Trident: Controlling Side Effects in Automated Program Repair

Nikhil Parasaram, Earl T. Barr, and Sergey Mechtaev

Abstract—The goal of program repair is to eliminate a bug in a given program by automatically modifying its source code. The majority of real-world software is written in imperative programming languages. Each function or expression in imperative code may have side effects, observable effects beyond returning a value. Existing program repair approaches have a limited ability to handle side effects. Previous test-driven semantic repair approaches only synthesise patches without side effects. Heuristic repair approaches generate patches with side effects only if suitable code fragments exist in the program or a database of repair patterns, or can be derived from training data. This work introduces Trident, the first test-driven program repair approach that synthesizes patches with side effects without relying on the plastic surgery hypothesis, a database of patterns, or training data. Trident relies on an interplay of several parts. First, it infers a specification for synthesising side-effected patches using symbolic execution with a custom state merging strategy that alleviates path explosion due to side effects. Second, it uses a novel component-based patch synthesis approach that supports lvalues, values that appear on the left-hand sides of assignments. In an evaluation on open-source projects, Trident successfully repaired 6 out of 10 real bugs that require insertion of new code with side effects, which previous techniques do not therefore repair. Evaluated on the ManyBugs benchmark, Trident successfully repaired two new bugs that previous approaches could not. Adding patches with side effects to the search space can exacerbate test-overfitting. We experimentally demonstrate that the simple heuristic of preferring patches with the fewest side effects alleviates the problem. An evaluation on a large number of smaller programs shows that this strategy reduces test-overfitting caused by side-effects, increasing the rate of correct patches from 33.3% to 58.3%.

Index Terms—program repair, program synthesis, symbolic execution, side effects

1 Introduction

Most existing software is written in imperative programming languages. Statements in imperative languages can have side effects: observable effects beyond returning a value to the invoker of the operation. Common side effects include changing the value of a variable, writing data to disk, or enabling or disabling a button in the user interface.

Despite the importance of modifications with side effects for real-world programs, state-of-the-art program repair tools have a limited ability to handle side effects, which restricts their applicability. Test-driven program repair approaches that rely on program synthesis, both symbolic [1] and enumerative [2], currently only synthesise side-effect-free expressions, or scale only to small programs [3]. Current heuristic repair approaches generate patches with side effects only if suitable code fragments exist in the buggy program [4] or in a pre-defined database of patterns [5], which may not contain the needed code. Machine learning approaches [6] can potentially generate patches with side effects, but their success depends on the size and the quality of training data. Finally, existing approaches for controlling test-overfitting [7], [8] do not take side effects into account. This lacuna is important, because our experiments demonstrate that patches with side effects are more prone to test-overfitting.

This work introduces TRIDENT, the first test-driven program repair approach that synthesizes patches with side effects without relying on the plastic surgery hypothesis, a database of patterns, or training data. TRIDENT effectively addresses the limitations of previous constraint-based program repair techniques [1], [9], and complements heuristic techniques [4], [10], since it does not rely on the plastic surgery hypothesis [11] or a repair pattern database.

TRIDENT takes a buggy program and its test suite, then follows the general workflow of semantic program repair [12]: it localises faulty statements, infers a specification for patching these statements using symbolic execution, and then constructs a patch via program synthesis. Assignment statements and function calls are two basic classes of statements that involve side effects. Compared to previous semantic techniques, TRIDENT supports two new defects classes: the insertion/modification of assignment statements and function calls. To repair a software defect, an automated program repair tool must (1) contain a correct patch in its search space, (2) find this patch within a time budget, and (3) reduce the chance of generating an incorrect, test-overfitting patch. TRIDENT tackles these challenges by enhancing specification inference and patch synthesis for assignment statements and function calls.

In order to include patches with side effects in its search space, TRIDENT extends existing component-based program synthesis approaches [13], [14] by introducing the concept of lvalue components, which can appear in the left-hand-sides of assignments, and rvalue components, which can appear in the right-hand-sides of assignments. Then, the basic building block of TRIDENT's patch synthesiser is a k-holed assignment that represents a simultaneous assignment of k rvalue components to k lvalue components (Section 3.4). This formalism captures the semantics of both assignment statements and function calls with side effects. Functions without loops can be precisely summarised as simultaneous assignments [15]. For functions with loops, TRIDENT com-

• *The authors are with University College London.*

Manuscript received April 19, 2005; revised August 26, 2015.

```
int clamp(int x, int l, int h) {
    int r;
    // missing assignment r = x;
    if (x < l)
        r = l;
    if (r == x && x > h)
        r = h;
  return r;
}
```

(a) A function missing an assignment.

```
int clamp(int x, int l, int h) {
    int r;
    □_1^L, ..., □_k^L = □_1^R, ..., □_k^R;
    if (x < l)
        r = l;
    if (r == x && x > h)
        r = h;
  return r;
}
```

(b) A buggy function with holes, placed before suspicious statements.

```
int clamp(int x, int l, int h) {
    int r;
    switch(PATCH_ID) {
        case 0: x = □^R; break;
        case 1: l = □^R; break;
        case 2: h = □^R; break;
        case 3: r = □^R; break; }
    if (x < l)
        r = l;
    if (r == x && x > h)
        r = h;
  return r;
}
```

(c) Eliminating lvalue-holes using a switch statement (for $k = 1$).

Fig. 1: Synthesising a patch that inserts an assignment statement.

putes summaries using loop unrolling.

The inclusion of patches with side effects in the search space causes a scalability problem during the symbolic execution that infers the specification of the holes because it exacerbates the path explosion problem. To alleviate this problem, TRIDENT leverages a novel merging strategy that rests on two insights. First, distinct variables can be updated and still generate states that traverse the same path to program exit. Second, even when many variables can be assigned, any concrete patch will affect only a few of them. When two paths write the same thing (the same rvalue) to two different variables (two different lvalues), TRIDENT exploits the first insight to efficiently merge both writes into a single state. To ensure consistency of this merging, TRIDENT appends an appropriate path constraint to the path condition (Section 3.4). It exploits the second insight to restrict the number of variables that a patch can update.

Although many bugs in imperative programs require side-effected patches to fix, some side effects can increase overfitting as our experiments demonstrate (Section 5). To address this issue, TRIDENT applies a simple patch prioritisation heuristic to minimise overfitting. We find that preferring a patch with the fewest side effects lowers overfitting.

To evaluate TRIDENT, we used three benchmarks: 10 bugs extracted from free GNU projects for evaluating TRIDENT's scalability on bugs that require patches with side effects, 36 bugs sampled from ManyBugs [16] for evaluating TRIDENT's scalability on generic defects and 110 defects extracted from Codeflaws benchmark [17] for evaluating propensity of TRIDENT's patches to overfit. The 10 bugs were the first 10 sampled uniformly whose fixes required side effects and the 110 were cut down from the complete Codeflaws dataset, again to those needing side effects (Section 5). The evaluation demonstrates that TRIDENT generates patches involving assignments and function calls for 6 out of the 10 realistic bugs. Besides, TRIDENT's patch prioritisation increases the rate of correct patches from 33.3% to 58.3% when applied to the 110 defects from Codeflaws. These results demonstrate the practicality and utility of TRIDENT.

The key contribution of this work is TRIDENT, the first scalable test-driven patch synthesis approach that addresses the memory updates/call function defect class without relying on the plastic surgery hypothesis, a database of patterns, or training data; it is enabled by a tight integration of two technical insights:

- An extension of component-based program synthesis that introduces lvalue and rvalue components to capture assignments and function calls;
- Multi-path specification inference, a state merging technique tailored to the synthesis of side effected patches that mitigates path explosion.

All code, scripts, and data necessary to reproduce this work are available at https://program-repair.s3-ap-southeast-1.amazonaws.com/trident-submission.zip.

2 OVERVIEW

For a given buggy program, test-driven program repair (TPR) techniques search for a plausible patch — a patch that passes a test-suite — in a search space of candidate patches. Since a test-suite is an incomplete specification, program repair systems utilize patch prioritization strategies to increase the probability of finding a correct patch. Thus, to repair a bug, a TPR system has to (1) contain the correct patch in its search space, (2) be efficient enough to find a plausible patch given a time budget, and (3) prioritize a correct patch over plausible, but incorrect, patches.

TRIDENT is the first TPR technique to synthesise side-effected patches. This fundamental advance exacerbates all three of TPR's seminal problems. Section 2.1 presents the resulting challenges; the rest of the section overviews how TRIDENT overcomes them. TRIDENT's novel primitive is its notion of k-holed assignment (Section 2.2), which can capture function calls (Section 2.3). Section 2.4 details TRIDENT's strategy for resisting overfitting.

2.1 The Challenges of Synthesis with Side Effects

The two key challenges of adding the insertion/modification of assignment statements into the TPR search space are (1) efficiency, i.e. how to efficiently search the extended space for plausible patches, and (2) test-overfitting, i.e. how to ensure that the generated patches are correct.

Consider the function `clamp` in Figure 1a; it restricts a number between two other numbers. It has a bug: it does not assign `r` to `x` before its checks. Suppose `clamp` fails

```
int buggy(int x, int y) {
    // missing call
        inc_if_zero(&x,&y)
    if (x > 0 && y > 0)
        return 1;
    else
        return 0;
}
```

(a) A function missing a function call.

```
void inc_if_zero(int &x, int &y) {
    if (x == 0)
        x++;
    if (y == 0)
        y++;
}
```

(b) The definition of `inc_if_zero`.

```
int buggy(int x, int y) {
    // Hole precedes suspicious code
    □1L, □2L = □1R, □2R;
    if (x > 0 && y > 0)
        return 1;
    else
        return 0;
}
```

(c) Assignment synthesis in TRIDENT.

Fig. 2: Synthesizing a patch that inserts an function call.

the test $\texttt{clamp}(2,1,4) \rightarrow 2$, where the " $\rightarrow 2$" denotes the expected output.

The first way to automatically repair this bug is *generate-and-validate* program repair: enumerate and test all possible insertions of assignments of the form $v = e$, where $v \in \{\texttt{x}, \texttt{l}, \texttt{h}, \texttt{r}\}$, the program variables in scope, and e is an expression over these variables. The main shortcoming of this method is that it requires a large number of test executions, and therefore does not scale to the large search spaces needed to repair realistic bugs. The second way to automatically repair this bug is *synthesis-based* program repair. Existing techniques, such as Angelix [9], do not synthesize assignments, because they are limited to side-effect-free expressions.

