Automatic Rootcausing for Program Equivalence Failures in Binaries

Shuvendu K. Lahiri[1], Rohit Sinha[2], and Chris Hawblitzel[1]

[1] Microsoft Research, Redmond, WA, USA
{shuvendu,chrishaw}@microsoft.com
[2] University of California, Berkeley, CA, USA
rsinha@berkeley.edu

Abstract. Equivalence checking of imperative programs has several applications including compiler validation and cross-version verification. Debugging equivalence failures can be tedious for large examples, especially for binary programs. In this paper, we propose a simple yet precise notion of *rootcause* for equivalence failures that leverages semantic similarity between two programs. Unlike existing works on program repair, our definition of rootcause avoids the need for a template of fixes or the need for a complete repair to ensure equivalence. We show progressively weaker checks for detecting rootcauses that can be applicable even when multiple fixes are required to make the two programs equivalent. We provide optimizations based on Maximum Satisfiability (MAXSAT) and binary search to prune the search space of such rootcauses. We have implemented the techniques and provide an evaluation on a set of real-world compiler validation binary benchmarks.

1 Introduction

Equivalence checking between two imperative programs has several applications in software validation. It has been used widely in the translation validation of compilers [19, 13, 10], regression verification [6], cross-version verification [11, 7] and checking independent implementations [17, 21]. Applications such as compiler validation [7], or automatic comparison of student attempts to reference implementations [21], can result in thousands of equivalence checking failures. Automated debugging and identification of the rootcause of a verification failure is crucial for the usability of these verification tools.

The problem of rootcausing is more involved while analyzing assembly or binary programs. Such problems come up naturally in various compiler validation tasks, such as comparing (i) intermediate representations with binaries, or (ii) binaries with different optimizations, or (iii) binaries generated for two different platforms (e.g. x86 vs. ARM), or even (iv) binaries from different versions of a compiler [7]. Comparing binaries (instead of source code or intermediate representations) allows discovering low-level bugs that are introduced during compilation and linking. However, debugging verification failures is tedious due to the lack of (type-based) non-aliasing; most instructions read or modify the registers, flags or the heap.

In a prior work on such binary comparisons, the number of failures even with a modest 2% failure rate, ran into thousands [7]. Equivalence failures resulted from diverse sources such as modeling imprecision, missing environmental specifications, and presence of true defects. Moreover, for many such applications the syntax of the two programs (e.g. x86 vs.

ARM) differ too much to benefit from syntactic difference based tools. To cope with such large-scale applications of equivalence checking for binaries, there is a growing need for automated techniques for understanding and bucketing failures.

In this paper, we provide a *simple* yet *precise* notion of rootcause for equivalence failures of two similar programs. Our work is inspired by work on program repair [18, 9, 14], however our technique attempts to leverage semantic similarity of the two programs. In a nutshell, we attempt to "fix" an equivalence failure by changing an assignment r in one program with a value computed by an assignment l in the other program. The pair of assignments (l, r) from the two programs constitute a rootcause (whenever it exists) by pointing to the two assignments where the two programs diverge.

The simple formulation has two-fold advantage when applicable: (a) it provides correspondence points in two programs that is useful in the setting of comparing two programs, (b) it is more automatic as it does not require any template of fix that is customary in program repair (e.g., [9, 14]), and (c) the rootcause can be found without the need to repair the program, which we find too stringent in the presence of multiple repairs.

On the other hand, the notion of rootcause may appear overly restrictive and seldom applicable in practice. In this work, we show several instances where the semantic structural similarity of the two programs can be exploited to find rootcauses using this notion. First, we formulate two progressively weaker checks (with progressively weaker guarantees) that work by looking for singleton fixes for more constrained equivalence problems. (a) The first check attempts to only fix a single counterexample path of failure in one of the two programs, thereby avoiding the need to make the entire program equivalent. (b) The second check leverages the presence of intermediate synchronization points such as procedure calls to look for fixing the earliest synchronization point where the two programs diverge. Second, we show examples of simple preprocessing to increase the applicability of the existence of such rootcauses.

Although it is easy to symbolically encode the search for such a rootcause in existing program synthesis tools [22, 9, 14], our initial attempt did not scale for the binary benchmarks studied in this paper. We therefore provide a more enumerative solution to search for a rootcause. We provide optimizations based on Maximum Satisfiability (MAXSAT) and binary search to prune the space of candidate fixes. We have implemented the techniques and provide preliminary evaluation on a set of real-world compiler validation benchmarks [7]. On a set of 46 such examples used in the earlier study, our technique finds verified rootcauses for 34 (i.e. 74%) of the cases.

1.1 Overview

We illustrate the concepts informally with the aid of two highly simplified examples of assembly programs generated from a common C# procedure using different compiler versions [7]. For most of these examples, the heap is modeled as an array variable M and the effect of a procedure call is modeled by applying uninterpreted functions such as `f` (line 7 in Figure 1) to the arguments. We refer the reader to earlier works for further details of translating assembly instructions into the language (§ 2.1) used in this paper [7].

Single fix Figure 1 describes two procedures p_1 and p_2 that differ in an extra load from memory M in p_2. The underlined lines are additional instrumentation inserted by our tool,

```
1  procedure p1(M:[int]int, x:int)
2  returns (r1:int, M1:[int]int)
3  {
4    M1 := M;
5    r5 := M1[x];             //ld r5, [x]
6    assume(r5@5 == r5);
7    M1, r1 := f(M1,r5); //call f
8    r1 := r5;           //mov r1, r5
9    M1, r1 := g(M1,r1); //call g
10   return;
11 }
```

```
1  procedure p2(M:[int]int, x:int)
2  returns (r2:int, M2:[int]int)
3  {
4    M2 := M;
5    r6 := M2[x];       //ld r6, [x]
6    M2, r2 := f(M2,r6); //call f
7    r2 := M2[x];             //ld r2, [x]
8    r2 := r5@5;
9    M2, r2 := g(M2,r2); //call g
10   return;
11 }
```

Fig. 1. Optimizing loads. The rootcause pair is highlighted and underlined lines are part of program instrumentation.

and not part of the original programs. Procedure p_2 has two loads (lines 5 and 7) from `x`; the two loads can yield different results if the procedure call `f` can modify M at the location `x`. The compiler for p_1 optimizes the second load in p_1 based on the knowledge that the call to `f` does not modify M at the location `x`. Such internal assumptions from the compiler are often not readily available to the equivalence checker, thereby resulting in equivalence failure. For this example, our tool first inserts the underlined lines in the two procedures. The `assume` in line 6 in p_1 uses a symbolic constant `r5@5` to capture the value computed in `r5` after the load in line 5. The assignment in line 8 in p_2 overwrites the assignment in line 7 with the value in `r5@5`, thereby making the two procedures equivalent. We highlight the rootcause as the pair of original instructions $(5, 7)$ that participate in the fix. Note that we do not actually "repair" the program p_2, since it contains values (e.g. `r5@5`) that are only computed by p_1.

