Semantic Program Repair Using a Reference Implementation

Sergey Mechtaev*
National University of Singapore
mechtaev@comp.nus.edu.sg

Manh-Dung Nguyen*
National University of Singapore
nguyenmanhdung1710@gmail.com

Yannic Noller
Humboldt University of Berlin
yannic.noller@informatik.hu-berlin.de

Lars Grunske
Humboldt University of Berlin
grunske@informatik.hu-berlin.de

Abhik Roychoudhury
National University of Singapore
abhik@comp.nus.edu.sg

ABSTRACT

Automated program repair has been studied via the use of techniques involving search, semantic analysis and artificial intelligence. Most of these techniques rely on tests as the correctness criteria, which causes the test overfitting problem. Although various approaches such as learning from code corpus have been proposed to address this problem, they are unable to guarantee that the generated patches generalize beyond the given tests. This work studies automated repair of errors using a reference implementation. The reference implementation is symbolically analyzed to automatically infer a specification of the intended behavior. This specification is then used to synthesize a patch that enforces conditional equivalence of the patched and the reference programs. The use of the reference implementation as an implicit correctness criterion alleviates overfitting in test-based repair. Besides, since we generate patches by semantic analysis, the reference program may have a substantially different implementation from the patched program, which distinguishes our approach from existing techniques for regression repair like Relifix. Our experiments in repairing the embedded Linux Busybox with GNU Coreutils as reference (and vice-versa) revealed that the proposed approach scales to real-world programs and enables the generation of more correct patches.

CCS CONCEPTS

• **Software and its engineering → Automatic programming**; *Software verification*;

KEYWORDS

Debugging, Program repair, Verification

ACM Reference Format:
Sergey Mechtaev, Manh-Dung Nguyen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. 2018. Semantic Program Repair Using a Reference Implementation. In *Proceedings of 40th International Conference on Software Engineering , Gothenburg, Sweden, May 27-June 3, 2018 (ICSE '18),* 11 pages.
https://doi.org/10.1145/3180155.3180247

*Joint first authors

ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180247

1 INTRODUCTION

Software developers spend a significant amount of time and resources for bug fixing. Automated program repair has gained prominence due to its potential to reduce the manual debugging effort by automatically suggesting patches for given defects. Indeed, state-of-the-art program repair systems have been shown to be able to address defects in real-world software. However, the low quality of automatically generated patches remains a great barrier to the adoption of this technology by software developers in practice.

Problem. The primary reason for the low quality of automatically generated patches is *the lack of specifications* of the intended behavior. Most program repair systems rely on tests as the correctness criteria, because a formal specification is often unavailable in practice. However, since tests is an incomplete specification, generated patches often do not correspond to developer intentions, but merely overfit the tests. In order to increase the quality of automatically generated patches, researchers have proposed such techniques as patch prioritization [22], anti-patterns [37], test generation [43, 46], etc. Although these techniques increase the probability of finding correct patches, they nevertheless do not provide any *correctness guarantees* beyond the tests in a given test suite.

Intuition. To address the overfitting problem, we propose to automatically infer the missing specification for a buggy program from a correct reference program. A reference program is an alternative realization of the same functionality, which is often available for libraries (e.g. standard library implementations, audio codecs, compression algorithms, parsers, cryptographic algorithms), network protocols [20], commodity software (e.g. GNU Coreutils and Busybox implement the same set of UNIX utilities), in the area of digital signal processing [19], web servers and database management systems. Note that a reference program may have a substantially different implementation (different data structures and algorithms), which distinguishes our approach from repair techniques [35] that employ previous program versions. The use of a reference program enables us to alleviate test overfitting and provides additional correctness guarantees.

Challenges. Ideally, a generated patch should enforce the equivalence of the patched and the reference programs, which poses two challenges: scalability and applicability. First, a recent work [14] reported that a straightforward combination of an equivalence checking system [13] with counterexample-guided inductive synthesis [1] to synthesize equivalence-enforcing patches is not scalable. Second, real-world implementations of the same functionality rarely follow precisely the same specification, e.g. GNU Coreutils

implements a superset of the functionality implemented in Busybox and therefore cannot be directly used for equivalence checking.

Solution. To address the above challenges, we introduce a methodology of patch generation based on a reference implementation that integrates the notion of conditional equivalence [10] and a recent scalable patch generation algorithm [26]. We rely on the user insight to provide an *input condition* for patch generation that should (1) include a bug-triggering input and (2) correspond to functionality shared by the buggy and the reference programs. Then, our system automatically generates a patch for the buggy program that enforces *conditional equivalence* of the patched and the reference programs, that is equivalence for all inputs satisfying the provided condition. Although the user is only required to provide an input condition (the *property* being checked is derived automatically from the reference program), this still may be non-trivial for applications that involve a complex execution setup. To tackle this problem, we propose a practical approach of introducing an input condition based on the idea of parameterized tests [38], i.e. the condition is defined by injecting symbolic parameters into existing tests.

Contributions. The main contributions of this work are:

(1) We propose to infer a correct specification from a reference implementation and use it to guide program repair in order to address the test overfitting problem.
(2) We introduce a scalable algorithm of patch generation based on a reference implementation that guarantees conditional equivalence of the patched and the reference programs w.r.t. a user-defined input condition.
(3) We conduct an evaluation on two implementations of UNIX utilities (GNU Coreutils and Busybox) that demonstrates that our methodology addresses the test overfitting problem of program repair and scales to real-world software.

2 OVERVIEW

Our approach takes four inputs: a test suite, a buggy and a reference program, and a user-defined input condition (Figure 1).

As the first step, the node *Fault localization* of Figure 1 represents the identification of suspicious expressions that might need to get repaired. This is done by applying statistical fault localization [9] based on the given test suite and the buggy program. The suspicious expressions in the buggy program get replaced with symbolic variables, denoted as the instrumented buggy program.

As the second step, the module *Symbolic analysis* of Figure 1 contains the symbolic execution of the reference program and the instrumented buggy program using the user-defined input condition as a precondition. The result of each symbolic execution is a set of pairs of resulting path conditions and symbolic output states (see Definition 4.2) that is used as a specification.

As the last step, the inferred specifications for the reference and the buggy programs are passed into the patch generator that performs a *counterexample-guided inductive repair* loop. Specifically, it performs the following iterations starting from the original buggy expression as the initial (empty) patch:

(1) Construct a verification condition (VC) for the patch.
(2) Generate a counterexample input that violates the conditional equivalence property by solving VC.

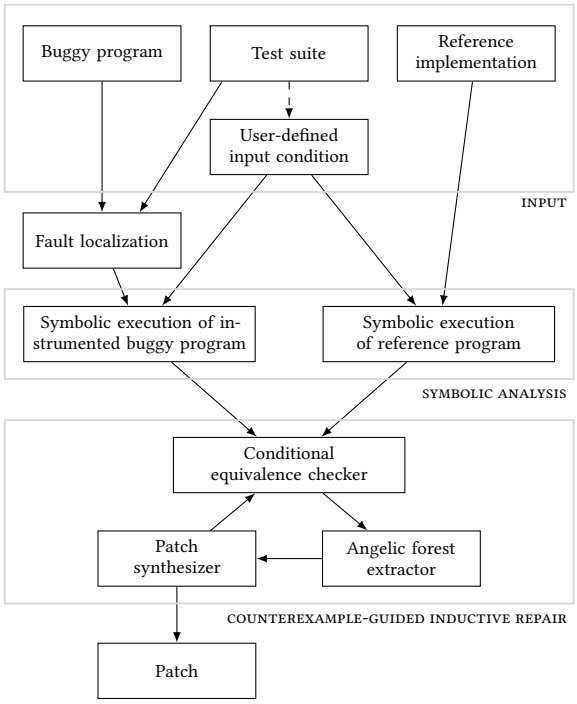

Figure 1: Overall workflow of the approach.

(3) Extract an *angelic forest* [26] for the generated input.
(4) Synthesize a patch that satisfies the angelic forest.
(5) Go to step (1).

This loop repeats until a conditional equivalence-enforcing patch is synthesized or the next patch/angelic forest cannot be found.