Since a test suite is an incomplete specification, TPR techniques are prone to test-overfitting, generating patches that pass the tests, but are incorrect. Extending the search space with side-effected patches poses additional challenges: (1) larger search spaces are more prone to test-overfitting [18], and (2) intuitively, changes with side effects are more likely to break functionality that is not covered by tests. A common approach to alleviate test-overfitting is to define a cost function on the search space, and search for a patch that passes the tests and minimises the cost. To the best of our knowledge, no cost function that takes side effects into account has been proposed.

2.2 Synthesising Assignments

To address the efficiency challenges that side-effected patches pose, we first introduce a new reification of a memory update that we call k-holed assignment. Then, we use this representation to define an efficient update-aware specification inference approach called multi-path specification inference.

A *lvalue-hole*, or $\square^L$, refers to a set of writable memory locations. The two-holed assignment $\square^L = \square^R$ combines an lvalue-hole and an rvalue-hole. Synthesising such assignment effectively means filling $\square^L$ with an lvalue (e.g. a variable), and $\square^R$ with an rvalue (e.g. an arithmetic expression). A *k-holed assignment* $\square_1^L, ..., \square_k^L = \square_1^R, ..., \square_k^R$ generalises two-holed assignment to the simultaneous assignment of up to k lvalue holes (Definition 6).

Following SemFix [12], TRIDENT inserts holes before suspicious locations (Figure 1b and Figure 2c), infers a logical specification for the holes, and then leverages the inferred specification to synthesise expressions to fill the holes. SemFix specification inference uses symbolic execution equipped to handle programs with rvalue-holes. Thus, a key prerequisite for supporting synthesis with side effects is extending specification inference to programs with k-holed assignments, as shown in Figure 1b.

A naïve approach to extend specification inference to k-holed assignments is to transform these assignments into switch statements as shown in Figure 1c for $k = 1$. Specifically, we can enumerate all possible variables that can appear on the left-hand side of an assignment as cases in a switch statement, and, for each case, insert an rvalue-hole for the right-hand expression. This transformation allows reusing SemFix' specification inference to synthesize an assignment statement. However, this approach is inefficient because adding a switch statement significantly increases the number of paths that symbolic execution must explore. Current general-purpose state-merging strategies do not efficiently handle the resulting path explosion because they require fixed state topology and do not have a mechanism for bounding the search of lvalues. Section 6.2 elaborates these limitations.

To alleviate path explosion of this naïve approach, we propose a more efficient semantics for inferences the specification of a k-holed assignment that we call *multi-path specification inference*. Multi-path specification inference rests on the insight that assignments to different variables along a path can leave that path's decisions unchanged. For example, inserting `x = l-1;` or `l = x+1;` at entry in Figure 1a both result in executing the true branch of first if-statement and the false branch of the second if-statement. Therefore, it is possible to *merge* the states induced by assignments to different variables, significantly reducing the number of paths to explore, thereby increasing the chance of finding a patch within a time budget. After merging states that correspond to assignments of different variables, the resulting path constraint effectively captures an equivalence class of assignments that drive test execution along this path.

To merge states corresponding to assignments of different variables, multi-path specification inference uses two groups of constraints: (1) those representing an assignment of a symbolic value to each writable memory location, controlled by a dedicated boolean selector variable, and (2) cardinality constraints over the selector variables that restrict the number of memory locations that a patch satisfying the specification assigns along a given path.

Consider the k-holed assignment statement $\square_1^L = \square_1^R, ..., \square_k^L = \square_k^R$ at the missing assignment in Figure 1b. Multi-path specification inference, when executing this

statement, constructs the following path constraint:

$$\begin{aligned}\phi \stackrel{\text{def}}{=} &(s_x \rightarrow x' = \alpha_x \wedge \neg s_x \rightarrow x' = x)\\ &\wedge (s_l \rightarrow l' = \alpha_l \ \ \wedge \neg s_l \rightarrow l' = l)\\ &\wedge (s_h \rightarrow h' = \alpha_h \wedge \neg s_h \rightarrow h' = h)\\ &\wedge (s_r \rightarrow r' = \alpha_r \ \wedge \neg s_r \rightarrow r' = r)\\ &\wedge \mathit{AtMost}(k, s_x, s_l, s_h, s_r)\end{aligned}$$

where, for $\beta \in \{x, l, h, r\}$, α_β is the symbolic variable representing values of the rvalue-hole $\square^R$, s_β is a boolean selector variable that enables/disables assignments to different memory locations, β' is the value of the program variable β after the statement, and $\mathit{AtMost}(k, -)$ is a cardinality constraint that encodes that at most k of its program variable arguments can be True. Here, $\mathit{AtMost}(k, -)$ means that at most k of the variables `x`, `l`, `h` and `r` can be modified as a result of executing the k-holed assignment.

TRIDENT's encoding merges the semantics of a subspace of assignments into the same state. Then, the values of the selector variables determine the subset of variables the synthesised statement updates. This significantly reduces the number of explored paths during symbolic execution. For example, consider a path that takes the false branches of the both if-statements that corresponds to the constraint $\psi \stackrel{\text{def}}{=} (x' \geq l') \wedge (r' \neq x' \vee x' \leq h')$. When TRIDENT explores this path in the program with k-holed assignments (Figure 1b), the path has the path condition $\phi \wedge \psi$. Lacking k-holed assignment, traditional symbolic execution would, to achieve an equivalent result, have to explore four paths in the program with switch statement (Figure 1c), yielding the path conditions $x' = \alpha_x \wedge \psi$, $l' = \alpha_l \wedge \psi$, $h' = \alpha_h \wedge \psi$ and $r' = \alpha_r \wedge \psi$. In this case, TRIDENT reduces the number of explored paths, for $k = 1$, from 4 to 1.

After exploring the path $\phi \wedge \psi$, TRIDENT can synthesize the patch `r = x`, since this patch is consistent with $\phi \wedge \psi$ if s_r is true, and produces the desired output $r' = 2$. To synthesize it, TRIDENT uses the variables {x, l, h, r} as both lvalue and rvalue components. TRIDENT automatically extracts the variables and data fields defined or used in the current function as components, as Section 3.5 details.

The cardinality constraint $\mathit{AtMost}(k, -)$ is essential for assignment synthesis. First, it allows synthesising statements that modify more than one memory location. Second, it reduces the search space by avoiding paths that are only feasible when more than k variables are modified. For example, consider `clamp` in Figure 1b and the test $\texttt{clamp}(2, 1, 4) \rightarrow 2$, which `clamp` fails returning 0, not 2. Using $\mathit{AtMost}(1, -)$ to restrict the search space to assignments that modify at most one variable makes the path that follows the false branch of the first if-statement and the true branch of the second if-statement infeasible. This is because changing evaluation of the second if-statements requires changing the binding of two variables, which $\mathit{AtMost}(1, -)$ forbids because it makes the following constraint unsatisfiable:

$$\begin{aligned}&(s_x \rightarrow x' = \alpha_x \wedge \neg s_x \rightarrow x' = 2)\\ \wedge\ &(s_l \rightarrow l' = \alpha_l \wedge \neg s_l \rightarrow l' = 1)\\ \wedge\ &(s_h \rightarrow h' = \alpha_h \wedge \neg s_h \rightarrow h' = 4)\\ \wedge\ &(s_r \rightarrow r' = \alpha_r \wedge \neg s_r \rightarrow r' = 0)\\ \wedge\ &\mathit{AtMost}(1, s_x, s_l, s_h, s_r)\\ \wedge\ &(x' \geq l')\\ \wedge\ &(r' = x' \wedge x' > h')\end{aligned}$$

We assume that k is relatively small, because large k implies that complex modifications are required and such programs have serious problems such as wrong algorithms or lacking functionality. TRIDENT is designed for program features that are almost correct with the exception of a small fragment of code. Currently, TRIDENT starts from $k = 1$. If it cannot synthesise a patch, it increments k and repeats, until it reaches a configurable bound on k.

2.3 Synthesising Function Calls

The abstraction provided by k-holed assignments is powerful enough to express more complex, side-effected modifications such as function calls. This is because loop-free functions are equivalent to simultaneous assignment, and program with loops can be summarised as simultaneous assignments using loop unrolling [15].

Consider the buggy program in Figure 2a with a missing call to the function `inc_if_zero` shown in Figure 2b. Assume `buggy` fails the test $\texttt{buggy}(0, 0) \rightarrow 1$.

TRIDENT computes function summaries for all functions that can be called at the target location. Currently, user must provide a library of functions, and TRIDENT computes the summaries via symbolic execution with loop unrolling. For example, TRIDENT computes the following summary for `inc_if_zero`:

$$\begin{aligned}&(x{=}0 \rightarrow x'{=}x{+}1 \wedge x{\neq}0 \rightarrow x'{=}x)\\ &(y{=}0 \rightarrow y'{=}y{+}1 \wedge y{\neq}0 \rightarrow y'{=}y)\end{aligned}$$

where x and y denote the values bound to the variables `x` and `y` before executing `inc_if_zero`, and x' and y' represent their values after executing it.

Given function summaries, TRIDENT executes `buggy` to infer its patch synthesis specification, as explained in Section 2.2. Here, we assume that TRIDENT infers this specification with $\mathit{AtMost}(k = 2, -)$, which corresponds to inferring a specification for lvalue-holes in the form of simultaneous assignment to at most 2 variables, as in Figure 2c. Specifically, TRIDENT infers this specification from the test-passing path:

$$\begin{aligned}&(s_x \rightarrow x' = \alpha_x \wedge \neg s_x \rightarrow x' = 0)\\ \wedge\ &(s_y \rightarrow y' = \alpha_y \wedge \neg s_y \rightarrow y' = 0)\\ \wedge\ &\mathit{AtMost}(2, s_x, s_y)\\ \wedge\ &(x' > 0 \wedge y' > 0)\end{aligned}$$

After inferring this patch synthesis specification, TRIDENT combines it with with `inc_if_zero`'s summary, if the two are consistent, to synthesise a call to `inc_if_zero(&x,&y)`, which replaces the pair of assignments in Figure 2c.

```
int f(int x, int y, bool ok)
{
        int r;
        r = x + 1;
        if (x < y)
            r = y + 1;
        if(ok > 0) return r;
        else return x+r;
}
```

(a) Low cost patch.

```
int f(int x, int y, bool ok)
{
        int r;
        r = x + 1;
        if (x < y)
            r = ++ok;
        if(ok > 0) return r;
        else return x+r;
}
```

(b) High cost patch.