Weaker fixes Figure 2 illustrates a case where our technique can identify a fix even though multiple fixes are required to make the two procedures equivalent. The source of difference between the two procedures is that fields of a class are laid out at different offsets by the two compilers. The field accesses are reflected in the accesses to the heap M using different offsets from a base location `x` (e.g. lines 7, 11, 14 in p_1). Note that the fields in p_2 have an additional offset of 4 compared to fields in p_1. This example is challenging since at least 3 fixes are required to make the two procedures equivalent. Searching for multiple fixes is significantly more expensive and a complete repair may be elusive when two programs have several differences. However, for the purpose of rootcausing, we have observed that it suffices to highlight the *earliest* instruction pair where the two programs diverge. By exploiting the semantic similarity, we can often pose weaker equivalence checks (with weaker global guarantees about the entire program) that may still be verified with a single fix.

Let us assume that the equivalence failure provides a counterexample cex that takes the "then" branch of the conditional in p_1 and the "else" branch in p_2.[1] We present two separate ideas to create a weaker equivalence problem. (i) First, we constrain p_1 to take only that path which was taken by the counterexample; this is achieved by instrumenting assumptions denoting branch conditions satisfied in the counterexample (e.g. line 5 in p_1). (ii) Second, we exploit the fact that semantically similar programs often have intermediate program points where the two programs are expected to *synchronize* — i.e. certain part of the states are expected to be equal. In the presence of such synchronization points, we can

[1] Notice that the branches are rearranged to mimic common compiler transformations.

```
1  procedure p1(M:[int]int, x:int)
2  returns (r1:int, M1:[int]int)
3  {
4    M1 := M;
5   assume(M1[x] == 42); //Fix path
6   if (M1[x] == 42) {
7        r1 := M1[x + 4]; assume(r1@7 == r1);
8        M1 := f(M1,r1); assume(M1@8 == M1);
9    }
10   else {
11       r1 := M1[x+16];
12       M1 := g(M1,r1);
13   }
14   r1 := M1[x+32];
15   ...
16 }
```

```
1  procedure p2(M:[int]int, x:int)
2  returns (r2:int, M2:[int]int)
3  {
4   M2 := M;
5   if (M2[x] != 42) {
6       r2 := M2[x+20];
7       M2 := g(M2,r2);
8   }
9   else {
10      r2 := M2[x + 8];
11      r2 := r1@7;
12      M2 := f(M2,r2);
13      assume (M1@8 != M2); //Early assume
14  }
15  r2 := M2[x+36];
16  ...
17 }
```

Fig. 2. Partial fix. The rootcause pair is highlighted and underlined lines are part of program instrumentation.

look for fixing the violations of such intermediate equalities in addition to the final equivalence. For compiler validation, it is often assumed that the two procedures synchronize on procedure calls — the sequence of procedure calls and the values returned from them are equal on both sides on a common input [13, 10, 7, 11]. One reason for this assumption is that the heap is passed as a map in and out of procedure calls in these settings (we discuss exceptions in Figure 4). Let us assume that the counterexample cex assigns different outputs for M1 and M2 at lines 8 and 12 respectively. We add the underlined `assume` after the update in line 12 in p_2 to make the two maps disequal, assuming procedures synchronize on calls to f. Intuitively, the assumption weakens the final equivalence assertion with an equality over intermediate state of the two programs. The singleton fix $(7, 10)$ does not satisfy this assumption and therefore blocks execution of the instrumented program after line 13 in p_2, which leads to the equivalence check to succeed.

Contributions. The contributions of the paper include (a) the first precise formulation of rootcause for the problem of equivalence failure that does not require a template of fixes or the need to repair a program, (b) mechanisms to improve the applicability of the rootcause by postulating weaker checks by leveraging similarity of the two programs, and (c) an implementation and evaluation on a set of challenging real-world binary equivalence failures.

Organization. We describe a simple programming language used to model binary programs in Section 2. We formalize our notion of rootcause for equivalence failures in Section 3 along with the weaker checks. We describe an algorithm to search for rootcauses in Section 4 along with various optimizations. We present our evaluation in Section 5, and discuss related work in Section 6.

2 Background

2.1 Programs

Figure 3 describes the syntax of programs. *Vars* denotes the set of variables and includes parameters and locals. We simplify exposition by assuming the programs contain no globals. We distinguish scalar variables (denoted by x) from array or map variables (denoted by X). *Consts* denotes a set of symbolic constants. *Relations* and *Functions* denote the set of relations and functions that appear in the program. Relations and functions can either be uninterpreted or be interpreted by an underlying theory such as arithmetic (e.g. $\{\leq, \geq\} \in \mathit{Relations}$ and $\{+, -, *\} \in \mathit{Functions}$). Map operations $\mathsf{x} := \mathsf{X}[\mathsf{y}]$ and $\mathsf{X}[\mathsf{y}] := \mathsf{x}$ are modeled as $\mathsf{x} := \mathit{sel}(\mathsf{X}, \mathsf{y})$ and $\mathsf{X} := \mathit{upd}(\mathsf{X}, \mathsf{y}, \mathsf{x})$ respectively, where *sel* and *upd* are functions in *Functions* interpreted by the theory of arrays [20]. Maps can also be updated at an *unbounded* number of locations by functions returning map values (e.g. $\mathsf{X} := f(\mathsf{X}, \ldots)$).

Statements in *Stmt* include skips (skip), assertions (`assert`), assumptions (`assume`), assignments, sequential composition ($s; s$) and choice statements ($s \diamond s$). Parallel assignments (e.g. line 7 in Figure 1) are desugared as assignments using additional temporary variables. A choice statement $s \diamond t$ non-deterministically executes either s or t. In this paper, choice statements are solely used to model conditional statements where a conditional if (ϕ) $\{s\}$ else $\{t\}$ is modeled as $\{\texttt{assume}\ (\phi); s\} \diamond \{\texttt{assume}\ (\neg\phi); t\}$.

$$
\begin{array}{lll}
\mathsf{x}, \mathsf{X} & \in \mathit{Vars} & \\
\theta & \in \mathit{Consts} & \\
q & \in \mathit{Relations} & \\
f, g, h & \in \mathit{Functions} & \\
e & \in \mathit{Expr} & ::= \mathsf{x} \mid \mathsf{X} \mid \theta \mid f(e, \ldots, e) \\
\phi & \in \mathit{Formula} & ::= \mathsf{true} \mid \mathsf{false} \mid e == e \mid \\
 & & \quad\; q(e, \ldots, e) \mid \phi \wedge \phi \mid \neg\phi \\
s & \in \mathit{Stmt} & ::= \mathsf{skip} \mid \mathsf{x} := e \mid s; s \mid s \diamond s \mid \\
 & & \quad\; \texttt{assert}\ \phi \mid \texttt{assume}\ \phi \\
p & \in \mathit{Proc} & ::= \ g(\mathsf{x}_g, \ldots) : (\mathsf{r}_g, \ldots)\ \{\ s_g\ \}
\end{array}
$$

Fig. 3. Syntax of programs.