To illustrate our approach, we consider the reference program in Figure 2a and the buggy program in Figure 2c. The reference program implements an algorithm of searching for an element of an array via linear search, while the buggy programs uses binary search. The buggy program contains a bug in line 16.

A crucial element of our approach is the input condition ϕ that has to be defined by the user. The most trivial choice of the input condition would be simply *True*, i.e., checking equivalence for all program input. However, this approach may have a poor scalability as it has been reported in previous works [14]. Instead, we suggest defining the input condition by parameterizing existing tests. Specifically, not all inputs of a test case must be considered concretely, some of them can be handled symbolically. Therefore, the user can transform the test cases in logical constraints and possibly add additional constraints. This represents a practical solution to balance the completeness and scalability of automated program repair. The wider the input condition is formulated, the more complete, in terms of covered input space, will be the generated patch.

To define the input condition ϕ, assume that the user formulates it informally in the following way: we only consider sorted arrays of the length 3 and without duplicates. First, we introduce a mapping between program variables and symbolic variables:

$$\{x \mapsto \gamma, a \mapsto [\alpha_0, \alpha_1, \alpha_2], length \mapsto \delta\}$$

```
1  int search(int x, int a[], int length) {
2    int i;
3    for (i=0; i<length; i++) {
4      if (x == a[i])
5        return i;
6    }
7    return -1;
8  }
```

(a) Reference program

ID	$\pi^{\mathbf{r}}$	$\theta^{\mathbf{r}}_{out}$
r_1	$\gamma = \alpha_0$	0
r_2	$\gamma \neq \alpha_0 \wedge \gamma = \alpha_1$	1
r_3	$\gamma \neq \alpha_0 \wedge \gamma \neq \alpha_1 \wedge \gamma = \alpha_2$	2
r_4	$\gamma \neq \alpha_0 \wedge \gamma \neq \alpha_1 \wedge \gamma \neq \alpha_2$	-1

(b) Summary of reference program

```
9  int search(int x, int a[], int length) {
10   int L = 0;
11   int R = length-1;
12   do {
13     int m = (L+R)/2;
14     if (x == a[m]) {
15       return m;
16     } else if (x < a[m]) { // bug fix: x > a[m]
17       L = m+1;
18     } else {
19       R = m-1;
20     }
21   } while (L <= R);
22   return -1;
23 }
```

(c) Buggy program

ID	$\pi^{\mathbf{b}}$	$\theta_{\mathbf{c}}$	$\theta^{\mathbf{b}}_{out}$
b_1	$\gamma = \alpha_1$	-	1
b_2	$\gamma \neq \alpha_1 \wedge \beta^0 \wedge \gamma = \alpha_2$	$\beta^0 : \{x \mapsto \gamma, a[m] \mapsto \alpha_1\}$	2
b_3	$\gamma \neq \alpha_1 \wedge \beta^0 \wedge \gamma \neq \alpha_2 \wedge \beta^1$	$\beta^0 : \{x \mapsto \gamma, a[m] \mapsto \alpha_1\}$ $\beta^1 : \{x \mapsto \gamma, a[m] \mapsto \alpha_2\}$	-1
b_4	$\gamma \neq \alpha_1 \wedge \beta^0 \wedge \gamma \neq \alpha_2 \wedge \neg\beta^1$	$\beta^0 : \{x \mapsto \gamma, a[m] \mapsto \alpha_1\}$ $\beta^1 : \{x \mapsto \gamma, a[m] \mapsto \alpha_2\}$	-1
b_5	$\gamma \neq \alpha_1 \wedge \neg\beta^0 \wedge \gamma = \alpha_0$	$\beta^0 : \{x \mapsto \gamma, a[m] \mapsto \alpha_1\}$	0
b_6	$\gamma \neq \alpha_1 \wedge \neg\beta^0 \wedge \gamma \neq \alpha_0 \wedge \beta^1$	$\beta^0 : \{x \mapsto \gamma, a[m] \mapsto \alpha_1\}$ $\beta^1 : \{x \mapsto \gamma, a[m] \mapsto \alpha_0\}$	-1
b_7	$\gamma \neq \alpha_1 \wedge \neg\beta^0 \wedge \gamma \neq \alpha_0 \wedge \neg\beta^1$	$\beta^0 : \{x \mapsto \gamma, a[m] \mapsto \alpha_1\}$ $\beta^1 : \{x \mapsto \gamma, a[m] \mapsto \alpha_0\}$	-1

(d) Specification of buggy program

Negative input	$x \mapsto 2$ $a \mapsto [2, 4, 6]$ $length \mapsto 3$
Expected output	0
Symbolic inputs	$x \mapsto \gamma$ $a \mapsto [\alpha_0, \alpha_1, \alpha_2]$ $length \mapsto \delta$
Input condition	$\phi \coloneqq \alpha_0 < \alpha_1 < \alpha_2 \wedge \delta = 3$

(e) Test and input condition

$$VC = \forall\alpha_0\forall\alpha_1\forall\alpha_2\forall\gamma \bigwedge_{(\pi^r,\theta^r_{out})} \bigwedge_{(\pi^b,\theta^b_{out})} \pi^r \wedge \pi^b \wedge (\beta = e[\![\theta_c]\!]) \Rightarrow \theta^r_{out} = \theta^b_{out}$$

$$\equiv \forall\alpha_0\forall\alpha_1\forall\alpha_2\forall\gamma($$

$$^{(r_1,b_3)}\ \neg(\gamma = \alpha_0 \wedge \gamma \neq \alpha_1 \wedge \beta^0 \wedge \beta^0 = \gamma < \alpha_1 \wedge \gamma \neq \alpha_2 \wedge \beta^1 \wedge \beta^1 = \gamma < \alpha_2)$$

$$^{(r_1,b_4)}\ \wedge \neg(\gamma = \alpha_0 \wedge \gamma \neq \alpha_1 \wedge \beta^0 \wedge \beta^0 = \gamma < \alpha_1 \wedge \gamma \neq \alpha_2 \wedge \neg\beta^1 \wedge \beta^1 = \gamma < \alpha_2)$$

$$^{(r_3,b_6)}\ \wedge \neg(\gamma = \alpha_2 \wedge \gamma \neq \alpha_1 \wedge \neg\beta^0 \wedge \beta^0 = \gamma < \alpha_1 \wedge \gamma \neq \alpha_0 \wedge \beta^1 \wedge \beta^1 = \gamma < \alpha_0)$$

$$^{(r_3,b_7)}\ \wedge \neg(\gamma = \alpha_2 \wedge \gamma \neq \alpha_1 \wedge \neg\beta^0 \wedge \beta^0 = \gamma < \alpha_1 \wedge \gamma \neq \alpha_0 \wedge \neg\beta^1 \wedge \beta^1 = \gamma < \alpha_0))$$

(f) Verification condition

Figure 2: Artifacts of motivating example

Then, the input constraint is defined as follows:

$$\phi \coloneqq \alpha_0 < \alpha_1 < \alpha_2 \wedge \delta = 3$$

The given test suite contains one negative test case with the input $\{x \mapsto 2, a \mapsto [2,4,6], length \mapsto 3\}$ and the expected output 0, because the first element is equal to the searched value 2. The test case passes for the reference program, but fails for the buggy program. The statistical fault localization identifies the expression in line 16 as a suspicious expression, hence, we introduce the symbolic variable β and generate an instrumented buggy program by replacing (x < a[m]) with β. Note that the test case is not encoded in the input condition ϕ, i.e., the repair steps themselves are independent from any given concrete test input. This test case is only needed for the identification of suspicious expressions.

Assuming ϕ, we execute the reference and instrumented buggy program with a *preconditioned symbolic execution*. Preconditioned symbolic execution (see Definition 4.1) explores only a subset of program paths that are consistent with the condition ϕ. We will get the results presented in Figure 2b and Figure 2d. The tables contain so called *specifications* (see Definition 4.3), which describe the path condition (π^r for the reference program and π^b for the buggy program), the current context for the suspicious expression θ_c, and the output symbolic state (θ^r_{out} for the reference program and θ^b_{out} for the buggy program). The superscript index of β indicates the occurrence id (or instance id) of this expression, since it can be visited more than once during the execution.