Fig. 3: Patches with different cost.

2.4 Resisting Overfitting

As demonstrated in Section 5.3, the inclusion of patches with side effects into the program repair search space increases the probability of generating test-overfitting patches. To alleviate this problem, we propose a patch prioritization strategy that assigns lower cost to patches that have fewer side effects.

Consider the code fragments in Figure 3a and Figure 3b. The function $f(x, y, ok)$ is expected to return $\max(x, y) + 1$ if $(ok > 0)$, otherwise should return $\max(x, y) + x + 1$. Assume that TRIDENT generated patches that inserted the highlighted assignments to pass the test $\texttt{f}(-3, 1, 1) \rightarrow 2$. Although both programs pass the test, the patch in Figure 3b is incorrect, as it breaks the functionality for inputs satisfying $(y > x) \wedge (ok = 0)$.

To reduce the chance of generating such overfitted patches, we propose a heuristic to minimise the number of side effects in a patch. This heuristic is based on the intuition that updating fewer variables decreases the chance of breaking the functionality that is not covered by the tests. Specifically, the patch in Figure 3a is assigned cost 1, since it only changes r. The patch in Figure 3b is assigned cost 2, since it changes r and ok. Therefore, TRIDENT prefers the patch in Figure 3a due to its lower cost. Although this method does not guarantee that the chosen patch is correct, our evaluation in Section 5.3 shows that it does, in practice, alleviate test-overfitting associated with side effects.

3 TRIDENT

First, we formally define a programming language to describe our algorithms in Section 3.1. Then, we formalise the notion of side-effected patches in Section 3.2. Finally, we define the semantics of symbolic execution for our language in Section 3.3. The last three sections are devoted to the core contributions of TRIDENT. Section 3.4 describes the state-merging strategy for alleviating path explosion when inferring specification for patch synthesis. Section 3.5 defines a component based synthesis approach that explicitly reasons about side effects. Finally, Section 3.6 introduces a patch prioritization strategy for alleviating test-overfitting due to side effects.

3.1 The Programming Language $\mathcal{L}$

We consider an imperative programming language $\mathcal{L}$ whose syntax is described in Figure 4. *Program* $p \in \mathcal{L}$ is a set of function declarations, i.e. $\mathcal{L} \stackrel{\text{def}}{=} 2^{decl}$, with distinct names.

$$
\begin{array}{rcl}
\langle \text{stmt} \rangle & ::= & \langle \text{lvalue} \rangle = \langle \text{rvalue} \rangle \mid \langle \text{call} \rangle \mid \\
& & \texttt{if (}\langle \text{rvalue} \rangle\texttt{) \{ }\langle \text{stmt} \rangle\texttt{ \} else \{ }\langle \text{stmt} \rangle\texttt{ \} } \mid \\
& & \texttt{while (}\langle \text{rvalue} \rangle\texttt{) \{ }\langle \text{stmt} \rangle\texttt{ \} } \mid \\
& & \langle \text{stmt} \rangle\texttt{ ; }\langle \text{stmt} \rangle \\
& & \texttt{int }\langle \text{var} \rangle = \langle \text{rvalue} \rangle \\
& & \texttt{return }\langle \text{rvalue} \rangle \\
\langle \text{rvalue} \rangle & ::= & \langle \text{const} \rangle \mid \langle \text{var} \rangle \mid \texttt{*}\langle \text{var} \rangle \mid \texttt{\&}\langle \text{var} \rangle \mid \\
& & \langle \text{rvalue} \rangle + \langle \text{rvalue} \rangle \mid \textit{other operations...} \\
\langle \text{lvalue} \rangle & ::= & \langle \text{var} \rangle \mid \texttt{*}\langle \text{var} \rangle \\
\langle \text{rvlist} \rangle & ::= & \langle \text{rvalue} \rangle \mid \langle \text{rvalue} \rangle\texttt{ , }\langle \text{rvlist} \rangle \\
\langle \text{call} \rangle & ::= & \langle \text{fun} \rangle\texttt{ (}\langle \text{rvlist} \rangle\texttt{)} \\
\langle \text{vlist} \rangle & ::= & \langle \text{var} \rangle \mid \langle \text{var} \rangle\texttt{ , }\langle \text{vlist} \rangle \\
\langle \text{decl} \rangle & ::= & \texttt{int }\langle \text{fun} \rangle\texttt{(}\langle \text{vlist} \rangle\texttt{) \{ }\langle \text{stmt} \rangle\texttt{ \}}
\end{array}
$$

Fig. 4: Syntax of the programming language $\mathcal{L}$.

$\mathcal{L}$ is a subset of C with only integer values, without global variables or switch statements, among other simplifications.

$\mathcal{L}$ uses CPL's notion of value category[1] [19]. An expression in a program is an *rvalue* when it appears in a condition or on the right-hand side of an assignment or an argument of a function call, corresponding to the non-terminal $\langle$rvalue$\rangle$. An expression is an *lvalue* when it appears on the left-hand side of an assignment, which corresponds to the non-terminal $\langle$lvalue$\rangle$.

To avoid complicating the grammar, we assume that all expressions can be evaluated in "right-hand mode", but only certain expressions can be evaluated in "left-hand mode". For example, `x` can be both an lvalue and an rvalue, but `x+1` can only be an rvalue.

Memory is a function $\mu : \mathbb{N} \rightarrow \mathbb{Z}$ from addresses to values. The *stack frame* β is a finite subset of $\mathbb{N}$. An *environment* γ is a mapping from variable names to their addresses. The *program stack* σ is a stack of pairs $\{(\beta_i, \gamma_i)\}_i$ of the stack frame β_i and the environment γ_i, with the usual stack operations *pop* and *push*. $Dom(\gamma)$ indicates the set of visible program variables. Stack frame allocator is a procedure $newframe$ that, each time it is called, returns a new stack frame that is disjoint from previously allocated stack frames. The variable allocator $newloc$ is a procedure that, for a given stack frame, returns a location within this stack frame that is not allocated to any variable.

The semantics of a statement s in $\mathcal{L}$[2] is the relation $\langle s, \mu, \sigma \rangle \Downarrow \langle \mu', \sigma' \rangle$, where μ' and σ' are the memory and the stack obtained by executing the statement s in the context of memory μ and stack σ according to the semantics of the C language. Similarly, the semantics of an rvalue expression e in $\mathcal{L}$ is the relation $\langle e, \mu, \sigma \rangle \Downarrow \langle \mu', \sigma', r \rangle$, where r is the result of evaluating the expression.

$\mathcal{L}$ programs have C-like semantics, under several simplifying assumptions: all values are allocated on stack, stack

1. CPL is an an ancestor of C; we use its definition of value categories, as opposed to the more complex categories in C, to simplify the presentation.

2. We omit the, rather standard, formal definition of the semantics of a C-like language with pointers.

$$\text{NUM}\quad \frac{}{\langle n,\theta,\sigma,\pi\rangle \Downarrow_s \langle \theta,\sigma,\pi,n\rangle}$$

$$\text{OP}\quad \frac{\langle e_1,\theta,\sigma,\pi\rangle \Downarrow_s \langle \theta_1,\sigma_1,\pi_1,\phi_1\rangle \qquad \langle e_2,\theta_1,\sigma_1,\pi_1\rangle \Downarrow_s \langle \theta',\sigma',\pi',\phi_2\rangle \qquad \phi = \phi_1\ op\ \phi_2}{\langle e_1\ op\ e_2,\theta,\sigma,\pi\rangle \Downarrow_s \langle \theta',\sigma',\pi',\phi\rangle}$$

$$\text{RVALUE-VAR}\quad \frac{((\_,\gamma),\_) = \text{pop}(\sigma)}{\langle v,\theta,\sigma,\pi\rangle \Downarrow_s \langle \theta,\sigma,\pi,\theta(\gamma(v))\rangle}$$

$$\text{DEREF-RVALUE-VAR}\quad \frac{((\_,\gamma),\_) = \text{pop}(\sigma)}{\langle *v,\theta,\sigma,\pi\rangle \Downarrow_s \langle \theta,\sigma,\pi,\gamma(v)\rangle}$$

$$\text{ASSIGN}\quad \frac{\langle e,\theta,\sigma,\pi\rangle \Downarrow_s \langle \theta',\sigma',\pi',\phi\rangle \qquad ((\_,\gamma),\_) = \text{pop}(\sigma)}{\langle v = e,\theta,\sigma,\pi\rangle \Downarrow_s \langle \theta'[\gamma(v)\mapsto\phi],\sigma',\pi'\rangle}$$

$$\text{SYMB}\quad \frac{}{\langle \alpha,\theta,\sigma,\pi\rangle \Downarrow_s \langle \theta,\sigma,\pi,\alpha\rangle}$$

$$\text{IF-TRUE}\quad \frac{\langle e,\theta,\sigma,\pi\rangle \Downarrow_s \langle \theta_1,\sigma_1,\pi_1,\phi\rangle \qquad \pi_1\wedge\phi \text{ is SAT} \qquad \langle s_1,\theta_1,\sigma_1,\pi_1\wedge\phi\rangle \Downarrow_s \langle \theta',\sigma',\pi'\rangle}{\langle \texttt{if}\ (e)\ \{\ s_1\ \}\ \texttt{else}\ \{\ s_2\ \},\theta,\sigma,\pi\rangle \Downarrow_s \langle \theta',\sigma',\pi'\rangle}$$

$$\text{IF-FALSE}\quad \frac{\langle e,\theta,\sigma,\pi\rangle \Downarrow_s \langle \theta_1,\sigma_1,\pi_1,\phi\rangle \qquad \pi_1\wedge\neg\phi \text{ is SAT} \qquad \langle s_2,\theta_1,\sigma_1,\pi_1\wedge\neg\phi\rangle \Downarrow_s \langle \theta',\sigma',\pi'\rangle}{\langle \texttt{if}\ (e)\ \{\ s_1\ \}\ \texttt{else}\ \{\ s_2\ \},\theta,\sigma,\pi\rangle \Downarrow_s \langle \theta',\sigma',\pi'\rangle}$$