A procedure p consists of a list of parameters and return variables, and a body ($s_g \in \mathit{Stmt}$). Procedures are side-effect free, and all the modifications are reflected explicitly by the return variables. As is standard in most prior works on compiler equivalence checking [13, 10, 6, 11], a procedure call is either inlined or the effect of a call is modeled by assigning the return variables an uninterpreted function over the parameters (e.g. line 7 in Figure 1). The treatment in this paper ignores loops; we assume they are either unrolled to a bounded depth or modeled as tail-recursive procedures [12].

A state σ of a program at a given program location is a valuation of the variables in scope. Let Σ be the set of all program states. We omit the definition of an execution as it is quite standard for the statements [1]. We recall that the semantics of `assume` ϕ is to block execution when executed in a state σ that does not satisfy ϕ. For a procedure p, an *input state* is a valuation of the parameters at entry and an *output state* is a valuation of the returns at exit. The semantics of a procedure p is given by a relation $\mathcal{R}(p) \subseteq \Sigma \times \Sigma$ over pairs of input and output states, where $(\sigma, \sigma') \in \mathcal{R}(p)$ if and only if there is an execution of p starting at σ and ending in σ'.

2.2 Equivalence checking

Given two procedures p_1 and p_2 and a one-one mapping of the parameters $\overrightarrow{x}$ and returns $\overrightarrow{r}$, we define p_1 and p_2 to be *partially equivalent* if for every $(\sigma, \sigma') \in \mathcal{R}(p_1)$ and $(\sigma, \sigma'') \in \mathcal{R}(p_2)$, $\sigma' = \sigma''$. In other words, if both p_1 and p_2 do not block (due to assumes) on an input σ, then the outputs are equivalent. We will drop the term partial henceforth when referring to partial equivalence. We check equivalence of two such procedures p_1 and p_2 by creating the composed procedure p_{12} (where p_1 and p_2 are inlined) and checking the final assertion:

$$p_{12}(\overrightarrow{x})\{\overrightarrow{r_1} := p_1(\overrightarrow{x}); \overrightarrow{r_2} := p_2(\overrightarrow{x}); \texttt{assert}\ \overrightarrow{r_1} == \overrightarrow{r_2};\}$$

Since we have assumed p_1 and p_2 are loop-free, p_{12} is a bounded program. Several well-known techniques [1] exist to transform a loop-free and call-free procedure with assertions into a compact logical formula in the Satisfiability Modulo Theories (SMT) format by a process called verification-condition (VC) generation. For our purpose, we define $VC(p)$ to be a logical formula that is *valid* if and only if p does not fail any assertion. If $VC(p_{12})$ is valid then p_1 and p_2 are equivalent; otherwise we obtain a counterexample cex along paths in p_1 and p_2 for which at least one of the return variables differ.

3 Problem formulation

When p_1 and p_2 are not equivalent, the counterexample cex allows the user to debug the equivalence failure. However, such counterexamples can often be hundreds of lines long and finding the relevant instructions that lead to the failure can be cumbersome. Understanding counterexamples for equivalence failures is often laborious due to several factors: (a) most statements in a program are relevant to an equivalence assertion, and (b) one has to proceed simultaneously along both p_1 and p_2. In our prior experience of debugging equivalence failures from compiler validation, summarizing the “core reason” (or rootcause) for equivalence failure was the main ask for the adoption of equivalence checking tools in a production setting [7].

In this section, we formulate a natural notion of rootcause for equivalence failure that exploits the structure of both programs. We pose the rootcause problem as the problem of finding a pair of scalar assignments $l : \texttt{x}_1 := e_1$ (from p_1) and $r : \texttt{x}_2 := e_2$ (from p_2) at labels l and r respectively, such that replacing e_2 in p_2 with the value of e_1 computed in p_1 makes the two procedures equivalent. Observe that the proposal is different from replacing the expression e_2 with e_1 in p_2; such a change may not even yield a well-typed program as the expression e_1 may contain local variables not in scope in p_2. Thus, the rootcause does not really repair the program p_2, but rather yields (when the pair exists) a pair of program points where the two procedures should have been equivalent. We term such a pair (l, r) as a *fix*.

A reader may be concerned about *trivial fixes* in the form of setting the outputs of p_2 to the outputs of p_1. In our experience this seldom happens due to the following reasons: (a) Binary programs contain arrays to model the heap. Most equivalence failures result in the output maps being different at a large (even unbounded) number of locations. Since we do not consider updates to maps for potential fixes, a trivial fix does not work in such cases. (b) A similar argument holds when a procedure has multiple outputs that differ. In addition, for cases when multiple such fixes exist, we always pick the fix that appears earliest in the lexicographic ordering of the pair of labels.

Another concern would be the adequacy of the space of our fixes. This concern is indeed justified due to either (i) several paths may require a fix, or (ii) a long counterexample requires several fixes to align the outputs. We leverage the semantic similarity between the two programs to formulate two progressively weaker checks (with progressively weaker guarantees) that work by looking for singleton fixes for more constrained equivalence problems. Let us refer to the first (original) check that checks to fix p_{12} with the pair (l, r) as AllFix check. The second check ($\mathit{LeftPathFix}$) attempts to only fix a single counterexample path of failure, thereby avoiding the need to make the entire program equivalent. The third check ($\mathit{LeftPathEarliestFix}$) leverages the presence of intermediate synchronization points such as procedure calls to look for fixing the earliest synchronization point where the two programs diverge.

In the next few sections, we formalize the different notions of rootcauses with the aid of a program instrumentation.

3.1 Instrumentation

For a pair of procedures p_1 and p_2, let L and R be the sequence of labels in the left (respectively p_1) and right (respectively p_2) procedures. Each label l corresponds to a scalar assignment $\mathtt{x}_l := \mathtt{e}_l$. We sometimes treat L and R as sets instead of a sequence. We define an instrumentation that transforms a statement to another statement:

- For each scalar assignment instruction $l : \mathtt{x} := \mathtt{e}$ with a label $l \in L$, we transform it to:

$$l : \mathtt{x} := \mathtt{e}; \mathtt{assume}(\theta@l == \mathtt{x})$$

where $\theta@l$ is a *fresh* constant for storing the value of $\mathtt{x}$ after the assignment at label l.
- For each assignment instruction $r : \mathtt{x} := \mathtt{e}$ with a label $r \in R$, we transform it to:

$$r : \mathtt{x} := \mathtt{e};\ \mathtt{x} := \gamma_r ?\ \theta@r :\ \mathtt{x};\ \mathtt{assume} \bigwedge_{l \in L} (\beta_r^l \Rightarrow \mathtt{x} == \theta@l)$$

Here $\theta@r$ and γ_r are fresh constants for label r. Setting γ_r to true replaces the current assignment at r with a completely unconstrained value $\theta@r$ in p_2. For each $l \in L$, we also create a Boolean constant β_r^l to denote a candidate fix (l, r). The constant β_r^l constrains $\mathtt{x}$ to be equal to the value assigned at label $l \in L$. It is easy to see that setting γ_r to true and exactly one of β_r^l to true (and other candidates $\beta_r^{l'}$ to false) is equivalent to an assignment $\mathtt{x} := \theta@l$, which is the intended fix.

