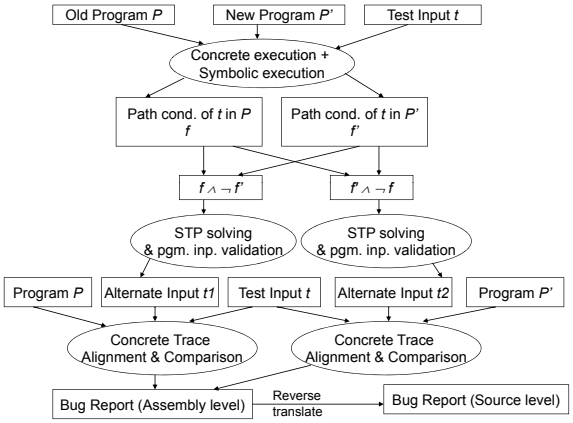

Figure 6: Architecture of our DARWIN toolkit. It takes an old program P, a new program P' and a test input t which passes in P but fails in P'. The output is a bug report explaining the behavior of test t. The entire flow is automated.

The concrete execution is carried out by the TEMU component, which is a whole-system emulator based on QEMU [4]. 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. 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 "relevant" instructions — instructions whose operands are (directly or transitively) dependent on the program input. Note that such input dependence information is present as annotations in the trace recorded by TEMU. Finally, VINE performs a backwards traversal of the trace in the intermediate language. During this backwards traversal, a weakest pre-condition calculation (w.r.t. the formula $true$) is performed thereby producing the path condition.

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 STP solver [15]. STP is an automated satisfiability checker for first-order logic with built-in theories for bit-vectors and arrays. The STP checker serves as a decision procedure for satisfiability of quantifier-free first order logic formula. Indeed this is the case for us, since our formulas do not have universal quantification and any variable is implicitly existentially quantified.

5.2 Constructing Bug Report

Given the solutions of $f \wedge \neg f'$ we validate them, that is, we check whether the concrete traces for these inputs are the same as the trace of the failing input in the old/stable program version P, and different from the failing input's trace in the new/buggy program version P'. We then generate the execution traces of each of these validated test inputs in P'. Each such trace is aligned and compared with the execution trace of the failing test t in P'. This yields a sequence of branches. As mentioned in Section 3.3, we take the first branch from each sequence and put it in our bug report. 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. Again, this can yield many solutions which we then validate. For the validated solutions, we align/compare their traces in the old program version P with the trace of t in P. Each such trace comparison produces a sequence of branch instructions — branches which are aligned in the two traces, but have different outcomes. From each of these branch sequences, we take the first branch and put it in the bug report.

For the trace comparison experiments, we need to set two constants (i) α the cost of inserting/deleting a symbol, and (ii) β, the cost of changing a symbol. Note that we need to choose these parameters in such a way that $0 < \alpha < \beta$. In our experiments, we used $\alpha = 1, \beta = 2$.

By following the steps mentioned in the preceding (solving either $f \wedge \neg f'$ or $f' \wedge \neg f$), we obtain a set of branches at the assembly level as bug report. Using standard compiler level debug information, these can be reverse translated back to the source code level, allowing use of the bug report by the programmer.

Bug Report Size. Finally, we highlight certain low-level issues which make a *substantial* difference to the utility of our bug reports. 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 last section, to solve for $f \wedge \neg f'$ we solve the m formulae $\{\varphi_i \mid 0 \leq i < m\}$ where $\varphi_i \stackrel{def}{=} f \wedge \psi_1 \wedge \ldots \psi_i \wedge \neg\psi_{i+1}$ Further, we take only one solution for each φ_i (if one exists). Our 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 STP solver. Since each φ_i formula contributes at most one alternate input — this reduces the number of alternate inputs. Moreover, the bug report is constructed by taking one branch from the alignment of each alternate input with the buggy input (the first branch where the traces of the two inputs differ). Since the size of the bug report is equal to the number of alternate inputs, we get a smaller sized bug report by reducing the number of alternate inputs. If the bug report size is still high (due to large number of alternate inputs), we can *prioritize* the examination of the branches in the bug report, via a prioritization of the alternate inputs generated by our method. As mentioned in Section 3.3, we investigate the branch instructions contributed by "successful" alternate inputs first, before investigating other branch instructions in the bug report. The successful inputs are those which exhibit same output in both old and new program. Since the observable behavior of successful inputs is bug-free, in our debugging we prefer to investigate the deviating branch instruction contributed by these inputs first.