$$\text{WHILE-TRUE}\quad \frac{\langle e,\theta,\sigma,\pi\rangle \Downarrow_s \langle \theta_1,\sigma_1,\pi_1,\phi\rangle \qquad \pi_1\wedge\phi \text{ is SAT} \qquad \langle s,\theta_1,\sigma_1,\pi_1\wedge\phi\rangle \Downarrow_s \langle \theta_2,\sigma_2,\pi_2\rangle \qquad \langle \texttt{while}\ (e)\ \{\ s\ \},\theta_2,\sigma_2,\pi_2\rangle \Downarrow_s \langle \theta',\sigma',\pi'\rangle}{\langle \texttt{while}\ (e)\ \{\ s\ \},\theta,\sigma,\pi\rangle \Downarrow_s \langle \theta',\sigma',\pi'\rangle}$$

$$\text{WHILE-FALSE}\quad \frac{\langle e,\theta,\sigma,\pi\rangle \Downarrow_s \langle \theta_1,\sigma_1,\pi_1,\phi\rangle \qquad \pi_1\wedge\neg\phi \text{ is SAT}}{\langle \texttt{while}\ (e)\ \{\ s\ \},\theta,\sigma,\pi\rangle \Downarrow_s \langle \theta,\sigma,\pi\rangle}$$

$$\text{SEQ}\quad \frac{\langle s_1,\theta,\sigma,\pi\rangle \Downarrow_s \langle \theta_1,\sigma_1,\pi_1\rangle \qquad \langle s_2,\theta_1,\sigma_1,\pi_1\rangle \Downarrow_s \langle \theta',\sigma',\pi'\rangle}{\langle s_1\ ;\ s_2,\theta,\sigma,\pi\rangle \Downarrow_s \langle \theta',\sigma',\pi'\rangle}$$

$$\text{CALL}\quad \frac{\begin{array}{c}\langle e_1,\theta,\sigma,\pi\rangle \Downarrow_s \langle \theta_1,\sigma_1,\pi_1,\phi_1\rangle \qquad ... \qquad \langle e_n,\theta_{n-1},\sigma_{n-1},\pi_{n-1}\rangle \Downarrow_s \langle \theta_n,\sigma_n,\pi_n,\phi_n\rangle \qquad (\{v_1,...,v_m\},s) = \text{decl}(f)\\ \beta = \text{newframe}() \qquad \gamma = \{v_i\mapsto \text{newloc}(\beta)\} \qquad \sigma' = \text{push}(\sigma_n,(\beta,\gamma)) \qquad \langle s,\theta_n,\sigma',\pi_n\rangle \Downarrow_s \langle \theta',\sigma'',\pi'\rangle \qquad ((\_,\gamma'),\_) = \text{pop}(\sigma'')\end{array}}{\langle f(e_1,...,e_n),\theta,\sigma,\pi\rangle \Downarrow_s \langle \theta',\sigma_n,\pi',\theta(\gamma'(\text{return}))\rangle}$$

$$\text{VAR-DECL}\quad \frac{\langle e,\theta,\sigma,\pi\rangle \Downarrow_s \langle \theta',\sigma',\pi',\phi\rangle \qquad ((\beta,\gamma),t) = \text{pop}(\sigma') \qquad \gamma' = \gamma[v\mapsto \text{newloc}(\beta)] \qquad \sigma'' = \text{push}(t,(\beta,\gamma'))}{\langle \texttt{int}\ v = e,\theta,\sigma,\pi\rangle \Downarrow_s \langle \theta',\sigma'',\pi'\rangle}$$

Fig. 5: Semantics of symbolic execution.

$$\text{KHOLE-ASSIGN}\quad \frac{\begin{array}{c}((\_,\gamma),\_) = \text{pop}(\sigma) \qquad n = |\text{Dom}(\gamma)| \qquad s_1,...,s_n = \text{fresh bool symb. variables} \qquad \alpha_1,...,\alpha_n = \text{fresh int symb. variables}\\ \theta' = \theta[\gamma(v_i)\mapsto\alpha_i]_{v_i\in\text{Dom}(\gamma)} \qquad \pi' = \pi\wedge(\bigwedge_{v_i\in\text{Dom}(\gamma)} \neg s_i\to\alpha_i = \theta(\gamma(v_i)))\wedge AtMost(k,\{s_1,...,s_n\})\end{array}}{\langle \square_1^L,...,\square_k^L = \square_1^R,...,\square_k^R,\theta,\sigma,\pi\rangle \Downarrow_m \langle \theta',\sigma,\pi'\rangle}$$

Fig. 6: Semantics of multi-path specification inference.

frames have infinite size and never de-allocated, variables have dynamic scope, and the entry function and its arguments have to be specified explicitly. Specifically, executing an $\mathcal{L}$ program means evaluating the entry function applied to its arguments in the context of zeroed memory and an empty stack, as stated in the definition below:

Definition 1 (Execution)**.** *Let $p \in \mathcal{L}$ be a program, f be p's entry function, $A \stackrel{def}{=} [a_1,...,a_n]$ be an ordered set of integers (entry function arguments).* Execution *is the function exec defined as $exec(p,f,A) \stackrel{def}{=} (\mu,r)$ such that $\langle f(a_1,...,a_n),\lambda x.\ 0,\emptyset\rangle \Downarrow \langle \mu,\_,r\rangle$.*

TRIDENT uses tests as correctness criteria; a *test* is a pair $([a_1,...,a_n],\phi)$, where $a_1,...,a_n$ is a sequence of integer inputs, and ϕ is a predicate on the return value. A program p with the entry function f passes the test $([a_1,...,a_n],\phi)$ iff $\phi(r)$ where $(\_,r) \stackrel{\text{def}}{=} exec(p,f,[a_1,...,a_n])$.

3.2 Patches with Side Effects

Any modification that changes the behaviour of the program can be in principle considered as a patch with side effects. Thus, to make a more fine-grained distinction between side-effect-free and side-effected patches, we over-approximate side effects syntactically. Specifically, we consider a patch with side effects if any individual part of its diff explicitly updates memory.

A *patch* is a pair of programs (p,p'). The *difference* between p and p' — *diff*(p,p') — is the minimal (in terms of the number of AST nodes) substitution $\{\ s_1^p \mapsto s_1^{p'},...,s_n^p \mapsto s_n^{p'}\ \}$ of statements/expressions in p to statements/expressions in p' such that p' is obtained by simultaneously applying this substitution to p (we assume that s_i^p and $s_i^{p'}$ are all unique statements at different program locations). We call a pair $(s_i^p, s_i^{p'})$ in this mapping an *atomic substitution*. An atomic substitution has a side effect iff there exists a memory, stack pair that defines a context in which the execution of p and p' results in a different value for at least one memory location.

Definition 2 (Atomic Substitution with Side Effect)**.** *An atomic substitution $(s_i^p, s_i^{p'})$ has a side effect iff there exists an address $a \in \mathbb{N}$, a memory μ, and a stack σ such that $\langle s_i^p,\mu,\sigma\rangle \Downarrow \langle \mu_1,\_,\_\rangle$, $\langle s_i^{p'},\mu,\sigma\rangle \Downarrow \langle \mu_2,\_,\_\rangle$ and $\mu_1(a) \neq \mu_2(a)$.*

This definition considers side effects syntactically *w.r.t.* the *minimal* difference between a program and a patched

version under a given *diff* algorithm.

Definition 3 (Patch with Side Effect)**.** *A patch has a side effect iff an atomic substitution of its difference has a side effect.*

For example, consider a patch

$$(int\ f(y)\ \{\ x = y - 1\}, int\ f(y)\ \{\ x = y + 1\}).$$

The difference of this patch is the minimal substitution $\{y - 1 \mapsto y + 1\}$. The expressions $y - 1$ and $y + 1$ do not write to memory, so their evaluation results in the same memory values for any initial memory and stack. Therefore, this patch has no side effects.

Since our simplified language does not use heap and variables are never de-allocated, any call to a function with assignments is considered to be an expression with side effects. However, this definition can be trivially relaxed for languages with heap and stack frame de-allocations if we define side effects *w.r.t.* to the heap and the stack rather than the entire memory.

3.3 Symbolic Execution of $\mathcal{L}$

TRIDENT relies on SMT solving and symbolic execution. As is usual in SMT literature [20], we consider formulas and terms built from predicate and function symbols (e.g. " $+$ ", " $-$ ", " $>$ ") from a given signature Σ. We denote the set of all such formulas and terms as L_Σ. We also consider a background theory $\mathcal{T}$ that fixes the interpretations of the symbols in Σ.

We use the letters α, β, γ and δ to denote variables from L_Σ, and the letters π, ϕ and ψ to designate formulas from L_Σ. *Symbolic memory* θ is a function from memory addresses to logical terms from L_Σ (for an address a, the corresponding logical term is $\theta(a)$). We express the equality of two symbolic program states θ_1 and θ_2 for all initialised addresses as the formula $\theta_1 = \theta_2 \stackrel{\text{def}}{=} \bigwedge_{v \in Initialised} \theta_1(v) = \theta_2(v)$.

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 $True$.

The semantics of symbolic evaluation is defined as evaluation that transforms symbolic memory and augments current path constraint:

Definition 4 (Semantics of symbolic evaluation)**.** *The semantics of symbolic evaluation of a statement s in $\mathcal{L}$ is the relation $\langle s, \theta, \sigma, \pi\rangle \Downarrow_s \langle \theta', \sigma', \pi'\rangle$, where θ', σ' and Π' are the symbolic memory, the stack and the path condition obtained by executing the statement s in the context of symbolic memory μ, stack σ, and path condition π according to the semantics of symbolic execution in Figure 5.*

The semantics of symbolic evaluation of rvalue expressions is defined accordingly.