For all further discussions, we refer p_{12} to mean the instrumented version of p_{12}. We next describe the meaning of two operations $\mathit{ConstrainFix}$ and $\mathit{AssignFix}$, that strengthen the formula being checked by the SMT solver:

- $\mathit{ConstrainFix}(p_{12}, L', R')$ takes two sets of labels $L' \subseteq L$ and $R' \subseteq R$ and constrains all the candidates in $L' \times R'$ to true. It generates the following logical formula:

$$(\bigwedge_{r \in R} \neg\gamma_r \wedge \bigwedge_{(l,r) \in L' \times R'} \beta_r^l \wedge \bigwedge_{(l,r) \in (L \times R) \setminus (L' \times R')} \neg\beta_r^l) \Rightarrow VC(p_{12})$$

- $\mathit{AssignFix}(p_{12}, (l', r'))$ takes a fix (l', r') and overwrites the assignment at r' with value computed at l'. It generates the following logical formula:

$$(\gamma_{r'} \wedge \beta_{r'}^{l'} \wedge \bigwedge_{r \in R \setminus \{r'\}} \neg\gamma_r \wedge \bigwedge_{(l,r) \in L \times R \setminus \{(l',r')\}} \neg\beta_r^l) \Rightarrow VC(p_{12})$$

The encodings give rise to a few simple facts.

Lemma 1. *For $L_1 \subseteq L_2 \subseteq L$ and $R_1 \subseteq R_2 \subseteq R$, if $\mathit{ConstrainFix}(p_{12}, L_1, R_1)$ is valid, then $\mathit{ConstrainFix}(p_{12}, L_2, R_2)$ is valid.*

Lemma 1 follows from the fact that setting more β_r^l constants to true adds more assumes to p_{12}, thus making the specification weaker.

Lemma 2. *For $l \in L$ and $r \in R$, if the formula $\mathit{AssignFix}(p_{12}, (l, r))$ is valid, then the formula $\mathit{ConstrainFix}(p_{12}, \{l\}, \{r\})$ is valid.*

Lemma 2 follows from the observation that replacing an unconstrained constant $\theta@r$ with a more constrained expression e in the assignment r : `x := e` can never change a valid formula into an invalid formula.

Theorem 1. *For $L_1 \subseteq L$ and $R_1 \subseteq R$, if $\mathit{ConstrainFix}(p_{12}, L_1, R_1)$ is not valid, then $\mathit{AssignFix}(p_{12}, (l, r))$ is not valid for any $(l, r) \in L_1 \times R_1$.*

The theorem follows immediately from the two lemmas. The utility of the theorem is in providing a sufficient condition to prune a subset of candidate fixes, without explicitly trying each of them. We use this for optimizations in section 4 and 4.

3.2 Different checks

We now formalize the different checks starting with the strongest check.

Definition 1 **(AllFix).** *$\mathit{AllFix}(p_1, p_2)$ is true if there exists a fix $(l, r) \in L \times R$ such that $\mathit{AssignFix}(p_{12}, (l, r))$ is valid.*

For the example in Figure 1, both $(5, 7)$ and $(8, 7)$ constitute a fix according to the AllFix check. We highlight the pair $(5, 7)$ since it is lexicographically smaller than $(8, 7)$.

If AllFix does not hold, then we can try a weaker check. Given a counterexample path cex, we define $\mathit{HoldLeftPath}(p_{12}, \mathit{cex})$ as constraining p_1 to only take the path taken in cex. Figure 2 shows an example (line 5 in p_1) where the branch condition of the taken branch is assumed before the conditional statement.

Definition 2 **($\mathit{LeftPathFix}$).** *$\mathit{LeftPathFix}(p_1, p_2, \mathit{cex})$ is true if there exists a fix $(l, r) \in L \times R$ such that $\mathit{AssignFix}(\mathit{HoldLeftPath}(p_{12}, \mathit{cex}), (l, r))$ is valid.*

Observe that the check $\mathit{LeftPathFix}$ does not yield a fix for the example in Figure 2. This is because even this single path requires at least two fixes to constants in lines 10 and 15.

We can further weaken the final assertion by exploiting statically defined intermediate synchronization points for the two procedures, where certain variables are expected to match up on the two sides. For example, for compiler translation validation, it is common to assume that the sequence of procedure calls and the values returned from them are equal on both programs on a common input. In the presence of such synchronization points, we can look for fixing the violations of such intermediate equalities in cex in addition to the final equivalence.

For a counterexample cex, let $l_1, \ldots, l_m$ and $r_1, \ldots, r_n$ be the sequence of assignment labels from p_1 and p_2 respectively. Further, let a subset of instruction pairs $(l^1, r^1), \ldots, (l^j, r^j)$ (ordered by cex) are expected to be the synchronization points. We find the *earliest* pair where the synchronization is violated by asserting the equality between each pair and finding the first pair where the assertion does not hold; for these checks, we disable the final assertion and restrict the paths in p_1 and p_2 to follow cex. Let (l^k, r^k) be the earliest violation of synchronization (it may not always exist) and let x be the variable assigned in r^k. We insert the following assume statement after the assignment at r^k:

$$r^k : \mathsf{x} := e;\ \texttt{assume}(\mathsf{x} \neq \theta @ l^k)$$

Intuitively, the assume weakens the final equivalence check by pruning behaviors that satisfy the synchronization at (l^k, r^k). In turn, this expects less from a fix: a fix does not need to check the final equivalence if it can synchronize (l^k, r^k). For compiler validation, it is often assumed that p_1 and p_2 synchronize on procedure calls, for which x would represent the heap that is passed out of the procedure calls. We define the instrumentation $\mathit{AddEarlyDiseqAssumes}(p_{12}, cex)$ to insert such assumes into p_{12}. Figure 2 shows an instance of such assume in line 13 in p_2 for map variable M.

These assumptions are most useful when the counterexample path is constrained to cex. Otherwise, the verifier can find an alternate path and avoid the inserted assume statement. Hence we use this in conjunction with the instrumentation of $\mathit{HoldLeftPath}(p_{12}, cex)$.

Definition 3 **(**$\mathit{LeftPathEarliestFix}$**).** *$\mathit{LeftPathEarliestFix}(p_1, p_2, cex)$ is true if there exists a fix $(l, r) \in L \times R$ such that the logical formula represented by $\mathit{AssignFix}(\mathit{HoldLeftPath}(\mathit{AddEarlyDiseqAssumes}(p_{12}, cex), cex), (l, r))$ is valid.*

The example in Figure 2 satisfies this weaker check and yields the rootcause pair highlighted in the figure.