6. DEBUGGING EXPERIENCE

We report our experience in using DARWIN for locating error causes in real-life case studies.

6.1 Experience with `libPNG`

We first describe our experience in debugging the `libPNG` open source library [2], 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 missed, the buffer may overrun. 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 7, 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 7 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;
}
```

Figure 7: 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.

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 utility of the bug report.

In our debugging method, given the bad PNG image — we *construct* 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.

Employing DARWIN to the debugging task, we 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 nine (9) alternate inputs from the STP solver. These nine alternate inputs are in reality nine PNG images. We align and compare the traces of these nine PNG images with the trace of the bad PNG image in libPNG version 1.2.21 (the fixed version). This gives us nine branch sequences. We take the first instruction from each of these nine branch sequences, thereby getting a bug report containing nine (9) instructions.

We can now prioritize the examination of the bug report as follows. Among the nine alternate inputs contributing to the bug report, we find out which of them are "successful", that is, their observable behavior is as expected by the programmer. In other words, the program output for a successful program input should be the same in both the program versions. Only one of our nine alternate inputs is found to be successful. The branch instruction contributed (to the bug report) by this input corresponds to the branch

```
length > (png_uint_32)png_ptr->num_palette
```

thereby pinpointing the error cause.

Discovering New Errors. Interestingly, in the process of this debugging we found other potential problems in the `libPNG` code. As mentioned earlier, our DARWIN tool obtained nine alternate inputs, only one of which exhibits bug-free behavior, and pointed us to the error. Interestingly, the branch instructions contributed to the bug report by the other eight alternate inputs point us to other deviations between the two versions of `libPNG`. For example, by following one of these eight instructions we find that the two versions of `libPNG` use different functions to retrieve the length field of a chunk from the input. In version 1.0.7, we have

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

while in version 1.2.21 we have