Definition 5 (Symbolic execution)**.** *Let $p \in \mathcal{L}$ be a program, f be a function declared in p (entry function), $\Phi \stackrel{def}{=} [\phi_1, ..., \phi_n]$ be an ordered set of terms from L_Σ (entry function arguments). The function $symex$ is defined as $symex(p, f, A) \stackrel{def}{=} \{(\pi_i, \theta_i, \psi_i)\}_i$ (set of triples) such that, for all i, $\langle f(\phi_1, ..., \phi_n), \lambda x.\ 0, \emptyset, True\rangle \Downarrow_s \langle \theta_i, \_, \pi_i, \psi_i\rangle$.*

3.4 TRIDENT's Multi-Path Specification Inference

A key part of TRIDENT repair algorithm is inferring specification for patch synthesis. This specification is a logical formula that summarises how changes at the given location can affect the output of the program. The main difference between TRIDENT and previous semantic algorithms, such as Angelix, is that TRIDENT's specification encodes the effect of assigning multiple memory locations. A naïve way to implement specification inference shown in Figure 1c causes path explosion and therefore reduces the scalability of program repair as demonstrated in Section 5. Thus, we propose a multi-path specification inference approach that uses a specialised state merging strategy to reduce the number of explored paths.

In order to formalise multi-path specification inference, we extend language $\mathcal{L}$ with a k-holed assignment statement defined below:

Definition 6 (k-holed assignment)**.** *Let k, l be integers such that $l \leq k$, $\square_1^L, ..., \square_k^L$ be lvalue-holes, $\square_1^R, ..., \square_k^R$ be rvalue-holes, $x_1, ..., x_l$ be a sequence of lvalue expressions, $e_1, ..., e_l$ be a sequence of rvalue expressions. The semantics of k-*holed assignment *$\square_1^L, ..., \square_k^L = \square_1^R, ..., \square_k^R$ w.r.t. the substitution $x_1, ..., x_l$, $e_1, ..., e_l$ is defined as*

$$\begin{aligned}
&t_1 = x_1\\
&...\\
&t_l = x_l\\
&x_1 = e_1[x_1 \mapsto t_1, ..., x_l \mapsto t_l]\\
&...\\
&x_l = e_l[x_1 \mapsto t_1, ..., x_l \mapsto t_l]
\end{aligned}$$

where $t_1, ..., t_l$ are fresh variables, and $e_i[x_1 \mapsto t_1, ..., x_l \mapsto t_l]$ is the expression obtained by replacing $x_1, ..., x_l$ with $t_1, ..., t_l$ correspondingly.

The fresh variables $t_1, ..., t_l$ in the definition prevent the preceding assignments from affecting the result of the following ones. We refer to this extended language with k-holed assignments as $\mathcal{L}'$.

Multi-path specification inference is defined for programs from $\mathcal{L}'$ using an augmented version of symbolic execution. For simplicity, we assume that there is only a single test, but all definitions in this paper can be trivially generalised for multiple tests by considering the conjunction of constraints corresponding to these tests.

Definition 7 (Multi-path specification inference)**.** *Let $p \in \mathcal{L}'$ be a program, f be a function declared in p (entry function), $([a_1, ..., a_n], \phi)$ be a test. The function $infer$ is defined as $infer(p, f, [a_1, ..., a_n]) \stackrel{def}{=} symex_m(p, f, [a_1, ..., a_n])$ where $symex_m$ is given in Definition 5 with the semantics $\Downarrow_m$, where the relation $\Downarrow_m$ is an extension of $\Downarrow_s$ to $\mathcal{L}'$ in Figure 6.*

3.5 TRIDENT's Patch Synthesis

TRIDENT employs component-based program synthesis that constructs a patch as a combination of components that satisfies a given logical specification. The key novelty relative to previous techniques [13], [14] is that it supports side-effected components. Specifically, we consider components of three types identified as sets of labels: $\{R\}$

Algorithm 1: Patch synthesis

Input:
$Components$: A list of components
S: inferred specification

for $tree \in enumerate\_trees(Components)$ **do**
  ϕ = $encode(tree, S)$
  $is\_sat, valuation = solve(\phi)$
  if *is_sat* **then**
    return $decode(tree, valuation)$

Algorithm 2: Patch Prioritization

Input:
$Patch$: A list of components
$Line$: Line number of patch
P: Program

if $has\_no\_side\_effects(Patch)$ **then**
  return 0
$P' := apply\_patch(P, Patch)$
return $mutated\_memory\_count(P', Line)$

for components that represent rvalue-expression, $\{L\}$ for components that represent lvalue expressions, and $\{R, L\}$ for components that are both rvalue expressions and lvalue expressions, such as program variables.

Definition 8 (Component)**.** *Component is a tuple* $(T, \{i_1^R, ..., i_n^R\}, \{i_1^L, ..., i_m^L\}, \phi)$*, where* T *is the type of the component (i.e.* $\{R\}$*,* $\{L\}$*, or* $\{R, L\}$*),* $\{i_1^R, ..., i_n^R\}$ *is the set of rvalue inputs,* $\{i_1^L, ..., i_m^L\}$ *is the set of lvalue inputs, and* ϕ *is the semantics of the component, a logical formula over the variables* $\{i_1^R, ..., i_n^R, i_1^L, ..., i_m^L\}$*, representing the input, and* $\{o^t\}_{t \in T}$*, representing the outputs.*

For example, a component that adds one to a given value can be represented as follows

$$(R, \{i_1^R\}, \{\}, o^R = i_1^R + 1),$$

because it is component that has one rvalue input and is itself an rvalue. Meanwhile, a component that increments a variable, is represented as

$$(R, \{i_1^R\}, \{i_1^L\}, o^R = i_1^L \wedge i_1^L = i_1^R + 1),$$

because it accepts an argument as both an rvalue (for reading) and an lvalue (for writing), updates the value of the lvalue input, and returns this value as rvalue output.

TRIDENT represents patches as trees of components. Specifically, a component tree is a pair $(c, \{i_1^R \mapsto t_1^R, ..., i_n^R \mapsto t_n^R, i_1^L \mapsto t_1^L, ..., i_m^L \mapsto t_m^L\})$ of the root component c and a mapping from its inputs to subtrees $t_1^R, ..., t_n^R, t_1^L, ..., t_m^L$. The semantics of the tree is a formula that connects input and outputs of the components:

$$[\![(c, \{i_1^R \mapsto t_1^R, ..., i_n^R \mapsto t_n^R, i_1^L \mapsto t_1^L, ..., i_m^L \mapsto t_m^L\})]\!] \stackrel{\text{def}}{=}$$
$$\phi \wedge i_1^R = o_1^R \wedge ... \wedge i_n^R = o_n^R \wedge i_1^L = o_1^L \wedge ... \wedge i_m^L = o_m^L$$
$$\wedge [\![t_1^R]\!] \wedge ... \wedge [\![t_n^R]\!] \wedge [\![t_1^L]\!] \wedge ... \wedge [\![t_m^L]\!],$$

where ϕ is the semantics of the component c and $o_1^R, ...$ are the outputs of the root components of the subtrees $t_1^R, ...$

TRIDENT uses an enumerative algorithm to find a patch that satisfies the given specification. However, an enumerative algorithm is not efficient for generating integer constants because of search space explosion. To address this, TRIDENT applies an SMT solver to generate these constants, similarly to SyGuS solvers [14]. For example, to represent a set of components that add different constants to a given value, instead of considering concrete semantics $o^R = i_1^R + 1$, $o^R = i_1^R + 2, ...$, we consider an abstract semantics $o^R = i_1^R + c$ and ask an SMT solver to find a value of the parameter c that satisfies the specification.

Algorithm 1 demonstrates the patch synthesis algorithm that combines enumeration and SMT-solving. First, this algorithm enumerates abstract trees, that is, trees of components in which the leafs corresponding to constants are represented as parameters. Then, it encodes each abstract tree and the specification into a formula that is satisfiable iff there is a assignment of constants that makes the patch satisfy the specification. Particularly, for a program p with a k-holed assignment and the entry function f, a test $([a_1, ..., a_n], \phi)$, a specification $\{(\pi_i, \psi_i)\}_i \stackrel{\text{def}}{=} infer(p, f, [a_1, ..., a_n])$, and a candidate components tree t, it constructs the following formula:

$$[\![t]\!] \wedge \bigvee_i \pi_i \wedge \phi(\psi_i)$$

For multiple tests, TRIDENT considers the conjunction of the corresponding formulas. If the formula is satisfiable, then TRIDENT constructs a concrete patch from the model by substituting constant parameters with concrete values.

3.6 TRIDENT's Patch Prioritization

TRIDENT employs patch prioritization strategy to alleviate test-overfitting. TRIDENT prioritizes patches based on the assumption that minimising the number of side effects in the patched expression will reduce overfitting. Algorithm 2 returns the patch priority where the patches with lower priority value are preferred by the patch prioritisation strategy. The algorithm takes as input the program, patch and the line number where the patch is applied. The function $apply\_patch$ applies the patch on program P. The function $mutated\_memory\_count$ returns the number of memory locations whose values are changed by the execution of the patched line $Line$ in the program P'. If the patch P has no side effects under Definition 3, then algorithm 2 gives it a priority of 0, to avoid synthesising patches with side effects when they are not required.

4 IMPLEMENTATION

Figure 7 shows the architecture of TRIDENT, which consists of three main components:

- The frontend transforms and analyses buggy programs;
- The inference engine infers synthesis specifications; and

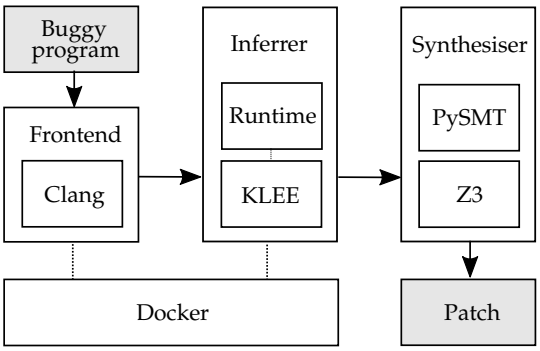


Fig. 7: Architecture of TRIDENT.

- The synthesiser constructs patches.

The frontend performs several source code focused tasks. First, it localises suspicious locations in the buggy program using Ochiai statistical fault localisation [21]. Second, it instruments suspicious locations by inserting holes in the form of calls to the function `__trident_khole_assignent` using Clang, LLVM's default frontend [22]. Finally, it calls the other components of TRIDENT to infer synthesis specification and synthesize patches.