It is worth pointing the difference with an alternate option of inserting an assertion $\texttt{assert}(\mathsf{x} == \theta @ l^k)$ at r^k and removing the final assertion. Such a check can be verified with spurious fixes that avoid the path leading to the assertion, and we have found it to be true in practice. Finally, the following theorem formalizes the relationship between the three checks:

Theorem 2. *Given procedures p_1 and p_2, and a counterexample cex to $VC(p_{12})$, $\mathit{AllFix}(p_1, p_2, cex) \Rightarrow \mathit{LeftPathFix}(p_1, p_2, cex) \Rightarrow \mathit{LeftPathEarliestFix}(p_1, p_2, cex)$.*

In summary, there are several advantages to our natural formulation of rootcause:

- We can exploit the semantic similarity of the two closely related programs by moving "values" computed in p_1 into p_2 for the fix. Our notion of rootcause is precise, but does not require a complete repair to the program.
- The formulation does not require separate templates for repairing a program [14, 21]. This is useful when the repair templates may not be obvious (e.g. the repair of p_2 in Figure 1 requires strengthening the environment assumptions of callees).
- When such a fix exists, it points to correspondence points in the two programs that differ under cex but are *necessary* for equivalence. We have found this to be much more informative than fixing one statement in p_2, as would be done by existing rootcause methods [21].
- The semantic similarity between the two programs (as opposed to a specification versus a program) can be exploited to formulate weaker checks that can yield a rootcause.

Algorithm 1 $FindRootCause(p_{12}, cex)$

```
Input: Combined procedure $p_{12}$
Input: A counterexample $cex$ to equivalence failure of $p_{12}$
Output: $\{NOROOTCAUSE, ROOTCAUSE(l, r)\}$
 1: $(L, R) \leftarrow$ Scalar assignment labels in $cex$
 2: $PruneCandidatesStatic(cex, L, R)$
 3: if $CheckSAT(ConstrainFix(p_{12}, L, R)) \neq UNSAT$ then
 4:     return $NOROOTCAUSE$ /* No fix exists */
 5: end if
 6: /* Binary search based pruning */
 7: $(low, up) \leftarrow (0, |R|)$
 8: while $(up - low > 1)$ do
 9:     $curr \leftarrow low + (up - low)/2$
10:     if $CheckSAT(ConstrainFix(p_{12}, L, [1, curr])) \neq UNSAT$ then
11:         $low \leftarrow curr$
12:     else
13:         $up \leftarrow curr$
14:     end if
15: end while
16: /* MAXSAT based pruning */
17: for $r \in [low + 1, |R|]$ do
18:     $L' \leftarrow L \setminus CheckMAXSAT(ConstrainFix(p_{12}, L, \{r\}), \{\beta_r^l \mid l \in L\})$
19:     for $l \in L'$ in program order do
20:         if $CheckSAT(AssignFix(p_{12}, (l, r))) == UNSAT$ then
21:             return $ROOTCAUSE(l, r)$
22:         end if
23:     end for
24: end for
25: return $NOROOTCAUSE$
```

4 Searching for a fix

Our first attempt was to leverage a counterexample-guided inductive synthesis (CEGIS) based program synthesis engine to symbolically encode the search for a fix [22, 15]. The algorithm searches for an assignment to boolean constants $\beta \cup \gamma$ (at most one fix for each $r \in R$) such that p_1 and p_2 are equivalent. Promisingly, CEGIS can find multiple fixes; in the case of figure 2, it finds all 3 fixes needed to make p_1 and p_2 equivalent. However, we did not succeed in scaling the CEGIS-based algorithm to the compiler validation benchmarks due to timeouts in the theorem prover. There are several reasons why CEGIS does not scale for our benchmarks: (a) the benchmarks contain several hundred lines along with heavy use of quantifiers to model semantics of binary programs, and (b) the size of the instrumented program fed to CEGIS is quadratic in the size of the two input programs. The combination of these two factors make the problem of generating a model or satisfiable input more difficult for SMT solvers.

We now present an alternate algorithm for searching for a fix in Algorithm 1. We assume that p_{12} has already been instrumented with one of the three checks $\{AllFix$, $LeftPathFix$, $LeftPathEarliestFix\}$. It returns $NOROOTCAUSE$ to denote that no singleton rootcause exists, and returns $ROOTCAUSE(l, r)$ for a pair (l, r) that fixes p_{12}. A naïve solution will enumerate every pair of assignments (l, r) over p_1 and p_2 and check for $AssignFix(p_{12}, (l, r))$. This can lead to a best case quadratic (in the size of p_{12}) number of theorem prover checks when no such fix exists. In section, we describe a few techniques to prune the space of candidate fixes where a fix cannot be found.

The search for a singleton fix enables a few simple optimizations. Given a counterexample cex, the first step is to collect into (L, R) only the scalar assignments that appear along cex (line 1). The method $PruneCandidatesStatic$ prunes pairs (l, r) such that $cex(l) = cex(r)$, i.e. the assignments that produce equal value in cex. For any such pair (l, r), applying that fix will not prevent the counter-example cex. Therefore, $PruneCandidatesStatic$ fixes the β^l_r constants to false permanently.

For the remaining pairs, we perform pruning based on calls to an automated theorem prover. The operation $CheckSAT(\phi)$ checks if $\neg\phi$ is satisfiable (SAT) or unsatisfiable ($UNSAT$); these cases correspond to ϕ being invalid and valid respectively. Line 3 checks if constraining p_{12} with all the remaining assumes guarded by β^l_r (except those disabled in line 2 earlier) can verify p_{12}. If the result is SAT, there can be no fix with (L, R) (Theorem 1). The check for a fix is done in line 20; if the formula is valid, then we return the rootcause pair (l, r). Lines 6-15 use *binary search* to prune a subset of candidates. Lines 16-24 use Maximum Satisfiability (MAXSAT) to prune a subset of candidates. We describe these optimizations in the next two sections.

Binary search based pruning We are interested in pruning the sub-range of R where no fix can lie. We observe that for any fix (l, r) (such that $AssignFix(p_{12}, (l, r))$ is valid), the following condition follows from Theorem 1: $r > r'$ for any $r' \in R$ for which $ConstrainFix(p_{12}, L, [1, r'])$ is not valid. We use binary search over $[1, |R|]$ to find the largest r (returned in the variable low) such that $ConstrainFix(p_{12}, L, [1, r])$ is invalid. We use two markers low and up with the following loop invariants: (i) $ConstrainFix(p_{12}$, L, $[1, low])$ is invalid, and (ii) $ConstrainFix(p_{12}$, L, $[1, up])$ is valid, and (iii) $low \leq up$. The binary search converges in at most $\log(|R|)$ steps since the distance between low and up is halved at each step.