```
length = png_get_uint_31(chunk_length);
```

In particular, the code for `png_get_uint_31` is as follows.

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

Thus, `png_get_uint_31` first uses `png_get_uint_32` and then performs a length check. If `png_get_uint_32` is directly used to find the length of a chunk, a length check w.r.t. the constant `PNG_UINT_31_MAX` is missing. Our bug report contains the branch instruction containing this missing length check, thereby pointing to another potential error in `libPNG`.

6.2 Experience with `miniweb-apache`

In our second case study, we study the web-server `miniweb` [3], 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 or stable program. We chose another HTTP server `apache` [1] as the reference implementation or stable program. The `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 — indeed 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 3.1, we get exactly five alternate inputs:

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

By aligning and comparing the traces of each alternate input with the trace of "`GET x`" in `apache`, we get five branch sequences. As per our methodology, we choose only the first branch instruction (the first place of deviation) from these five branch sequences. This gives us a bug report containing only five instructions. We can immediately localize the bug — `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 five alternate inputs was "successful", exhibiting same output in both program versions. The branch instruction contributed to the bug report by this input pointed us to the missing check for '/'. The other alternate inputs (and the branch instruction contributed by each of them to the bug report) pointed us to other missing checks in `miniweb`. Indeed, we can locate that `apache` contains checks for each of these five characters while `miniweb` misses the check for all five of them — leading to potential errors.

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

6.3 Experience with `savant-apache`

Finally, we discuss our study of `savant` [5], 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 — the latest version of `savant` also exhibits this error. As reference program, we again choose the `apache` webserver, which reports an error for the input query "`GOT /index.html`". Both `savant` and `apache` implement the HTTP protocol, and are expected to behave similarly.

In this case study, DARWIN found forty-six (46) alternate inputs. Out of these only one (1) is successful, that is, produces same output in both `savant` and `apache`. This is the input "`GET /index.html`". Using the branch instruction contributed (to the

Programs	Time in step 1	Time in step 2	Time in step 3
libPNG(v1.0.7-v1.2.21)	16m 40s	9m 15s	6.5s
Miniweb-Apache	11s	2.4s	1.3s
Savant-Apache	17m 43s	5m 11s	55s

Table 1: Performance of DARWIN (m=minutes, s=seconds)

Programs	LOC	Trace size (# instructions)	# Branches in trace
libPNG v1.0.7	31,164	87,336	13,635
libPNG v1.2.21	36,776	108,769	15,472
Miniweb	2,838	270,856	26,201
Savant	8,730	121,714	16,212
Apache	358,379	60,380 (miniweb)	5,388 (miniweb)
		74,002 (savant)	9,672 (savant)

Table 2: Properties of the subject programs

bug report) by 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.

In a Broader Perspective. Our experiments with `savant` also illustrate another additional feature of DARWIN — the ability of 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 rectification was easy and could have been done manually as well. In 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 DARWIN's alternate input generation.

6.4 Performance

Our method involves: (i) constructing and composing the path conditions, (ii) solving formulae, and (iii) comparing traces. Table 1 summarizes the time taken in these steps by DARWIN. In the first step, we construct the path conditions in the two program versions, and then construct several formulae as detailed in Section 3.1. The time taken in this step was less than 18 minutes in all the case studies. In the second step, we solve the various formulae constructed in the first step, thereby producing alternate program inputs. The time taken by the STP solver was less than 10 minutes in all the case studies. In the third step, we align and compare the traces of the alternate inputs with the trace of the buggy input. The time taken by this step was less than 1 minute in all the case studies. The lines of code and trace sizes for the case studies appear in Table 2.

Overall, our DARWIN tool took less than 30 minutes in all the case studies. This time can be significantly reduced by using a more efficient solver than STP — in future we are planning to migrate our tool to the Z3 solver [12]. Moreover, given the times developers usually spend in debugging (hours and days), we believe the time taken by our tool is very reasonable. At the end of the debugging process, our DARWIN tool not only pinpointed the error root cause being investigated, but also found sources of other potential errors.

7. 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 works 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 [10, 24] — 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 [14, 30] — ordering tests in existing test suite to better meet testing objectives of the changed program. Finally, most recently [25] has proposed test-suite augmentation — developing new tests to stress the effect of the program changes. We note that our technique is complementary to regression testing since regression testing seeks to detect or uncover software regressions, whereas we primarily seek to explain (already detected) software regressions.