The inference engine is built on top of KLEE symbolic execution engine [23] and extensions implemented as a C library that is linked to the buggy program when it is executed using KLEE. The runtime provides a function `__trident_khole_assignent` that represents a k-holed assignment $\square_1^L, ..., \square_k^L = \square_1^R, ..., \square_k^R$. This function take the values of program variables and addresses of assignable memory locations, and constructs path constraints and symbolic state according to the semantics of the rule KHOLE-ASSIGN in Figure 6.

The synthesizer constructs patches based on inferred specification in the form of KLEE path conditions and provided components and function summaries. It uses PySMT [24] to manipulate SMT formulas, and Z3 for constraint solving.

TRIDENT relies on Docker [25] virtual environments to execute the subject program for fault localization and patch validation, and to perform symbolic execution with KLEE.

The following transformation schemas for C programs, which adapt transformation schema successfully used in previous work [9], define TRIDENT's the search space:

$$\langle \text{stmt} \rangle \texttt{ ; } \mapsto \langle \text{stmt} \rangle \texttt{ ; } \square_1^L, ..., \square_n^L = \square_1^R, ..., \square_n^R \texttt{ ;}$$
$$\texttt{if } (\langle \text{expr} \rangle) \ \langle \text{stmt} \rangle \mapsto c, \square_2^L, ..., \square_n^L = \square_1^R, ..., \square_n^R \texttt{ ; if } (c) \ \langle \text{stmt} \rangle$$
$$\texttt{while } (\langle \text{expr} \rangle) \ \langle \text{stmt} \rangle \mapsto \texttt{while } (c) \ \{ \ \langle \text{stmt} \rangle \texttt{ ; } c, \square_2^L, ..., \square_n^L = \square_1^R, ..., \square_n^R \}$$
$$\texttt{for}(\_; \langle \text{expr} \rangle; \_) \ \langle \text{stmt} \rangle \mapsto \texttt{for } (\_; c; \_) \{ \langle \text{stmt} \rangle \texttt{ ; } c, \square_2^L, ..., \square_n^L = \square_1^R, ..., \square_n^R \ \}$$
$$\texttt{switch } (\langle \textit{expr} \rangle) \ \langle \text{stmt} \rangle \mapsto c, \square_2^L, ..., \square_n^L = \square_1^R, ..., \square_n^R \texttt{ ; switch } (c) \ \langle \text{stmt} \rangle$$
$$\langle \text{call} \rangle \texttt{ ; } \mapsto \square_1^L, ..., \square_n^L = \square_1^R, ..., \square_n^R \texttt{ ;}$$
$$\langle \text{assignment} \rangle \texttt{ ; } \mapsto \square_1^L, ..., \square_n^L = \square_1^R, ..., \square_n^R \texttt{ ;}$$

In these transformations, we synthesize a k-holed assignments $c, \square_2^L, ..., \square_k^L = \square_1^R, ..., \square_k^R$ with a dedicated variable c that is used as an rvalue expressions for conditional statements, loops and switch statements. Effectively, this emulates synthesis of expressions with side effects.

TABLE 1: OSS10 dataset of bugs from open source projects

Program	Commit	kLoC	Missing/incorrect statements
grep	191a84a	38	`match_lines=match_words=0;`
grep	7585d81	27	`strip_trailing_slashes(optarg);`
coreutils	c160afe	208	`x.preserve_xattr = true;`
coreutils	9944e47	249	`if (!nfiles) fstatus[0].failed=1;`
coreutils	ca99c52	249	`if (line_width<0) line_width=0;`
coreutils	9f5aa48	224	`relative_to = relative_base;`
coreutils	2a80912	247	`f[i].fd = -1;`
busybox	0506e29	297	`end = key->range[3];`
busybox	5c13ab4	331	`flags = option_mask32;`
busybox	6f7a009	351	`xtc &= ~TC_UOPPOST;`

TABLE 2: CF110 dataset of bugs from Codeflaws benchmark

Class	Number	Description
SISA	38	Insert assignment
DRWV	14	Replace variable with variable
DMAA	3	Insert/replace array access
ORRN	21	Replace relational operator
OILN	15	Tighten/loosen condition
OAIS	19	Insert/delete arithmetic operator
Total	110	

4.1 Limitations

TRIDENT inherits the usual limitations of symbolic execution approaches. First, it faces the usual path explosion problem, which the state merging algorithm, described in Section 3.4, alleviates. Second, state-of-the-art symbolic execution engines cannot automatically model the environment, like network communication. Third, SMT solver are necessarily incomplete and cannot solve all expressions in actual programs. Finally, handling pointers and data structures is an open challenge for symbolic execution.

Aliasing occurs when the same data can be accessed through different pointers. Our synthesis algorithm assumes that there is no aliasing in the considered L-values. Specifically, all L-values passed to the synthesizer must refer to different memory locations in the context of given tests. TRIDENT's implementation trivially guarantees this property since it only passes references to local and global variables as L-values. To support dynamic data structures, such as linked lists, an alias analysis technique can be applied. Supporting dynamic data structures also requires modelling them symbolically. TRIDENT's implementation does not support dynamic data structures, because KLEE does not explicitly model them.

5 EVALUATION

To evaluate TRIDENT, we first demonstrate its utility: we show that TRIDENT can synthesise patches with side-effects for bugs in real-world programs. Synthesising side-effected patches exacerbates two seminal challenges: the path explosion problem of symbolic execution and the test-overfitting problem of program repair. TRIDENT combats path explosion with multi-path specification inference introduced in Section 3.4. Section 5.2 reports the effectiveness of this countermeasure. Section 5.3 shows how much more prone to test-overfitting patches with side effects are than side-effect-free patches, and how well TRIDENT's prioritisation heuristic alleviates this problem.

Benchmarks: To answer our research questions, we constructed three bug datasets:

- **OSS10** 10 bugs from open source projects that require patches containing addition/modification of assignments and function calls, extracted from Coreutils, Grep and Busybox (Table 1);
- **CF110** 110 bugs from Codeflaws [17]. Among these 110 version, 55 require patches with side effects, and 55 require patches without side effects.
- **MB37** 37 bugs taken from ManyBugs [16].

To construct OSS10, we identified bugs from Coreutils, Grep and Busybox that require patches with side effects. We chose these projects for our dataset, because they are well-supported by KLEE, and also their version control history links bug fixing commits with corresponding regression tests. Specifically, we uniformly sampled bugs from these projects, keeping the first 10 that manual assessment determined involved statements with side effects, those that add/modify assignments or function call. Table 1 lists the size of the codebase from which each bug is drawn.

ManyBugs [16] benchmark consists of 185 defects taken from nine large, open source C projects. This benchmark is commonly used in evaluating automatic program repair tools [9], [26], [27]. Prior work [28] argues for explicitly defining the defect classes while evaluating various program repair tools, to ensure a fair comparison of tools on comparable classes. A defect class is a family of bugs that share a common attribute. For instance, GenProg [4] does not repair expression-level bugs, while Angelix [9] does not fix bugs pertaining to insertion of new statements or modifying existing statements in a way that induces side effects. TRIDENT supports the defect classes of Angelix[3] while additionally supporting those defect classes with side effects that k-holed assignment can model. Following previous work [27], we eliminated the samples that do not belong to TRIDENT's defect classes or that TRIDENT could not compiled due to version incompatibilities; this led to MB37, which is a dataset of 37 samples.

We chose Codeflaws as the source for our second dataset, because it contains bugs from a large variety of defect classes. Codeflaws bugs were not labeled with side effects in mind. To construct CF110, we therefore inspected bugs in defect classes that require, or rule out, patches with side effects, and then uniformly sampled bugs from the inspected set. Specifically, we selected 55 bugs from classes SISA, DRWV and DMAA (with side effects), and 55 bugs from classes ORRN, OILN and OAIS (without side effects). Section 6 details the reasons for constructing new datasets.

Tool Configurations: In our experiments, we use the following tool configurations:

- **TN** TRIDENT with disabled patch prioritisation;
- **TP** TRIDENT with enabled patch prioritisation;
- **PR** Prophet [26] with default configuration;
- **SOS** SOSRepair [27] with default configuration;
- **AKN** Angelix-like assignment synthesis with disabled KLEE merging;

3. Although TRIDENT supports Angelix' defect class, their search spaces are different: Angelix attempts to minimally modify existing expressions, while TRIDENT synthesises expressions from scratch. This explains differences in the results.

```
int clamp(int x, int l, int h) {
    int r;
    klee_open_merge()
    switch(PATCH_ID) {
        case 0: x = □^R; break;
        case 1: l = □^R; break;
        case 2: h = □^R; break;
        case 3: r = □^R; break; }
    klee_close_merge()
    if (x < l)
        r = l;
    if (r == x && x > h)
        r = h;
  return r;
}
```

Fig. 8: Applying KLEE state merging in AKP.

- **AKP** Angelix-like assignment synthesis with enabled KLEE merging;
- **ANG** Angelix [9] with default configuration; and
- **GP** GenProg [4] with default configuration.