MAXSAT based pruning For a fixed $r \in R$, we are also interested in pruning a subset of L where no fixes can lie. We use Maximum Satisfiability (MAXSAT) to perform this. For a given $r \in R$ if $ConstrainFix(p_{12}, L, \{r\})$ is valid, then we find the largest subset $L'' \subseteq L$ such that $ConstrainFix(p_{12}, L'', \{r\})$ is invalid. From Theorem 1 we know that a fix (l, r) cannot be found for any $l \in L''$. Computing the largest (invalid) subset L'' can be performed by the call to $CheckMAXSAT(ConstrainFix(p_{12}, L, \{r\}), \{\beta^l_r \mid l \in L\})$, where the first argument is a formula ϕ and the second argument is a set S of Boolean constants that are "soft". $CheckMAXSAT(\phi, S)$ returns the largest subset $S' \subseteq S$ such that $\neg\phi \wedge \bigwedge_{s \in S'} s$ is satisfiable. For our purpose, the set of soft constants consists of the set of all candidate fixes from L for a given $r \in R$.

As an example, consider this optimization in the context of Fig. 1. Consider the MAXSAT query when considering the statement $\mathtt{r2} := \mathtt{M2}[\mathtt{x}]$. The potential set of candidates from p_1 are (ignoring updates to maps):

$$L \doteq \{5 : \mathtt{r5} := \mathtt{M1}[\mathtt{x}], 7 : \mathtt{r1} := \mathtt{f}(\mathtt{M1}, \mathtt{r5}), 8 : \mathtt{r1} := \mathtt{r5}, 9 : \mathtt{r1} := \mathtt{g}(\mathtt{M1}, \mathtt{r1})\}$$

The instrumentation of $7 : \mathtt{r2} := \mathtt{M2}[\mathtt{x}]$ is as follows:

$$7 : \mathtt{r2} := \mathtt{M2}[\mathtt{x}];\ \mathtt{r2} := \gamma_7 ?\ \theta@7 :\ \mathtt{r2};\ \mathtt{assume} \bigwedge_{l \in [5,7,8,9]} (\beta_7^l \Rightarrow \mathtt{r2} == \theta@l)$$

For this program, both $(5, 7)$ and $(8, 7)$ are valid (singleton) fixes and make the two program equivalent.

Consider the case when γ_7 is false and the subset $\{\beta_7^7, \beta_7^9\}$ are true. The two programs are not equivalent under this constraint. In other words, the verification condition $VC(p_{12})$ is SAT even with these constraints. The call to $CheckMAXSAT$ with $\{\beta_7^l \mid l \in \{5, 7, 8, 9\}\}$ as the soft clauses will return the largest set $\{\beta_7^7, \beta_7^9\}$ that is satisfiable, thereby pruning the set of candidates that have to be explicitly tested by 2.

5 Evaluation

In this section, we describe an implementation of the techniques and an evaluation on a set of binary benchmarks from compiler validation. Our implementation is part of SYMDIFF sources [23], and takes as input a Boogie program p_{12} generated by the equivalence checker tool SYMDIFF [11]. The inputs to SYMDIFF are Boogie programs p_1 and p_2 and a mapping between the two procedures. These Boogie programs can be generated from various languages such as C [4], or from various compiler back-end formats [24, 7]. For this section, we only focus on Boogie programs generated from compiler validation benchmarks [7]. Note that the core algorithm is agnostic of the binary nature of the benchmarks; we do report some *preprocessing* heuristics to control the search in this section.

The main goal of our experiment is to determine how often our notion of rootcause can be found in real benchmarks. The evaluation consists of two parts. In Section 5.1, we evaluate the implementation on 15 *smallest* benchmarks of equivalence failures resulting from comparing the output of the .NET CLR compiler across two different optimization levels. In Section 5.2, we evaluate the implementation on 46 benchmarks comparing the output of the .NET CLR compiler in Just In Time (JIT) mode to the compiler in mostly-ahead-of-time mode (based on the MDIL [16] machine-dependent intermediate language). We restrict to the smallest 15 for the former category to be able to manually establish the ground truth (the true reason for failure) of these failures, which can be quite tedious. For the latter benchmarks, previous syntactic heuristics provided good starting points for establishing the ground truth. Thus we report more quantitative evaluation for the latter category.

5.1 Different optimization levels

We successfully find a total of 12 rootcauses (80% of the cases) out of the 15 benchmarks in this category. These benchmarks have 68 lines of assembly code on average, and the generated Boogie programs have 510 Boogie statements on average. In most cases, we found a fix with either the $LeftPathFix$ and $LeftPathEarliestFix$ checks. However, some of the examples violated the assumption that the two programs have equal set of callees. We illustrate the problem and an additional *preprocessing* step that alleviates the problem without any changes to the algorithm.

```
1  procedure p1(M:[int]int, x:int)
2  returns (r1:int, M1:[int]int)
3  {
4    M1 := M; r1 := M1[x];
5    assume(M1@5 == M1);
6    M1, r1 := getLength(M1, r1);
7    M1 := M1@5;
8    assume(r1@6 == r1);
9    if (r1 > 0) {
10     M1, r1 := writeToFile(M1,r1);
11   }
12   ...
13 }
```

```
1  procedure p2(M:[int]int, x:int)
2  returns (r2:int, M2:[int]int)
3  {
4    M2 := M; r2 := M2[x];
5    r2 := M2[r2 + 8];
6    r2 := r1@6;
7    if (r2 > 0) {
8      M2, r2 := writeToFile(M2,r2);
9    }
10   ...
11 }
```

Fig. 4. Example for side-effect free preprocessing heuristic.

Heuristic		*AllFix*		*LeftPathFix*		*LeftPathEarliestFix*	
	Candidates	R/NR	time(sec)	R/NR	time(sec)	R/NR	time(sec)
$callee$	125	12/27	85.4	14/25	77.8	24/15	85.0
$load$	128	2/37	113.1	3/36	98.6	6/32	129.1
imm	128	7/5	193.8	7/4	191.0	16/0	181.0
$callWind_1$	107	13/9	142.1	14/9	154.7	23/0	212.7
$callWind_2$	272	6/8	264.7	7/8	251.7	14/0	289.8
Total	140	15/24	134.1	18/22	128.4	34/5	154.4

Fig. 5. Summary on benchmarks returning either $ROOTCAUSE$ or $NOROOTCAUSE$.