With the results of the preconditioned symbolic execution we can build the formula of the verification condition for the considered expression (x < a[m]). A verification condition (VC) encodes the following idea: if both executions in the reference and buggy program follow the same path, then their outputs should be the same. The VC will also encode the values of the visible variables in the expression (x < a[m]) computed in the symbolic context θ_c that we denote as $\beta = (\text{x < a[m]})[\![\theta_c]\!]$. The simplified version of this first iteration VC is presented in Figure 2f. We skipped contradicting pairs of path conditions from the buggy and the reference program and discarded pairs that contradict the input condition ϕ. Additionally, we simplified the formula by removing lines where the symbolic output states match already, i.e., $\theta^r_{out} = \theta^b_{out}$. In such cases the implication is always $True$ and, hence, it does not provide any additional value. The remaining formula includes the following combinations of paths (represented by the according ids in Figure 2b and Figure 2d): (r_1, b_3), (r_1, b_4), (r_3, b_6), (r_3, b_7), as also indicated at the beginning of each line in the shown VC.

In order to check the validity of the verification condition, we check the unsatisfiability of its negation as in previous works [13, 28]. The negated VC will be solved using an off-the-shelf SMT solver to generate satisfying values representing counterexamples, which

do not satisfy the VC with the current replacement for the suspicious expression. After negating the VC, the SMT solver generates a counterexample input $\{x \mapsto -1, a \mapsto [-1, 0, 1]\}$.

In order to find the correct truth value for β, also called angelic value (see Definition 3.6), we look at the path condition of the buggy program that leads to the correct output symbolic state, which is here $\theta^r_{out} = 0$ according to the reference program. Comparing with the table in Figure 2d, this output can only be reached in the buggy program by following the path b_5. In order to take this specific path with the given input values, β^0 needs to be *False*. This leads to the following angelic forest, which is the input structure for our synthesizer and represents all needed values for the specific suspicious expression (see Definition 3.8):

$$\{(\beta^0, c, \sigma)\}, \text{ given that } c := \mathit{False}, \sigma := \{x \mapsto -1, a[m] \mapsto 0\}$$

where c represents an angelic value of the considered expression (a value that enables the program to pass the counterexample test) and σ represents an angelic state (the concrete values of program variables in the context in which the expression is executed).

The generated values are used to build the input for a component-based synthesizer, which generates a new patch matching the current synthesizer input. Given this input, the synthesizer returns a plausible patch (`x == a[m]`). After inserting this expression in the VC and negating it, the SMT solver generates a second counterexample input $\{x \mapsto 1, a \mapsto [-1, 0, 1]\}$. The correct output symbolic state for this input is $\theta^r_{out} = 2$ and this matches only the path b_2. In order to take this specific path with the second counterexample, β^0 needs to be *True*. This leads to an extension of our angelic forest to:

$$\{\ (\beta^0, \mathit{False}, \{x \mapsto -1, a[m] \mapsto 0)\},$$
$$(\beta^0, \mathit{True}, \{x \mapsto 1, a[m] \mapsto 0\})\ \}.$$

Given this input, the synthesizer returns the patch (`x >= a[m]`). After inserting this expression in the VC and negating it, the SMT returns *unsatisfiable*, i.e., the synthesized patch fulfills all requirements. Note that (`x >= a[m]`) is not syntactically equivalent with the correct patch (`x > a[m]`), but in this context (i.e. with the preceding if-condition) both expressions are *semantically* equivalent. Our approach results in a patch for the buggy program, so that given the input condition ϕ, the patched program is conditionally equivalent with the reference program.

For comparing with test-driven repair techniques, we applied Angelix [26], which uses a similar path generation algorithm, and hence, it represents the closest existing approach and means a more fair comparison than using any other test-driven repair technique. For our motivating example we observed that Angelix only can produce the plausible patch (`a[m] < a[m]`). It fixes only the given negative test case, so in order to generate a correct patch it would be necessary to include more test cases. Since our repair approach is capable of using another program as correctness reference, it generates the input for the synthesizer itself with the presented counterexample-guided approach.

In this motivating example we showed that with our approach it is possible to use a relatively simple reference program (e.g., the linear search) to repair an optimized program (e.g., the binary search). The two programs do not need to be structurally similar, as long as they solve the same problem.

3 BACKGROUND

In this section, we introduce the notation used to formally describe the presented algorithms, define underlying program analysis techniques, and also introduce related parts of previous patch generation methods that are reused in our approach.

We consider programs written in an imperative programming language. Programs are denoted as p_1, ..., p_k and the set of all program as $\mathcal{P}$. We define $p[e \mapsto e']$ as a program obtained from p by substituting an expression e with e'. Program variables are represented as $v_1, ..., v_k$ and the set of all program variables as $\mathcal{V}$.

The considered programming language contains a statement `assume` defined as follows:

$$\texttt{assume}(\phi) := \texttt{if}\ (\neg\phi)\ \{\ \texttt{LOOP : goto LOOP;}\ \}$$

that is this statement triggers a non-termination (an infinite loop) when the given condition does not hold.

Concrete program states (functions from program variables to values) are indicated as σ and the set of all concrete program states is denoted as Σ; two concrete program states σ_1 and σ_2 are equal iff $\forall v \in \mathcal{V}.\ \sigma_1(v) = \sigma_2(v)$. The value of an expression e evaluated in the context σ is denoted as $e[\![\sigma]\!]$. We define a concrete program execution as in the following:

Definition 3.1 (Concrete execution). A concrete execution procedure $\mathit{Exec} : \mathcal{P} \times \Sigma \to \Sigma \cup \{\omega\}$ is a function that for a given program p and a concrete input state σ_{in} returns the corresponding output state σ_{out} if the program terminates, and the literal ω otherwise.

We consider a first-order language $\mathcal{L}$. We use the letters α, β, γ and δ to denote variables from $\mathcal{L}$, and the letters π and ϕ to designate formulas from $\mathcal{L}$. We use the letter θ to indicate *symbolic program states*, that is functions from program variables to logical terms from $\mathcal{L}$ (for a program variable v, the corresponding logical term is $\theta(v)$). We express the equality of two symbolic program states θ_1 and θ_2 as the formula $\theta_1 = \theta_2 := \bigwedge_{v \in \mathcal{V}} \theta_1(v) = \theta_2(v)$. We denote a logical term computed by evaluating an expression e in the context θ as $e[\![\theta]\!]$.

Assume that $\{\alpha_1 \mapsto n_1, ..., \alpha_k \mapsto n_k\}$ is an assignment of the variables from $\mathcal{L}$ (a mapping from the variables to values). We say that this assignment *satisfies* a formula π iff a substitution of the variables α_i with the corresponding values n_i (denoted as $\pi[\![\alpha_1 \mapsto n_1, ..., \alpha_k \mapsto n_k]\!]$) evaluates to *True*. We also introduce a concretization of symbolic states defined as follows:

Definition 3.2 (Concretization). Let θ be a symbolic state, $\{\alpha_1 \mapsto n_1, ..., \alpha_k \mapsto n_k\}$ be an assignment of the variables from $\mathcal{L}$. A *concretization* $\theta[\![\alpha_1 \mapsto n_1, ..., \alpha_k \mapsto n_k]\!]$ of θ with the assignment $\{\alpha_1 \mapsto n_1, ..., \alpha_k \mapsto n_k\}$ is defined as follows:

$$\theta[\![\alpha_1 \mapsto n_1, ..., \alpha_k \mapsto n_k]\!] := \lambda v.\ \theta(v)[\![\alpha_1 \mapsto n_1, ..., \alpha_k \mapsto n_k]\!]$$

that is as a concrete program state (a mapping of program variables into values expressed using lambda notation) computed by substituting all the variables α_i in the logical terms in the codomain of θ with the corresponding values n_i.