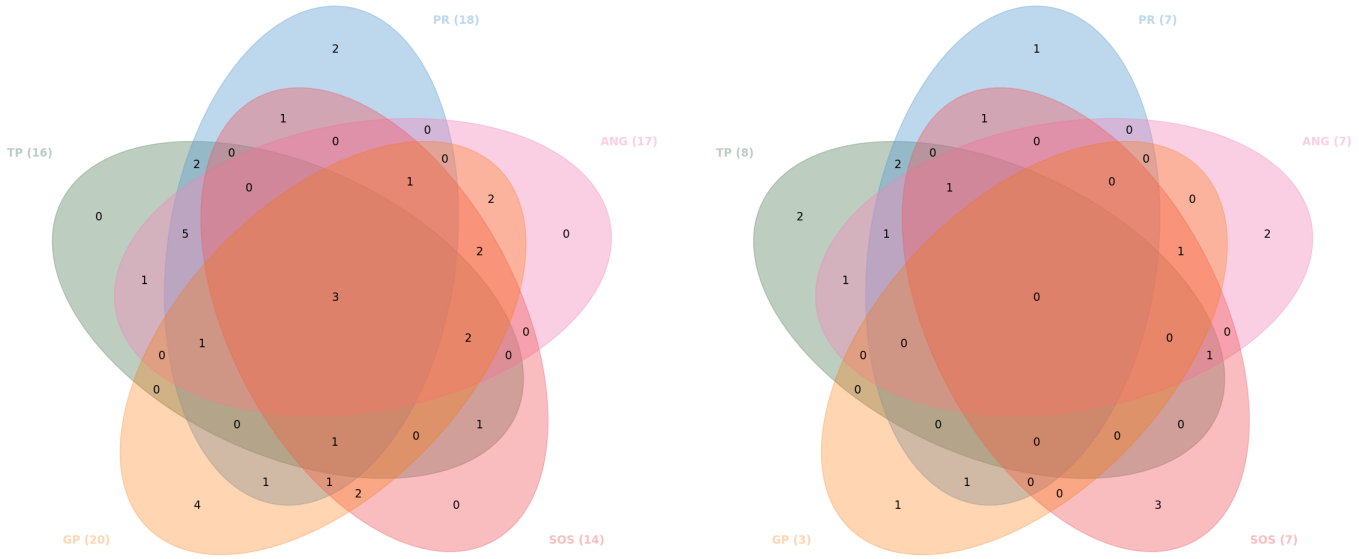
The TP and TN configurations employ multi-path specification inference described in Section 3.4 with cardinality constraints with $K = 2$. AKN is an application of Angelix for assignment synthesis that uses a switch statement to enumerate possibly writable memory locations (Section 2.2). AKP is an extension of AKN that applies KLEE's built-in state-merging mechanism by surrounding the switch statement with `klee_open_merge()` and `klee_close_merge()` as shown in Figure 8. ANG is Angelix [9] applied only to side-effect-free expressions, but re-implemented using the same synthesiser as TRIDENT with only rvalue components. We used GenProg [4] in our experiments because, although it does not synthesise patches with side effects, it can potentially generate them by copying from other parts of the same program. PR is the original version of Prophet [26] with the default configuration specified in their replication package. For SOS, GP, and ANG we used the generated patches listed in their replication packages to compile the results.

Experimental Setup: We conducted all experiments inside a Docker container on an Intel® Core™ i7-2600 CPU 2.7 GHz machine running on Ubuntu 16.04 with 16GB of memory. We used 2 hours as the timeout for each tool.

5.1 TRIDENT Fixes Real Bugs

To investigate the applicability of TRIDENT for realistic projects, we ran it on the real bugs in the OSS10 and MB37 datasets. On OSS10 that contain bugs involving side effects, we executed three configurations: TP, AKP and GP. Here AKP serves as the baseline approach. GP is an alternative approach, because it cannot synthesise new statements, but can copy/move them from a code bank, by default the rest of the same program. On OSS10 that contain both bugs involving side effects and side-effect-free bugs, apart from TP, we executed four configurations that represent state-of-the-art C program repair tools: ANG, PR, SOS and GP.

In order to identify if the generated patches are correct, we conservatively compared them with human patches, classifying a patch as correct only if it is syntactically identical to the human patch, or can be obtained from the human patch through a trivial refactoring.



(a) The number of plausible patches generated by each tool.

(b) The number of correct patches generated by each tool.

Fig. 9: Venn diagrams describing the intersection of the repaired bugs in MB37 dataset by different repair tools. Generally, the tools are complementary. TRIDENT correctly fixes 2 bugs that other tools do not repair correctly.

```
...
  case EXCLUDE_DIRECTORY_OPTION:
    if (!excluded_directory_patterns)
      excluded_directory_patterns = new_exclude ();
      strip_trailing_slashes (optarg);
      add_exclude (excluded_directory_patterns, optarg,
           EXCLUDE_WILDCARDS);
      break;
...
```

(a) Function call generated by TRIDENT for Grep.

```
...
  if (!TIFFFillStrip(tif,strip))
    return((tmsize_t)(-1));
  size = strip_size;

  if ((*tif->tif_decodestrip)(tif,buf,size,plane)<=0)
    return((tmsize_t)(-1));
  (*tif->tif_postdecode)(tif,buf,size);
  return(size);
...
```

(b) Assignment statement generated by TRIDENT for Libtiff.

Fig. 10: Examples of patches generated by TRIDENT.

TABLE 3: Generated patches for OSS10: ● indicates correct patch, ◐ — plausible patch, ○ — no patch found.

Bug	Patch			Time (s)		
	TP	AKP	GP	TP	AKP	GP
191a84a	○	○	○	1273.2	Timeout	Timeout
7585d81	●	●	○	469.8	554.2	Timeout
c160afe	●	○	○	94.6	Timeout	Timeout
9944e47	◐	○	○	76.3	Timeout	Timeout
ca99c52	○	○	○	Timeout	Timeout	Timeout
9f5aa48	○	○	○	Timeout	Timeout	Timeout
2a80912	●	○	○	75.6	Timeout	Timeout
0506e29	○	○	○	Timeout	Timeout	Timeout
5c13ab4	●	●	○	367.9	348.3	Timeout
6f7a009	◐	◐	○	283.4	349.6	Timeout
Overall	4+2	2+1	0+0			

Table 3 summarises the results of our experiment on OSS10. The time column indicates the time(in seconds) taken to generate a patch for the corresponding buggy location. TP generated more patches than AKP and GP on the considered dataset. TP generated more patches than AKP due to the efficiency of its state merging algorithm. GP could not generate patches for the considered bugs, because required statements are not present in the source code. TP repaired all versions, except for Grep 7585d81, using one or more assignments. For Grep 7585d81, TP generated a function call shown in Figure 10a, which is identical to the human patch. In order to enable this function call synthesis, we first generated summaries for all supported functions in Grep, and used them as components for patch synthesis.

Table 4 summarises the results of our experiments on MB37 dataset: TRIDENT's performance is comparable to other state of the art tools. Figure 9 shows the overlap of generated patches for different tools. TRIDENT generates 2 correct patches that no other tool could repair, Figure 9b. Figure 10b shows one of these two patches.

Even though TRIDENT supports the defect classes of Angelix, we can see from Figure 9b and Figure 9a that Angelix synthesises some bugs that TRIDENT cannot and vice-versa. This is due to the difference in their search spaces: Angelix attempts to minimally modify existing expressions, whereas TRIDENT synthesises expressions from scratch. One illustrative example is Figure 11. Here, Angelix successfully generates a patch, since it is easy to modify the existing expression to reach the patch by simply dropping the expression `tif->tif_rawcc != orig_rawcc`.

TABLE 4: The number of plausible and correct patches each program repair tool generates on bugs in MB37 dataset. TRIDENT generates 16 out of 37 patches of which 8 are equivalent to patches written by the developers.

Bug	kLoC	Total	Plausible					Correct				
			TN	ANG	PR	SOS	GP	TN	ANG	PR	SOS	GP
gmp	145	2	1	2	1	0	1	0	0	0	0	0
gzip	491	4	3	3	2	0	1	2	1	2	0	0
libtiff	77	10	5	5	5	8	7	3	3	1	2	2
php	1,099	19	6	6	9	4	9	3	3	4	3	0
wireshark	2,814	2	1	1	1	2	2	0	0	0	2	0
Overall		37	16	17	18	14	20	8	7	7	7	2

```
* by the compression close+cleanup routines.  But
* be careful not to write stuff if we didn't add data
* in the previous steps as the "rawcc" data may well be
* a previously read tile/strip in mixed read/write mode.
*/
-  if (tif->tif_rawcc > 0 && tif->tif_rawcc != orig_rawcc
+  if ((tif->tif_rawcc > 0)
&& (tif->tif_flags & TIFF_BEENWRITING) != 0
&& !TIFFFlushData1(tif))
{
```

Fig. 11: Angelix's patch for libtiff-2007-11-02-371336d-865f7b2.

TRIDENT, in contrast, does not generate a patch, since the patch requires an expression with 11 components, which is infeasible due to the vast number of candidate patches that use up to 11 components.

TRIDENT generated two correct patches exclusively, because the correct patches are not in the search space of the other approaches. These correct patches require inserting an assignment. Angelix cannot generate a patch that inserts an assignment. GenProg and Prophet could not generate these patches, since the needed assignments do not appear in the buggy programs. SOSRepair could not generate them because its performance depends on the size and quality of the database of patterns.

> TRIDENT repaired 3 more real bugs from OSS10 dataset that require patches with side effects than the baseline, state of the art semantic repair augmented to synthesise assignments. TRIDENT correctly repaired 2 new bugs from MB37 dataset that the other state of the art tools such as Prophet, SOSRepair, Angelix and GenProg did not repair.

5.2 Containing Path Explosion

Concretely, path explosion manifests itself during a symbolic run in terms of the number of paths visited. To investigate how TRIDENT's state merging mitigates path explosion in our setting, we executed three configurations — TP, AKN, and (3) AKP— on the both OSS10 and CF110 datasets. We compare these three configurations in terms of the average number of paths each visits during k runs over corpus against the time limit of 10 hours.

Table 6 summarises the results of our experiments on CF110 dataset, and Table 5 summarises the results of our experiments on OSS10 bugs. Both tables display the average

TABLE 5: The average number of paths explored in the OSS10 dataset. The buggy versions for which TRIDENT, under its TP configuration, generated a patch are **bolded**. On the corpus, TP visits almost 1000 fewer paths than the closest baseline, on average.

Version	Paths (Average)		
	TP	AKP	AKN
191a84a	5.4	13.2	22.5
7585d81	95.0	96.0	96.0
c160afe	10.0	268.0	1602.5
9944e47	22.0	490.5	616.0
ca99c52	5688.0	6024.2	6352.0
9f5aa48	1175.0	1320.0	9000.0
2a80912	8.0	39.0	104.0
0506e29	12922.0	16807.0	16807.0
5c13ab4	88.0	317.0	936.0
6f7a009	12647.0	15966.0	21378.0
Overall	3266.1	4133.9	5691.3

TABLE 6: The average number of paths explored in the CF110 dataset. Here, TRIDENT, under TP, visits 8 fewer paths on average than the closest baseline.