Figure 4 illustrates a case where our technique fails to identify a fix when a procedure call in p_1 is replaced by access to the object's field in p_2. This is an instance of a common compiler optimization of inlining simple methods (such as side-effect free "getter" methods) with their implementations. Our tool fails to find a rootcause because (i) procedure call to getLength can modify the heap M1 arbitrarily and cause it to differ from M2, and (ii) return value r1 of getLength is allowed to differ from the field access $\mathsf{M2}[\mathsf{r1} + 8]$. To account for (i), we exploit the fact that semantically similar programs often have identical procedure calls — i.e. if a procedure only appears in one program, then it is likely to be side-effect free. In a preprocessing step, we modify p_1 to insert an assume in line 5 and restore M1 in line 7. Then, our rootcause analysis identifies the singleton fix $(6, 5)$. Similarly, we find examples that make several calls to the same procedure in p_1 that are optimized to only one call in p_2 because the compiler is able to prove idempotence. We perform a similar instrumentation as figure 4 to handle this case.

5.2 JIT vs. compiled binaries

Benchmarks These benchmarks have 165 lines of assembly code on average, with the largest benchmark having 574 lines.[2] The generated Boogie programs have 1242 Boogie statements on average, with the largest benchmark having 4323 statements. The considerable sizes make it difficult to apply program synthesis to find the rootcauses. In fact, we describe a few heuristics to (unsoundly) prune the search space to be able to find the rootcauses within 800 seconds.

Heuristics for pruning candidates We define heuristics to match certain syntactic patterns in the assignments being considered. We define the following three heuristics for finding

[2] We are unable to release these specific compiler benchmarks due to confidentiality agreement.

rootcauses of x86 programs. The $callee$ heuristic only considers assignments of the callee before making an indirect call. The heuristic $load$ only considers instructions that load a register from a memory address, and the heuristic imm only considers assignments that contain an arithmetic constant or an arithmetic operation in the expression. In addition, we also use a heuristic that exploits the synchronization of p_1 and p_2 across procedure calls. Let $f_1, \ldots, f_m$ and $g_1, \ldots, g_n$ be the sequence of procedure calls along cex in p_1 and p_2 respectively. We define a pair of calls (f_i, g_i) as the *earliest mismatched calls* if the calls $(f_1, g_1), \ldots, (f_{i-1}, g_{i-1})$ return matching outputs and (f_i, g_i) mismatch. We define a heuristic $callWind_k$ that only considers fixes in the region between $f_{i-k}, \ldots, f_i$ and $g_{i-k}, \ldots, g_i$ in p_1 and p_2 respectively.

Experimental setup Each benchmark consists of two procedures p_1 and p_2 being compared, and a syntactic filter $\in \{callee, load, imm, callWind_1, callWind_2\}$. With each of the 5 syntactic filters, we experiment with 46 pairs of procedures, giving us 230 benchmarks in total. Each benchmark is run with all optimizations from Sec. 4 enabled. We instantiate $LeftPathEarliestFix$ by synchronizing at procedure call boundaries.

Results Figure 5 presents the results. Excluding those benchmarks that return $UNKNOWN$, the table presents the following metrics: 1) average number of candidates generated for each benchmark, 2) benchmarks for which a rootcause is found (R) or not found (NR), and 3) average runtime (in seconds). The benchmarks that return $UNKNOWN$ fail either from out-of-memory exceptions in Boogie/Z3, or from our 800 second timeout. These results indicate that progressively weakening the check to $LeftPathFix$ and $LeftPathEarliestFix$ identifies rootcause in more benchmarks. The "Total" row describes the total number of distinct rootcauses found across the different heuristics. We find a total of 34 out of 46 rootcauses (74% of the cases), which we find quite encouraging. For the remaining examples, the most common reasons for $NOROOTCAUSE$ include: (i) the need for multiple fixes even for the weakest check $LeftPathEarliestFix$, (ii) insufficient semantic similarity between p_1 and p_2, whereby p_1 is devoid of a value that fixes p_2, and (iii) several missing assumptions about the read and write sets of callees and aliasing assumptions.

We also investigate the effect of our optimizations on runtime and the number of candidate fixes. For each optimization, we study the effect of the optimization by disabling it while enabling the remaining optimizations. We measure the impact of an optimization $\in$ {Binary Search, MAXSAT} with respect to (a) the number of candidates pruned, and (b) the reduction in runtime. Figure 6 and Figure 7 (in Section A) show the effect of disabling Binary Search and MAXSAT, respectively. Both optimizations perform a substantial reduction in candidates, more pronounced for larger instances, with MAXSAT (resp., Binary Search) giving us a significant 49% (resp., 12%) reduction in runtime and 691% (resp., 34%) reduction in candidates. Observe that the average runtime improvement is lower than the average improvement in candidates. In fact, there are few cases in Binary Search and MAXSAT where the optimization results in a slowdown because performing the optimization itself requires invoking the solver.

6 Related Work

Automated debugging and repair are certainly not new problems. Our work is inspired in part by program repair techniques [21],[2],[14], and in part by error localization techniques [3]. The novelty of our work is in providing formal guarantees for the rootcause (unlike localization approaches) without requiring to completely repair the program. Unlike our work, none of these approaches deal with the complexities of analyzing binary programs.

Error Localization. BugAssist by Jose et al. [8] analyzes a specific failing input to compute a minimal set of program statements that can be potentially changed to prevent the failing execution. Ermis et al. [5] propose a concept of error invariants to slice error traces using interpolants. In our context, we observe that most instructions in the program are relevant for equivalence failure. Consequently, both techniques end up retaining most of the instructions along the counterexample path. However, there is no guarantee that these rootcauses (program expressions) can be changed to repair the program.

Repair. On the other hand, there is active research in using synthesis for repairing programs. Nguyen et al. [14] assume a single-fix assumption to synthesize a repair such that the program passes all its test cases. Könighofer et al. [9] relax the single-fix assumption and perform template-based repair using a counterexample guided inductive synthesis (CEGIS) loop. Singh et al. [21] use constraint-based synthesis to automatically provide feedback to students in an introductory programming course. They use the instructor's solution only as a specification for synthesizing a set of fixes to the student's solution i.e. their approach extends to the multiple-fix model. The sizes of examples from compiler validation are at least an order bigger than the benchmark sizes for student attempts; Furthermore, the space of all repairs is quite large in our setting (all x86 instructions with all possible operands). Our work differs from all three of [14], [21], and [9] by (i) exploiting similarity in the two programs and therefore not requiring repair templates, and (ii) alleviating scalability issues by not insisting on a complete fix. However, our approach may fail to identify a rootcause when the program requires multiple fixes, or when p_1 does not possess a value that can fix p_2. In other related work, Samanta et al. [18] repair boolean programs with the single-fix assumption using QBF solving. In our setting, we do not abstract assembly language programs as boolean programs.