In order to infer a specification, we rely on *symbolic execution* [12] that we non-constructively define as follows:

Definition 3.3 (Symbolic execution). A symbolic execution procedure $\mathit{SymExec} : \mathcal{P} \times \Theta \to 2^{\mathcal{L} \times \Theta}$ is a function that for a given

program p and a symbolic input state θ_{in} returns a finite set of pairs $\{(\pi, \theta_{out})\}$, where π is the path condition and θ_{out} is the corresponding symbolic output state. For each assignment of the symbolic variables $\{\alpha_1 \mapsto n_1, ..., \alpha_k \mapsto n_k\}$, if this assignment satisfies the formula π, then $\sigma_{out} = Exec(p, \sigma_{in})$, given that $\sigma_{in} := \theta_{in}[\![\alpha_1 \mapsto n_1, ..., \alpha_k \mapsto n_k]\!]$ and $\sigma_{out} := \theta_{out}[\![\alpha_1 \mapsto n_1, ..., \alpha_k \mapsto n_k]\!]$.

Definition 3.4 (Partial equivalence under σ). Let p_1 and p_2 be programs, σ be a program state. We say that p_1 and p_2 are *partially equivalent* under σ iff at least one of the following holds:

- $Exec(p_1, \sigma) = \omega$;
- $Exec(p_2, \sigma) = \omega$;
- $Exec(p_1, \sigma) = Exec(p_2, \sigma)$.

Since proving partial equivalence for all inputs is currently infeasible when dealing with large real-world programs, this work focuses on conditional equivalence [10] — a relaxed notion of partial equivalence that checks equivalence only for a subset of inputs.

Definition 3.5 (Conditional partial equivalence). Let p_1 and p_2 be programs, $\phi \in \mathcal{L}$ be an input condition. We say that p_1 and p_2 are *conditionally partially equivalent* under an input condition ϕ iff they are partially equivalent under each input in $\{\sigma \mid \phi[\![\sigma]\!] = True\}$.

In order to synthesize patches, we rely on a scalable patch generation methodology proposed in Angelix [26] that infers a compact synthesis specification based on *angelic values* [5] and synthesizes a patch using component-based patch synthesis [26].

Definition 3.6 (Angelic value). Let p be a program, t be a failing test, e be a program expression and e^i be its i-th instance in the execution trace of t. *Angelic value* c is such that replacing expressions e^i with c during the execution of t makes p pass the test t.

Definition 3.7 (Angelic path). Let p be a program, E be a set of program expressions in p, t be a test. An *angelic path* is a set of triples (e^i, c, σ) where e^i is the i-th instance of an expression $e \in E$ appearing in the execution trace of t, c is an angelic value of e^i, and σ represents an angelic state at e^i, such that replacing all e^i in the execution trace of t with the corresponding angelic values c forces (1) the program p to pass the test t and (2) the program p to be in the state σ when the expression e^i is evaluated.

Definition 3.8 (Angelic forest). Let p be a program, E be a set of program expressions in p, t be a test. *Angelic forest* A_t for the test t is a set of angelic paths for t.

An angelic forest can be extracted as shown in Algorithm 1. For a given program p with an expression e, this algorithm accepts a test (a pair of input and output states) and a set of triples $\{ (\pi, \theta_c, \theta_{out}) \}$. The set of triples is computed using symbolic execution in such a way that π represents a path condition in p, θ_c depicts a symbolic state that the program p reaches when the expression e is evaluated along the path π, and θ_{out} represents the symbolic output state computed along the path π. The algorithm iterates through the given triples and extracts angelic values by solving the path constraint conjoined with the given input-output relation.

Given an angelic forest, Angelix [26] can construct an expression from a given library of components that satisfies a given angelic forest. More formally, for a given angelic forest A_t and a set of components c_1, ..., c_n, it produces an expression e constructed from c_1, ..., c_n that satisfies the following property:

$$\bigvee_{path \in A_t} \quad \bigwedge_{(e^i, c, \sigma) \in path} e[\![\sigma]\!] = c$$

Such e takes the angelic value c for each angelic state σ and therefore passes the test t by construction.

Algorithm 1: Angelic forest extraction

```
Data: test (σ_in, σ_out), set of triples { (π, θ_c, θ_out) }
Result: angelic forest A_t
A_t := ∅;
foreach (π, θ_c, θ_out) do
    ψ := π ∧ θ = σ_in ∧ θ_out = σ_out;
    if isSAT(ψ) then
        {α ↦ n_1, β ↦ n_2, ...} := getSatisfyingAssignment(ψ);
        c := n_2;
        σ_c := θ_c[[α ↦ n_1, β ↦ n_2, ...]];
        A_t := A_t ∪ { (e, c, σ_c) };
    end
end
return A_t;
```

4 OUR APPROACH

In this section, we formally define three main components of our algorithm: specification inference, verification condition construction and patch synthesis.

4.1 Specification inference

The main intuition of our approach is that it is possible to infer a correct specification from a reference implementation and synthesize a patch that enforces this specification in a given buggy program. Hereinafter, we refer to the given reference implementation as the program p_r and the given buggy program as the program p_b.

In order to infer a specification, we use preconditioned symbolic execution defined as follows:

Definition 4.1 (Preconditioned symbolic execution). A *preconditioned symbolic execution* procedure $PSymExec : \mathcal{P} \times \Theta \times \mathcal{L} \to 2^{\mathcal{L} \times \Theta}$ is a symbolic execution, in which each path condition is conjoined with a given formula. It can be defined as $PSymExec(p, \theta, \phi) := SymExec(p', \theta)$, where $p' :=$ `assume` ϕ; p.

The implementation of preconditioned symbolic execution is discussed in Section 5. As a result of adding the condition ϕ, preconditioned symbolic execution is significantly more efficient than conventional symbolic execution, since it explores only a subset of program paths that is consistent with the condition ϕ.

For a given reference program and an input condition, we infer a symbolic summary of the program computed through preconditioned symbolic execution:

Definition 4.2 (Symbolic summary). Let p be a program, ϕ be an input condition, θ is a symbolic state. A *symbolic summary* is a set of pairs $Sum(p, \theta, \phi) := \{(\pi, \theta_{out})\}$ such that $\{(\pi, \theta_{out})\} = PSymExec(p, \theta, \phi)$.

For a given buggy program, a suspicious expression and an input condition, we infer the following specification:

Definition 4.3 (Specification). Let p be a program, e be a program expression, ϕ be an input condition, θ is a symbolic state over variables $\alpha_1, ..., \alpha_k$. A *specification* is a set of triples $Spec(p, e, \theta, \phi) := \{(\pi, \theta_c, \theta_{out})\}$ such that $\{(\pi, \theta_{out})\} = PSymExec(p', \theta, \phi)$, where $p' := p[e \mapsto \texttt{choose()}]$, choose() is a function that returns a fresh symbolic variable β_i each time it is executed. For each path π, θ_c indicates a symbolic state in the context of which the function choose() is called when the program is symbolically executed along π.

We say that a summary $Sum(p, \theta, \phi)$ is *complete* if for each input σ satisfying the condition ϕ one of the following holds:

- $Exec(p, \sigma) = \omega$;
- $\exists(\pi, \theta_{out}) \in Sum(p, \theta, \phi).\ \pi[\![\sigma]\!] = True$.

The completeness of a specification $Spec(p, e, \theta, \phi)$ can be defined in a similar manner.

In order to simplify further definitions, we assume (without loss of generality) that all formulas contain only a single variable α representing program inputs and a single variable β representing the values of the replaced program expression.

4.2 Verification condition

To check program equivalence, we construct a verification condition for a given patch using the inferred specification. Ideally, this condition should express the property "for each input satisfying a given input condition, if there is a path in the reference program followed by this input, then there should be a path in the patched program followed by this input and the outputs produced along these paths are equal". We refer to this condition as strict.

Definition 4.4 (Strict verification condition). Let p_r be a reference program, p_b be a buggy program, e be a suspicious expression in p_b, e' be a candidate patch, ϕ be an input condition, θ_{in} is a symbolic state over the variable α. A *strict verification condition* VC_{strict} for the program $p_b[e \mapsto e']$ is defined as follows:

$$\forall\alpha\exists\beta \bigwedge_{(\pi^r, \theta^r_{out})} (\pi^r \Rightarrow \bigvee_{(\pi^b, \theta^b_c, \theta^b_{out})} \pi^b \wedge \beta = e'[\![\theta^b_c]\!] \wedge \theta^r_{out} = \theta^b_{out})$$

where the symbolic summary $\{(\pi^r, \theta^r_{out})\} := Sum(p_r, \theta_{in}, \phi)$ and the specification $\{(\pi^b, \theta^b_c, \theta^b_{out})\} := Spec(p_b, e, \theta_{in}, \phi)$.