The issues in comprehending program changes for an evolving code base have been articulated in [27]. Program differencing methods [18, 6, 22] 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 [22] — where the analysis identifies not only the changes, but also which tests are affected by which changes). 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.*, [16], 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 specific 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 [8]). 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 [8] appears to employ techniques similar to ours — the goal of [8] 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 [8] can be fed as an input 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.*, [19]), while others espouse a combination of static and dynamic analysis to find test inputs which expose errors (*e.g.*, [11]). 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.*, [29, 34, 35]), or (b) comparison of the failing program execution with the set of all "correct" executions (*e.g.*, see [7]), or (c) comparison of the failing program execution with one chosen program execution which does not manifest the observable error in question (*e.g.*, [32, 23, 17]). Our work bears some resemblance to works which proceed by comparing the failing program execution with one chosen program execution. The first phase of our approach tries to construct an alternate input with whose trace we compare the failing program execution, and the second phase of our approach involves a trace alignment/comparison. However, the *main novelty* in our approach lies in its ability to consider two different programs in the debugging methodology.

Comparing with delta debugging [33], 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 [31] 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 was 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.

8. CONCLUSION

In this paper, we have presented a debugging methodology and tool for evolving programs. Our DARWIN toolkit takes in two programs and explains the behavior of a test input which passes in the stable program, while failing in the buggy program. Our experience with real-life case studies demonstrate the utility of our method for localizing real bugs. The alternate inputs generated by our method can also help *detect* new errors, apart from localizing a given observable error. We believe that this ability to detect new errors can be a useful feature of our method in practice.

Acknowledgments. This work was partially supported by a Defense Innovative Research Programme (DIRP) grant from Defense Science and Technology Agency (DSTA), Singapore.

9. REFERENCES

[1] Apache webserver. `http://httpd.apache.org/`.
[2] libPNG library. `http://www.libpng.org`.
[3] Miniweb webserver. `http://miniweb.sourceforge.net/`.
[4] QEMU emulator. `http://www.qemu.org`.
[5] Savant webserver. `http://savant.sourceforge.net/info.html`.
[6] T. Apiwattanapong, A. Orso, and M. Harrold. A differencing algorithm for object-oriented programs. In *ASE*, 2004.
[7] T. Ball, M. Naik, and S. Rajamani. From symptom to cause: Localizing errors in counterexample traces. In *POPL*, 2003.
[8] D. Brumley, J. Caballero, Z. Liang, J. Newsome, and D. Song. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In *USENIX Security Conf.*, 2007.
[9] K.-M. Chao, R. Hardison, and W. Miller. Recent developments in linear space alignment methods: A survey. *Journal of Computational Biology*, 1, 1994.
[10] Y. Chen, D. Rosenblum, and K. Vo. Testtube: a system for selective regression testing. In *ICSE*, 1994.
[11] C. Csallner and Y. Smaragdakis. DSD-Crasher: a hybrid analysis tool for bug finding. In *ISSTA*, 2006.
[12] L. de Moura and N. Bjorner. Z3: An efficient SMT solver. In *TACAS*, 2008.
[13] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 2005. `http://www.cse.unl.edu/~galileo/sir`.
[14] S. Elbaum, A. Malishevsky, and G. Rothermel. Prioritizing test cases for regression testing. In *ISSTA*, 2000.
[15] V. Ganesh and D. Dill. A decision procedure for bit-vectors and arrays. In *CAV*, 2007.
[16] O. Giroux and M. Robillard. Detecting increases in feature coupling using regression tests. In *FSE*, 2006.
[17] L. Guo, A. Roychoudhury, and T. Wang. Accurately choosing execution runs for software fault localization. In *CC*, 2006.
[18] S. Horowitz. Identifying the semantic and textual differences between two versions of a program. In *PLDI*, 1990.
[19] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *OOPSLA Onward!*, 2004.
[20] B. Liblit. *Cooperative Bug Isolation*. PhD thesis, UC Berkeley, 2005.
[21] B. Liblit, M. Naik, A. Zheng, A. Aiken, and M. Jordan. Scalable statistical bug isolation. In *PLDI*, 2005.
[22] X. Ren, F. Shah, F. Tip, B. Ryder, and O. Chesley. Chianti: a tool for change impact analysis of java programs. In *OOPSLA*, 2004.
[23] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *ASE*, 2003.
[24] G. Rothermel and M. J. Harrold. A safe efficient regression test selection technique. *TOSEM*, 6, 1997.
[25] R. Santelices, P. Chittimalli, T. Apiwattanapong, A. Orso, and M. Harrold. Test-suite augmentation for evolving software. In *ASE*, 2008.
[26] R. Seacord, D. Plakosh, and G. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices*. Addison-Wesley, 2003.
[27] J. Sillito, G. Murphy, and K. De Volder. Questions programmers ask during software evolution tasks. In *FSE*, 2006.
[28] D. Song et al. Bitblaze: A new approach to computer security via binary analysis. In *ICISS (Keynote Invited Paper)*, 2008. `http://bitblaze.cs.berkeley.edu`.
[29] M. Sridharan, S. Fink, and R. Bodik. Thin slicing. In *PLDI*, 2007.
[30] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in a development environment. In *ISSTA*, 2002.
[31] A. Zeller. Yesterday, my program worked. Today, it does not. Why? In *ESEC/FSE*, 1999.
[32] A. Zeller. Isolating cause-effect chains from computer programs. In *FSE*, 2002.
[33] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28:2002, 2002.
[34] X. Zhang, N. Gupta, and R. Gupta. Pruning dynamic slices with confidence. In *PLDI*, 2006.
[35] X. Zhang, S. Tallam, N. Gupta, and R. Gupta. Towards locating execution omission errors. In *PLDI*, 2007.