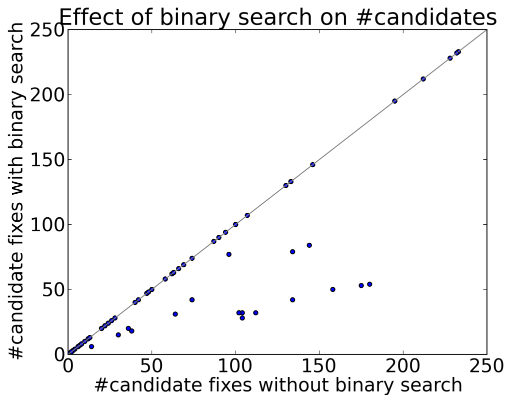
7 Conclusion

In this effort, we have proposed a new formulation of rootcause for equivalence failures of similar binary programs. We have implemented our technique and evaluated it on several real-world binary equivalence failures and report the potential to be useful. We believe the idea is general and can be applied to other equivalence checking domains (e.g. grading assignments). We are currently extending the formulation to handle multiple fixes and combining our search with counterexample-guided synthesis methods.

References

1. M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In *PASTE '05*, pages 82–87, 2005.
2. S. Chandra, E. Torlak, S. Barman, and R. Bodik. Angelic debugging. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 121–130, New York, NY, USA, 2011. ACM.
3. H. Cleve and A. Zeller. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 342–351, New York, NY, USA, 2005. ACM.

4. J. Condit, B. Hackett, S. K. Lahiri, and S. Qadeer. Unifying type checking and property checking for low-level code. In *POPL*, pages 302–314, 2009.
5. E. Ermis, M. Schäf, and T. Wies. Error invariants. In *FM 2012: Formal Methods*, pages 187–201. Springer Berlin Heidelberg, 2012.
6. B. Godlin and O. Strichman. Regression verification. In *DAC*, pages 466–471, 2009.
7. C. Hawblitzel, S. K. Lahiri, K. Pawar, H. Hashmi, S. Gokbulut, L. Fernando, D. Detlefs, and S. Wadsworth. Will you still compile me tomorrow? static cross-version compiler validation. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 191–201, New York, NY, USA, 2013. ACM.
8. M. Jose and R. Majumdar. Cause clue clauses: Error localization using maximum satisfiability. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 437–446, New York, NY, USA, 2011. ACM.
9. R. Konighofer and R. Bloem. Automated error localization and correction for imperative programs. In *Formal Methods in Computer-Aided Design (FMCAD), 2011*, pages 91–100, Oct 2011.
10. S. Kundu, Z. Tatlock, and S. Lerner. Proving optimizations correct using parameterized program equivalence. In *Programming Language Design and Implementation (PLDI '09)*, pages 327–337. ACM, 2009.
11. S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo. Symdiff: A language-agnostic semantic diff tool for imperative programs. In *Computer Aided Verification (CAV'12)*, LNCS 7358, pages 712–717, 2012.
12. S. K. Lahiri, K. L. McMillan, R. Sharma, and C. Hawblitzel. Differential assertion checking. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 345–355, New York, NY, USA, 2013. ACM.
13. G. C. Necula. Translation validation for an optimizing compiler. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '00)*, pages 83–94, 2000.
14. H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 772–781, Piscataway, NJ, USA, 2013. IEEE Press.
15. H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 772–781, Piscataway, NJ, USA, 2013. IEEE Press.
16. S. Ramaswamy. Deep dive into the kernel of .NET on Windows Phone 8. In *Build Conference*, 2012.
17. D. A. Ramos and D. R. Engler. Practical, low-effort equivalence verification of real code. In *Computer Aided Verification (CAV'11)*, LNCS 6806, pages 669–685, 2011.
18. R. Samanta, J. V. Deshmukh, and E. A. Emerson. Automatic generation of local repairs for boolean programs. In *Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design*, FMCAD '08, pages 27:1–27:10, Piscataway, NJ, USA, 2008. IEEE Press.
19. H. Samet. Compiler testing via symbolic interpretation. In *in Proceedings of the ACM 29th Annual Conference*, pages 492–497, 1976.
20. Satisfiability Modulo Theories Library (SMT-LIB). Available at `http://goedel.cs.uiowa.edu/smtlib/`.
21. R. Singh, S. Gulwani, and A. Solar-Lezama. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 15–26, New York, NY, USA, 2013. ACM.
22. A. Solar-Lezama, G. Arnold, L. Tancau, R. Bodik, V. Saraswat, and S. Seshia. Sketching stencils. *SIGPLAN Not.*, 42(6):167–178, June 2007.
23. SymDiff source code. Available at `http://symdiff.codeplex.com`.



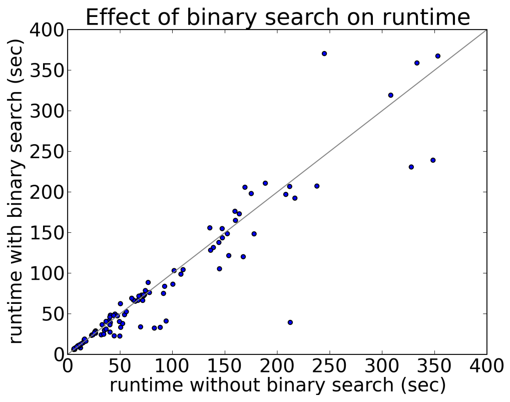


Fig. 6. The improvement in number of candidates is 0% at the minimum, 271% at the maximum, 34% on average (geomean 21%). The runtime improvement is -47% at the minimum, 439% at the maximum, 12% on average (geomean 6%).

24. J. Yang and C. Hawblitzel. Safe to the last instruction: Automated verification of a type-safe operating system. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 99–110, 2010.

A Plots of results

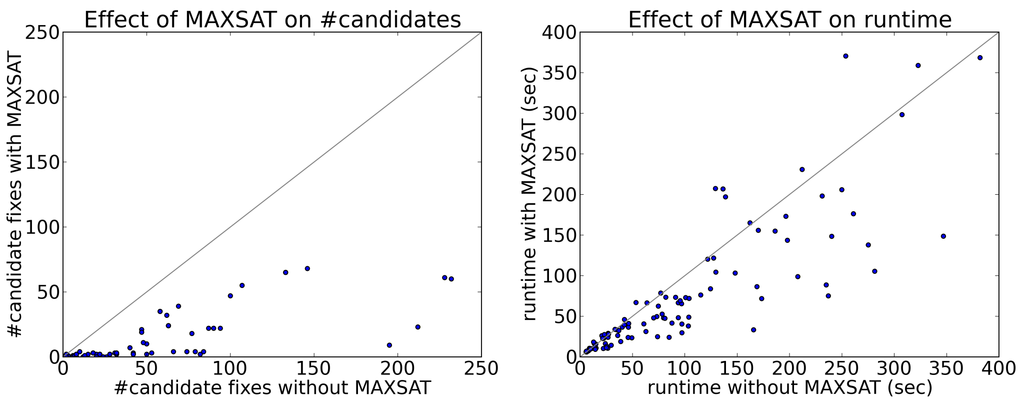


Fig. 7. The improvement in number of candidates is 0% at the minimum, 4000% at the maximum, 691% on average (geomean 408%). The runtime improvement is -38% at the minimum, 483% at the maximum, 49% on average (geomean 35%).