However, the above condition cannot be used in many practical situations, where the existing symbolic execution engines reach their limits. For instance, we have to restrict the number of explored paths by performing loop unrolling in up to k iterations. As a result, the inferred specification is incomplete, and the introduced strict verification condition may classify two equivalent programs as non-equivalent. For example, if an input is captured by some path condition π^r, but not captured by any π^b, then the programs will be considered non-equivalent. To address this, we use a more practical version of the verification condition that we refer to as liberal.

Definition 4.5 (Liberal verification condition). Let p_r be a reference program, p_b be a buggy program, e be a suspicious expression in p_b, e' be a candidate patch, ϕ be an input condition, θ_{in} is a symbolic state over the variable α. A *liberal verification condition* $VC_{liberal}$ for the program $p_b[e \mapsto e']$ is defined as follows:

$$\forall\alpha\exists\beta \bigwedge_{(\pi^r, \theta^r_{out})} \bigwedge_{(\pi^b, \theta^b_c, \theta^b_{out})} \pi^r \wedge \pi^b \wedge \beta = e'[\![\theta^b_c]\!] \Rightarrow \theta^r_{out} = \theta^b_{out}$$

where the symbolic summary $\{(\pi^r, \theta^r_{out})\} := Sum(p_r, \phi, \theta_{in})$ and the specification $\{(\pi^b, \theta^b_c, \theta^b_{out})\} := Spec(p_b, e, \phi, \theta_{in})$.

Compared with the strict verification condition, the liberal one only requires that for all intersections between a path condition π^r in the reference program and π^b in the buggy program (i.e. inputs satisfying $\pi^r \wedge \pi^b$), the symbolic outputs are the same in both programs. In the other words, this verification condition only checks equivalence of the functionality for which a specification has been inferred from both programs.

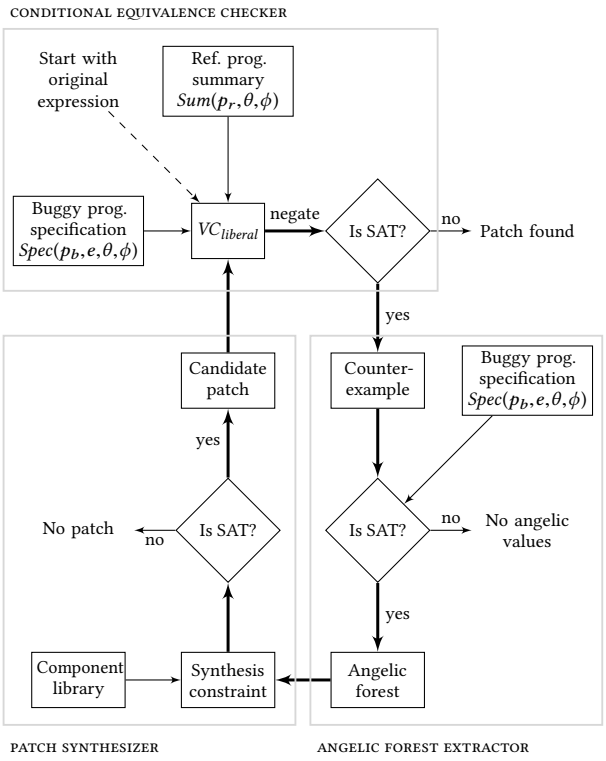

Figure 3: Counterexample-guided inductive repair.

4.3 Patch generation

To implement a scalable patch generation that enforces conditional equivalence of the reference and the buggy programs, we propose a methodology of *counterexample-guided inductive repair* (CEGIR) that effectively combines counterexample-guided inductive synthesis (CEGIS) [1] and a patch synthesis algorithm of Angelix [26].

The overall workflow of the patch generator is shown in Figure 3 and illustrated by an example in Section 2. It performs a counterexample-guided refinement loop starting from the original expression as the initial candidate patch. The loop combines three modules: a conditional equivalence checker, an angelic forest extractor and a patch synthesizer that are described in details below.

Conditional equivalence checker. In order to verify that a given candidate patch makes the buggy program conditionally equivalent to the reference program, we solve the liberal verification condition given in Definition 4.5. In order to solve the universally-quantified formula, we check the unsatisfiability of its negation, so as in previous works [13, 28]. Note that the introduced verification condition is an ∀∃ formula, therefore its negation also produces a universally-quantified formula. However, the quantifiers ∀∃ can be replaced with ∀∀, since for each α the values of β is uniquely identified by the constraint $\pi^b \wedge \beta = e'[\![\theta_c^b]\!]$. Thus, the negation of the liberal verification condition in Definition 4.5 is

$$\exists\alpha\exists\beta \bigvee_{(\pi^r,\theta_{out}^r)} \bigvee_{(\pi^b,\theta_c^b,\theta_{out}^b)} \pi^r \wedge \pi^b \wedge \beta = e'[\![\theta_c^b]\!] \wedge \theta_{out}^r \neq \theta_{out}^b$$

The above formula is an ∃∃ formula, therefore it can be solved using an off-the-shelf SMT solver. If the formula is unsatisfiable, then the patch is correct (conditionally equivalent to the reference program). Otherwise, a counterexample test is generated.

Angelic forest extractor. Given a counterexample test and a specification inferred from the buggy program, our algorithm computes a compact specification for the expression based on angelic values (angelic forest). The algorithm of angelic forest extraction is similar to that used in Angelix [26]. It is presented in Algorithm 1. If angelic values cannot be extracted, then the bug cannot be fixed at the considered location (or the specification is incomplete). Otherwise, the values are extracted and passed to the synthesizer.

Patch synthesizer. Since the input to the synthesizer is an angelic forest, we used the Angelix implementation of a patch synthesizer that extends SMT-based component-based program synthesis [26]. If a patch cannot be synthesized, then the search space (the set of considered transformations) is insufficient to find a repair. If a patch is found, it is passed to the conditional equivalence checker.

Proposition 4.6 (Correctness guarantee). *Let p_b be a buggy program, p_r be a reference program, ϕ be an input condition, e be a suspicious expression in p_b. Assume that e' is a patch that is produced by the CEGIR algorithm (Figure 3) given complete specifications $Sum(p_r,\theta,\phi)$ and $Spec(p_b,e,\theta,\phi)$ as inputs. Then, $p_b[e \mapsto e']$ and p_r are conditionally partially equivalent w.r.t. the condition ϕ.*

5 IMPLEMENTATION

We have implemented the tool SemGraft for evaluating our technique. SemGraft consists of three main components: *preconditioned symbolic executor*, *verification condition generator* and *patch generator*. SemGraft receives a buggy and a reference program, an input condition and a test suite as input, and produces a patch for the buggy program as the output. Figure 4 shows the architecture of our tool. Below, we explain how these components are implemented.

Preconditioned symbolic executor (PSE). PSE is built on top of KLEE [4] — a widely used symbolic execution engine. To support preconditioned symbolic execution, the modified version of KLEE takes the user-defined input condition ϕ in SMTLIB2 format as input. Specifically, we modify the function *fork* of the KLEE interpreter which is called when KLEE encounters a symbolic branch. The path conditions of both branches are conjoined with the input condition to determine whether a path is terminated immediately or further explored. We integrate Z3 solver [6] with KLEE and pass symbolic constraints between them for checking the satisfiability of symbolic expressions. PSE outputs symbolic formula in SMTLIB2 format and invokes Z3 solver via a wrapper function.

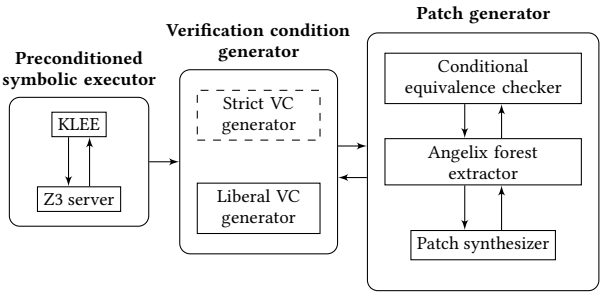

Figure 4: Architecture of SemGraft.

Verification condition generator (VCG). VCG takes the symbolic summary of the reference program and the specification of the buggy program, both of which are obtained by executing PSE with the input condition. Our tool SemGraft supports both kinds of verification condition as per Subsection 4.2, but the default option of VCG is the liberal verification condition which is more practical for real-world programs. We use an open-source library jSMTLIB[1] for processing SMT files generated by PSE.

Patch generator (PG). PG takes the liberal verification condition in SMTLIB2 format and executes a counterexample-guided inductive repair loop until it finds a patch that satisfies the desired property. The workflow of PG is shown in Figure 3.

6 EXPERIMENTAL EVALUATION

To evaluate the effectiveness of our approach, we aim to investigate the following research questions:

(RQ1) Can SemGraft generate repairs for real-world software?
(RQ2) Can SemGraft alleviate the overfitting problem of existing test suite based program repair techniques using the reference implementation?

RQ1 is designed to investigate the applicability of our approach for repairing real-world applications. A previous study [14] has reported that a straightforward combination of a state-of-the-art equivalence checking system with counterexample-guided inductive systems scales only to small programs. To be applicable to real-world programs such as Busybox and GNU Coreutils, our approach sacrifices discovers a partial specification for checking conditional equivalence w.r.t. a user-provided condition. Therefore, we also discuss how such a condition can be derived from existing tests.

RQ2 assesses the correctness of generated patches compared with test-driven program repair approaches. As in previous works [22, 26], we identify a generated patch as *correct* only if it is *syntactically* equivalent to the developer patch (modulo trivial refactorings).

[1] jSMTLIB website: `https://github.com/smtlib/jSMTLIB`

Table 1: Busybox subject programs

Buggy Prog.	Buggy Commit	Ref. Prog.	Ref. Prog. Version	Failure Description	Angelix'	SemGraft
sed	c35545a	sed of GNU sed	version 3.01	Failed to handle zero-length match	Correct	Correct
seq	f7d1c59	seq of Coreutils	version 6.10	Wrong output when 2 input arguments are equal	Correct	Correct
sed	7666fa1	sed of GNU sed	version 3.01	Wrong output when handling `s///NUM`	Incorrect	Correct
sort	d1ed3e6	sort of Coreutils	version 8.27	Wrong output when handling`-kSTART,N.ENDCHAR`	Incorrect	Correct
seq	d86d20b	seq of Coreutils	version 8.27	`seq` no longer accepts 0 value as increment argument	Incorrect	Correct
sed	3a9365e	sed of GNU sed	version 3.01	Failed to handle `s///` which has empty matches	Incorrect	Correct

Table 2: Coreutils subject programs

Buggy Prog.	Buggy Commit	Ref. Prog.	Ref. Prog. Version	Failure Description	Angelix'	SemGraft
mkdir	f7d1c59	mkdir of Busybox	version 1.27.2	Segmentation fault	Incorrect	Correct
mkfifo	cdb1682	mkfifo of Busybox	version 1.27.2	Segmentation fault	Incorrect	Correct
mknod	cdb1682	mknod of Busybox	version 1.27.2	Segmentation fault	Incorrect	Correct
copy	f3653f0	copy of Busybox	version 1.27.2	Failed to copy a file	Correct	Correct
md5sum	739cf4e	md5sum of Busybox	version 1.27.2	Segmentation fault	Correct	Correct
cut	6f374d7	cut of Busybox	version 1.27.2	Failed to handle`-b 2-,3-` like`-b 2-`	Incorrect	Correct

6.1 Experimental setup

In order to address the described research questions, we choose the subjects in our experiments according to the following four criteria.

(1) The subjects are real-world software that is widely used.
(2) Reference programs are available that process the same inputs as the buggy programs but exhibit the correct behavior.
(3) The buggy and the reference program are substantially different in their structure.
(4) The developer patches are within the search spaces of our implementation. By the search space we mean the set of considered transformations defined through the components used for component-based synthesis.

The last criteria allows us to reason about correctness of generated patches (e.g. if the developer patch was not in the search space, then any generated patch would *a priori* be identified as incorrect).

Our subjects are 12 real software errors of two open-source C projects Busybox [39] and GNU Coreutils [40] extracted from (1) commit logs, (2) bug reports and (3) previous research [4]. Both Busybox and GNU Coreutils provide many common UNIX utilities, but Busybox has been implemented with size-optimization, limited resources, and is mainly used for small or embedded systems. We employ our tool SemGraft to repair the embedded Linux Busybox with GNU tools like Coreutils as reference, and vice versa.

To address the second question (RQ2), we compared our technique with a state-of-the-art test-driven program repair approach, Angelix [26]. Angelix is closely related to our technique since it also applies symbolic execution to infer specification and synthesizes patches. We selected this approach for our evaluation because this enables us to more objectively investigate the impact of specification inferred from a reference program. Specifically, since our implementation reuses the synthesizer of Angelix, both Angelix and SemGraft explore the same space of candidate patches. In order to ensure that the systems have access to the same semantic information about the program, we integrated the algorithm of Angelix into SemGraft in such a way that both tools use the same specification inferred from the buggy program (we refer to this version of Angelix as Angelix'). The main difference of the two tools is that SemGraft performs a counterexample-guided inductive repair loop to ensure conditional equivalence with the reference implementation, while Angelix' relies solely on the test suite provided by GNU Coreutils/Busybox developers and stops immediately when finding an expression satisfying the tests.

All our experiments were performed on Intel Xeon CPU E5-2630 v4 @ 2.20GHz CPU with Ubuntu 14.04 64-bit operating system.

6.2 Summary of experiments

Table 1 summarizes our experiments with Busybox and Table 2 summarizes our experiments with GNU Coreutils. For each pair of a buggy and a reference program, the tables show the name of the buggy program and its version in the commit history, the reference program and its version, a description of the bug, and the results of executing Angelix' and SemGraft for repairing the defect.

SemGraft demonstrated that the proposed approach can be applied to real-world programs. Specifically, it managed to repair all defects that are repaired by Angelix. Since the workflow of SemGraft also includes inferring specification for a reference program and checking verification conditions, it required a longer time to generated patches. Specifically, Angelix' required 15 minutes on average to generate patches, while SemGraft required 45 minutes.

In the experiments, SemGraft inferred specifications consisting of up to 81 paths from a reference program and up to 250 paths in a buggy program. The number of paths, in general, depends on the structure of the buggy and the reference programs, bounds used for symbolic execution and the chosen condition ϕ. Typically, the specification inferred from the buggy program includes more

```
if (!rhs_specified)
  {
    if (eol_range_start == 0 || eol_range_start == 3)
      eol_range_start = initial;
    field_found = true;
  }
```

(a) Patch generated by Angelix' based on tests.

```
if (!rhs_specified)
  {
    if (eol_range_start == 0 || initial < eol_range_start)
      eol_range_start = initial;
    field_found = true;
  }
```

(b) Patch generated by SemGraft based on reference program.

Figure 5: Generated patches for `cut` (ver. 6f374d7).

paths due to additional symbolic variables injected into the buggy program for suspicious expressions.

As can be seen from the tables, SemGraft generated repairs equivalent to developer patches for all considered examples, while Angelix' that relies only on tests repaired less than half of the defects correctly. This shows that the reference implementation indeed can help to alleviate test overfitting.

6.3 Deriving input condition

An input condition used for enforcing conditional equivalence of the patched and the reference program is an important part of our approach and it has to be defined by the user. We use an example of a bug in `cut` (ver. 6f374d7) to demonstrate how such a condition can be defined in practice. `cut` extracts sections from each line of its input. The buggy version of `cut` of GNU Coreutils wrongly interprets the command `-b 2-,3-` as `-b 3-` (extract input bytes staring from the third byte) instead of `-b 2-` (extract input bytes staring from the second byte). The developer provides the following two tests that cover the buggy functionality:

```
echo -ne '1234' | cut -b 2-,3-
echo -ne '1234' | cut -b 3-,2-
```

For both of these tests, the expected output is `234`. The above two tests cover program behavior for two concrete pairs of indexes $(2,3)$ and $(3,2)$. However, this is insufficient for a test-driven program repair to produce a patch that *generalizes* beyond the tests.

In this work, we propose to define an input condition for generating conditional equivalence-enforcing patches by *parametrizing* existing tests. Note that the purpose of parametrizing the test is to make generated patches generalize, yet ensuring tractability of specification inference. Therefore, the user should parametrize the essential part in the test related to the failure. In this case, we inject parameters instead of the indexes $\{2,3\}$ that affect the way the data is processed. As a result, we obtain the following input:

```
echo -ne '1234' | cut -b α0-,α1-
```

where α_0 and α_1 indicate the injected parameters. Given such a parametrization, the condition ϕ will be accordingly defined as:

$$\phi := arg0[0] = \text{“}-\text{”} \wedge arg0[1] = \text{“}b\text{”} \wedge$$
$$arg1[1] = \text{“}-\text{”} \wedge arg1[2] = \text{“},\text{”} \wedge arg1[4] = \text{“}-\text{”} \wedge in = \text{“}1234\text{”}$$

where *arg0* and *arg1* denote command-line arguments, *in* is the standard input stream.

This example demonstrates that defining an input condition for our approach may require a small effort from the user. However, we believe that such a condition can be potentially derived automatically. One possible way to do that would be to execute the failing test *concolically* to collect input constraints (e.g. using ZESTI [23]), and construct an input condition for our approach by generalizing the obtained constraint. We leave this for future work.

6.4 Impact of reference program

In this section, we show how the use of a reference implementation can enable SemGraft to find a correct path, while Angelix' finds a plausible (passing the given tests), but incorrect repair.

For the discussed bug in `cut` program, Angelix' uses the tests given above to construct a patch in Figure 5a. This patch adds a condition into the program that changes the way how indexes in the given command are handled. The expression includes the disjunct `eol_range_start == 3` that ensures that the index 3 is not used by the command `-b 2-,3-`. However, this condition does not generalize to other values of the indexes that can appear in such command but merely overfit the given test.

As opposite to Angelix', SemGraft extracts a specification from the buggy and the reference programs via preconditioned symbolic execution with ϕ. In this example, it extracts 30 paths from the buggy program and 18 paths from the donor program (Busybox `cut`). Then, it performs a counterexample-guided inductive repair loop until it finds a patch that enforces conditional equivalence of Coreutils `cut` and Busybox `cut` w.r.t. ϕ. Specifically, after obtaining a candidate patch as in Figure 5a, it generates a counterexample test `-b 3-,4-` for which the output of Coreutils `cut` diverges from Busybox `cut`. Based on this test, it extracts the angelic path

$$\{\ (\beta^0, \mathit{True}, \{\texttt{initial} \mapsto 3, \texttt{eol\_range\_start} \mapsto 0, \}),$$
$$(\beta^1, \mathit{False}, \{\texttt{initial} \mapsto 4, \texttt{eol\_range\_start} \mapsto 3, \})\ \}.$$

Given the extracted path, SemGraft generates the patch in Figure 5b, which is identical to the developer repair. SemGraft also proves that it is correct (equivalent to Busybox `cut`) for all possible combinations of indexes in the described command.

7 THREATS TO VALIDITY

Internal validity. The main threat to internal validity is the manual construction of the input condition ϕ for our experiments that were based on the negative, developer-written tests. In general, creating such conditions may not be trivial. However, existing techniques like delta debugging [48] could help the user to minimize the test input arguments still showing the buggy behavior, and hence, to simplify the input condition. Additionally, the performed manual inspection of the experimental results might be error-prone. To mitigate this, two authors of the paper double-checked the created input conditions and the generated patches.

External validity. The main threat to external validity is that the chosen selection of subjects may not generalize to other programs. We could not use existing repair benchmarks [17, 36] due to the

lack of reference programs. However, we focused on subjects from the GNU Coreutils and Busybox, which are well-known throughout the community and provide implementation of similar functionality. Another threat to external validity may be the scalability of our approach. It mainly depends on the number of explored paths during the symbolic analysis and the size of the generated verification condition. Therefore, among others, we introduced the user-defined correctness property, which enables the limitation of the search space to a practical usable scope. Furthermore, the approach is limited by the capabilities of the underlying symbolic execution engine, which we tried to improve by using an efficient preconditioned symbolic execution approach.

Construct validity. The main threat to construct validity is the correctness of our implementation because our statement about the conditional partial equivalence of the generated patch holds only if the implementation is correct. We implemented our approach as extensions of existing works: KLEE symbolic execution engine and a synthesizer adapted from Angelix [26]. Therefore, our extension inherits the incorrectness of the baseline. However, face-validity showed that the results are consistent with the expected outcome.

8 RELATED WORK

8.1 Program repair

In the last years, several program repair approaches have been developed. These approaches can be classified into syntax-based and semantics-based techniques. Syntax-based (generate-and-validate) program repair techniques such as e.g., GenProg [18, 42], AE [41], RSRepair[30] and ACS [44] require subtasks including fault localization, patch generation and execution of regression test cases. Mechanizing these tasks has received significant attention from the automated and search-based software engineering community. Approaches developed by Le Goues et al. [16, 18, 42], Martinez and Monperrus [7, 24] show meaningful results towards the automation of bug fixing. The main idea of these approaches is to use failed test cases to localize potential faults and then apply mutations to the source code until the program satisfies all unit test cases. The mutations that are applied to the program code can range from small changes like modification, addition or removing a single code line [18, 42] to complex edit operations [24] mined from software repositories. Relifix [35] utilizes previous program versions in order to perform automated repair of regression bugs, however it relies on syntactic similarity of the previous and the buggy programs, which distinguishes it from our approach. The quality of patches produced by test-based repair approaches may be low, since the patches may overfit the test data. A study on the correlation of patch quality with the quality of the test-suite guiding the repair has been recently conducted [47]. Semantics-based techniques like SemFix [27], Nopol [45], DirectFix [25], SPR [21], Angelix [26] and JFIX[15] split patch generation into two phases. First, they infer a synthesis specification for the identified program statements. Second, they synthesize a patch for these statements based on the inferred specification. However, since most of semantics-based techniques rely on tests as the correctness criteria, the inferred specification only captures the property of "passing the tests".

Our approach seeks to solve the test overfitting problem in program repair. We show that, subject to the availability of a reference implementation, our scalable symbolic analysis can produce higher quality patches and provide partial correctness guarantees.

8.2 Software transplantation

Automated software transplantation [2, 8, 29] tools like μScalpel [3] and CodeCarbonCopy [31] aim at transplanting new functionality from a donor application into a recipient application. CodePhage [32] can fix program errors like out of bounds access, integer overflow, and divide by zero errors. They focus on finding an error checking code in the donor application that can serve as a compensation for a missing check in the host application. Since their approach tries to copy code, it is necessary to translate the check from the data structures and name space of the donor into an application-independent representation suitable for insertion into the recipient application. The advantage of SearchRepair [11] compared to other repair approaches is the use of semantic code search [33, 34] to identify suitable code fragments for repair. Our approach differs from software transplantation literature, since we do not seek to inject any new functionality lifted from a donor program. Our goal is to increase the quality of generated patches using a different correctness criteria, namely the reference program. We also differ from recent works on grafting of code clones [49], since this line of work seeks to achieve greater test-reuse across code clones for the sake of differential testing.

8.3 Equivalence checking

Lahiri et al. [13] proposed an approach to find the rootcause for equivalence failures by leveraging semantic similarity between two program binaries. Since they aim to extract a complete specification, their approach scales only to small programs. Our work sacrifices completeness for the sake of applicability by checking conditional equivalence of a buggy program and a reference program w.r.t. a user-defined input condition.

9 CONCLUSION

We proposed a methodology of generating patches based on a reference implementation. Our technique addresses the test-overfitting problem by providing additional correctness guarantees. Specifically, it synthesizes a conditional equivalence-enforcing patch w.r.t. a user-defined input condition. Our experiments demonstrated that our method scales to real-world programs such as GNU Coreutils and Busybox and helps to generate more correct repairs.

ACKNOWLEDGEMENTS

We thank the participants of Dagstuhl Seminar 17022 on Automated Program Repair for inspiring discussions. This research is supported in part by the National Research Foundation, Prime Minister's Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2014NCR-NCR001-21) and administered by the National Cybersecurity R&D Directorate. This work is also supported by the German Research Foundation (GR 3634/4-1 EMPRESS). This work is also supported by Office of Naval Research grant ONRG-NICOP-N62909-18-1-2052.

REFERENCES

[1] Rajeev Alur, Rastislav Bodík, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shambwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2015. Syntax-Guided Synthesis. In *Dependable Software Systems Engineering*. 1–25.

[2] Earl T. Barr, Yuriy Brun, Premkumar T. Devanbu, Mark Harman, and Federica Sarro. 2014. The plastic surgery hypothesis. In *FSE*. ACM, 306–317.

[3] Earl T. Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. 2015. Automated Software Transplantation. In *ISSTA*. ACM, New York, NY, USA, 257–269.

[4] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.. In *OSDI*, Vol. 8. 209–224.

[5] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodík. 2011. Angelic debugging. In *ICSE*. 121–130.

[6] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS*. Springer-Verlag, Berlin, Heidelberg, 337–340.

[7] Thomas Durieux, Matias Martinez, Martin Monperrus, Romain Sommerard, and Jifeng Xuan. 2015. Automatic Repair of Real Bugs: An Experience Report on the Defects4J Dataset. *CoRR* abs/1505.07002 (2015).

[8] Mark Harman, Yue Jia, and William B. Langdon. 2014. Babel Pidgin: SBSE Can Grow and Graft Entirely New Functionality into a Real World System. In *SSBSE*, Vol. 8636. Springer, 247–252.

[9] James A Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of test information to assist fault localization. In *ICSE*. ACM, 467–477.

[10] Ming Kawaguchi, Shuvendu Lahiri, and Henrique Rebelo. 2010. *Conditional equivalence*. Technical Report.

[11] Yalin Ke, Kathryn T. Stolee, Claire Le Goues, and Yuriy Brun. 2015. Repairing Programs with Semantic Code Search. In *ASE*. IEEE Computer Society, 295–306.

[12] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394.

[13] Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. 2012. SYMDIFF: A Language-Agnostic Semantic Diff Tool for Imperative Programs. In *CAV*. 712–717.

[14] Shuvendu K. Lahiri, Rohit Sinha, and Chris Hawblitzel. 2015. Automatic Root-causing for Program Equivalence Failures in Binaries. In *CAV*. 362–379.

[15] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. JFIX: semantics-based repair of Java programs via symbolic PathFinder. In *ISSTA*. ACM, 376–379.

[16] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each. In *ICSE*. IEEE Press, 3–13.

[17] Claire Le Goues, Neal Holtschulte, Edward K Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. 2015. The ManyBugs and IntroClass benchmarks for automated repair of C programs. *TSE* 41, 12 (2015), 1236–1256.

[18] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A generic method for automatic software repair. *IEEE TSE* 38, 1 (2012), 54–72.

[19] Rainer Leupers. 2013. *Code optimization techniques for embedded processors: Methods, algorithms, and tools*. Springer Science & Business Media.

[20] Qing Li, Tatuya Jinmei, and Keiichi Shima. 2010. *IPv6 Core Protocols Implementation*. Morgan Kaufmann.

[21] Fan Long and Martin Rinard. [n. d.]. Staged program repair with condition synthesis. In *ESEC/FSE*. ACM, 166–178.

[22] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *POPL*. ACM, 298–312.

[23] Paul Dan Marinescu and Cristian Cadar. 2012. Make test-zesti: A symbolic execution solution for improving regression testing. In *ICSE*. IEEE Press, 716–726.

[24] Matias Martinez and Martin Monperrus. 2015. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering* 20, 1 (2015), 176–205.

[25] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. Directfix: Looking for simple program repairs. In *ICSE*. IEEE Press, 448–458.

[26] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *ICSE*. 691–701.

[27] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program Repair via Semantic Analysis. In *ICSE*. IEEE Press, Piscataway, NJ, USA, 772–781.

[28] Suzette Person, Matthew B. Dwyer, Sebastian G. Elbaum, and Corina S. Pasareanu. 2008. Differential symbolic execution. In *FSE*. 226–237.

[29] Justyna Petke, Mark Harman, William B. Langdon, and Westley Weimer. 2014. Using Genetic Improvement and Code Transplants to Specialise a C++ Program to a Problem Class. In *EuroGP*, Vol. 8599. Springer, 137–149.

[30] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The strength of random search on automated program repair. In *ICSE*. ACM, 254–265.

[31] Stelios Sidiroglou-Douskos, Eric Lahtinen, Anthony Eden, Fan Long, and Martin Rinard. 2017. CodeCarbonCopy. In *ESEC/FSE*. ACM, 95–105.

[32] Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. 2015. Automatic Error Elimination by Horizontal Code Transfer Across Multiple Applications. In *PLDI*. ACM, New York, NY, USA, 43–54.

[33] Kathryn T. Stolee, Sebastian G. Elbaum, and Daniel Dobos. 2014. Solving the Search for Source Code. *ACM Trans. Softw. Eng. Methodol.* 23, 3 (2014), 26:1–26:45.

[34] Kathryn T. Stolee, Sebastian G. Elbaum, and Matthew B. Dwyer. 2016. Code search with input/output queries: Generalizing, ranking, and assessment. *Journal of Systems and Software* 116 (2016), 35–48.

[35] Shin Hwei Tan and Abhik Roychoudhury. 2015. relifix: Automated repair of software regressions. In *ICSE*. IEEE Press, 471–482.

[36] Shin Hwei Tan, Jooyong Yi, Sergey Mechtaev, Abhik Roychoudhury, et al. 2017. Codeflaws: a programming competition benchmark for evaluating automated program repair tools. In *ICSE Companion*. 180–182.

[37] Shin Hwei Tan, Hiroaki Yoshida, Mukul R Prasad, and Abhik Roychoudhury. 2016. Anti-patterns in search-based program repair. In *FSE*. ACM, 727–738.

[38] Nikolai Tillmann and Wolfram Schulte. 2005. Parameterized unit tests. In *ACM SIGSOFT Software Engineering Notes*, Vol. 30. ACM, 253–262.

[39] Busybox Website. 2017. Busybox. https://busybox.net/. (2017).

[40] GNU Coreutils Website. 2017. GNU Coreutils. https://www.gnu.org/software/coreutils/coreutils.html. (2017).

[41] Westley Weimer, Zachary P. Fry, and Stephanie Forrest. 2013. Leveraging program equivalence for adaptive program repair: Models and first results. In *ASE*. IEEE, 356–366.

[42] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically Finding Patches Using Genetic Programming. In *ICSE*. IEEE Computer Society, 364–374.

[43] Qi Xin and Steven P Reiss. 2017. Identifying test-suite-overfitted patches through test case generation. In *ISSTA*. ACM, 226–236.

[44] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise condition synthesis for program repair. In *ICSE*. IEEE / ACM, 416–426.

[45] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian R. Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2017. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE TSE* 43, 1 (2017), 34–55.

[46] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. 2017. Better test cases for better automated program repair. In *ESEC/FSE*. ACM, 831–841.

[47] Jooyong Yi, Shin Hwei Tan, Sergey Mechtaev, Marcel Boehme, and Abhik Roychoudhury. 2018. Correlation of Test-suite Metrics with Patch Quality in Program Repair. *Empirical Software Engineering* To appear (2018).

[48] Andreas Zeller. 1999. Yesterday, My Program Worked. Today, It Does Not. Why?. In *FSE*. Springer-Verlag, London, UK, UK, 253–267.

[49] Tianyi Zhang and Miryung Kim. 2017. Automated transplantation and differential testing for clones. In *ICSE*. IEEE / ACM, 665–676.