Class	Paths (Average)		
	TP	AKP	AKN
SISA	12.87	18.43	26.69
DRWV	9.15	13.62	25.23
DMAA	38.86	76.18	98.24
ORRN	15.50	30.36	56.77
OILN	16.47	24.33	40.68
OAIS	16.88	24.72	31.92
Overall	27.39	35.51	37.32

number of paths that each approach explores per suspicious location; for CF110, this average is over all versions from a defect class. In both cases, TRIDENT, under its TP configuration, visits fewer paths than either baseline, demonstrating the effectiveness of its state-merging strategy at coping with path explosion in practice. Figure 12 shows a violin plot for the distribution of number of paths for each class in CF110; here, we see that, for each of TP's runs, the bulk of the area of each violin plot is lower than for the baselines and, crucially, that its tail of outliers is even more markedly lower.

Table 7 demonstrates the solving time and the number of solver queries of the patch synthesiser for OSS10. On average, TRIDENT required fewer solver calls, but since the constraints it passes to the solver have additional clauses to control patch side effects, the total solving time does not sig-

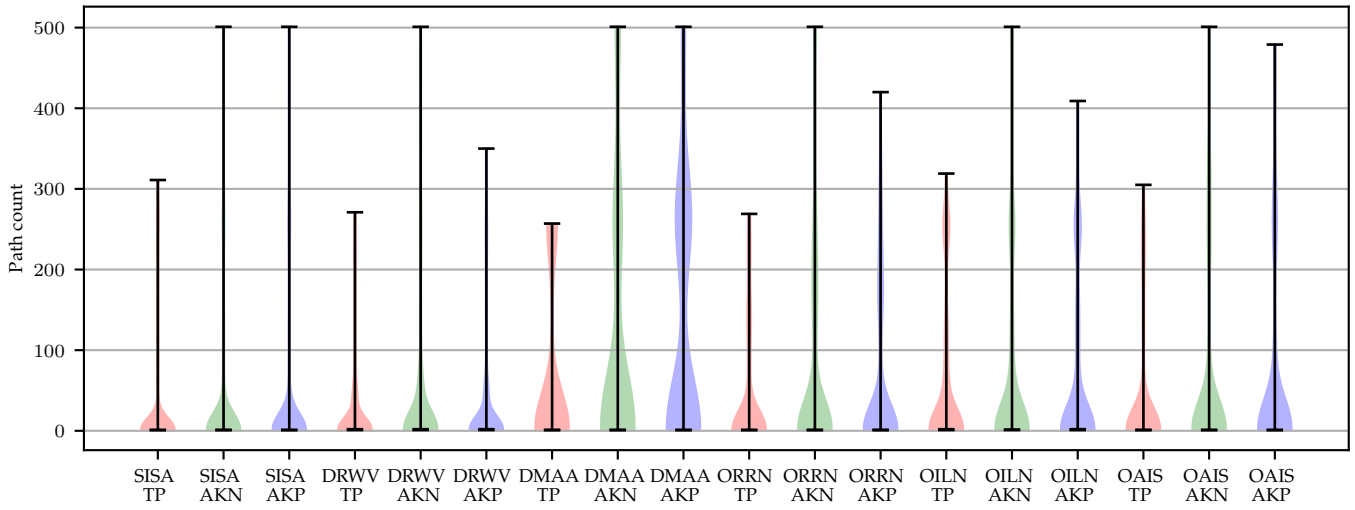


Fig. 12: The distribution of paths explored in the CF110 dataset. The x-axis contains the defect class name and the name of tool configuration. The weight of each violin plot is lower and their tails are shorter than either baseline.

TABLE 7: The patch synthesizer's solving time and the number of solver queries for each tool for the bugs in OSS10 dataset. The versions for which TP generated a patch exclusively are **bolded**. TP has, on average, fewer solver queries, but since its constraints involve additional clauses, the solving time does not considerably differ across configurations.

Version	Solving Time (Seconds)			Query Count		
	TP	AKP	AKN	TP	AKP	AKN
191a84a	99.0	107.3	76.9	1794	1199	372
7585d81	2.4	2.4	2.1	9	9	9
c160afe	17.3	246.0	164.5	1197	8258	7827
9944e47	21.7	162.4	194.8	1328	6836	1213
ca99c52	241.7	170.6	110.2	184	579	461
9f5aa48	1131.1	661.2	1467.4	1570	3617	7824
2a80912	29.1	94.6	72.2	620	564	561
0506e29	79.3	74.7	70.5	72	64	83
5c13ab4	106.2	173.5	171.1	10873	10034	3206
6f7a009	92.8	95.5	98.1	2833	2985	2523
Total	1820.6	1788.2	2427.8	20480	34145	24079

nificantly differ from other techniques. The key advantage of TRIDENT is not in reducing the patch generation time, but reducing the number of paths that are necessary to explore to find a test-passing path, which increases the chance of finding a patch, as demonstrated in Section 5.1.

Overall, TP explores 22.9% fewer paths on average compared to AKP and 26.6% fewer paths when compared to AKN on CF110 dataset, and 21% fewer paths on average compared to AKP and 43% fewer paths when compared to AKN on OSS10 dataset.

> TRIDENT's novel state merging strategy reduces the number of explored paths by 22–43% compared to the baselines — state of the art semantic repair augmented to synthesis assignments and KLEE's general-purpose state merging strategy.

TABLE 8: Plausible and correct patches generated for CF110 bugs; as expected, TRIDENT slightly increases test-overfitting on patches without side effects (first three rows) and its patch priorisation strategy reduces test-overfitting, as the comparison between TP and TN on patches with side-effects (the last three rows) shows.

Class	LoC	Plausible			Correct		
		TP	TN	ANG	TP	TN	ANG
OILN	46 ± 21	2	2	1	0	0	0
OAIS	36 ± 25	8	8	7	1	1	1
ORRN	53 ± 36	7	7	7	2	2	3
Overall		17	17	15	3	3	4
SISA	59 ± 35	16	16	1	9	6	0
DRWV	46 ± 20	8	8	0	5	2	0
DMAA	83 ± 28	0	0	0	0	0	0
Overall		24	24	1	14	8	0

5.3 Resisting Overfitting

A program repair technique overfits a test suite when it produces patches that pass the test suite, but are incorrect. Following convention, we call patches that pass a test suite *plausible*. We measure the degree of test-overfitting as $1 - \frac{\mathfrak{C}}{\mathfrak{P}}$, one minus the ratio of correct $\mathfrak{C}$ to plausible patches $\mathfrak{P}$. In this experiment, we define correctness as passing the held-out test suite provided by Codeflaws benchmark.

We first establish how extending the search space with side effects exacerbates test-overfitting, then show that TRIDENT produces a higher proportion of correct patches than the baseline. For the former, we compare TRIDENT with ANG, which is only applied to expression without side effects. For the later, we compare the overfitting rate of two

TRIDENT's configurations: TP (with enabled patch prioritisation), and TN (with disabled patch prioritisation). We execute all the above configurations on CF110.

Table 8 summarises the results. The top three classes OILN, OAIS and ORRN contain programs that can be fixed with a side-effect-free patch and the bottom three classes SISA, DRWV and DMAA require side effects. TP and TN generate more patches than ANG for the side-effect-free classes OILN, OAIS and ORRN but the extra patches fail the held-out tests. TP and TN both have the test-overfitting ratio of 82.4%. Meanwhile, the test-overfitting of ANG is 73.3%. These results indicate that adding patches with side effects to the search space increases test-overfitting. For classes with side effects SISA, DRWV and DMAA, TP has the test-overfitting rate of 41.7%, and TN has the test-overfitting rate of 66.7%. ANG, in contrast, generates one patch which is incorrect. The results demonstrate that TP's new prioritisation heuristic alleviates test-overfitting by decreasing the rate test-overfitting from 66.7% to 41.7%.

> Despite the fact that handling patches with side effects expands the search space thereby increasing the rate of test-overfitting, TRIDENT's specialised patch prioritisation alleviates this negative effect, reducing overfitting from 66.7% to 41.7%.

6 RELATED WORK

Our work is relevant to three existing research areas: program synthesis, symbolic execution and automated program repair (APR). It leverages the first two to propose TRIDENT, the first test-driven program repair approach that synthesizes patches with side effects without relying on the plastic surgery hypothesis, a database of patterns, or training data, in the form of assignments and calls to loop-free functions. Along the way, TRIDENT proposes a state merging strategy tailored for side effect that alleviates path explosion.

6.1 Program Synthesis

Existing program synthesis techniques such as Synthetic Separation Logic [29], Jennisys [30] and Natural synthesis [31] can synthesize heap-manipulating programs, involving statements with side effects. These techniques are designed to efficiently synthesize small programs from a given specification. In contrast, we designed TRIDENT to infer the synthesis specification thereby compensating for the lack of specification in real-world software. Thus, TRIDENT is orthogonal to these techniques, and can potentially synergise with them to improve patch synthesis.

6.2 Symbolic Execution

TRIDENT relies on symbolic execution for (1) inferring specification for synthesis via state merging, and (2) computing function summaries.

Although various general-purpose state-merging strategies [32], [33] for symbolic execution have been proposed, they have two important limitations in the context of assignment synthesis. First, to align symbolic memory in complex real-world programs, general-purpose state-merging strategies make assumptions about the topology of the states to merge. For example, KLEE [23] assumes that the merged states have exactly the same allocated memory objects and the same set of symbolic variables. These assumptions do not hold in the context of assignment synthesis, reducing the effectiveness of such techniques as demonstrated in Section 5. Second, a generated patch can modify several memory locations, and the number of such locations has to be bounded during path exploration to prevent path explosion. However, it is not clear how general-purpose state-merging techniques can efficiently express such a bound.

The key difference of general-purpose state-merging approaches from our techniques is that they operate on regions of code that impact of which on the state they seek to merge with a dedicated split and merge operations. However, path explosion can occur within these regions. TRIDENT builds state merging for the impact of patches in its search space into its very representation of symbolic state, obviating explicit split and merge operations. This representation incorporates cardinality constraints to bound the number of memory locations that can be modified by the synthesised patch.

To compute function summaries, TRIDENT uses symbolic execution with loop unrolling as in compositional symbolic execution [15].

6.3 Automated Program Repair

We classify program repair techniques into static and test-driven, depending on the correctness criteria used. Static techniques rely on static analysis or formal specification, while test-driven techniques rely on a test suite. Test-driven techniques typically scale to large programs and large search spaces, but are subject to test-overfitting.

TRIDENT is a new test-driven automated program repair technique; its realisation necessitated a new program repair algorithm. It tackles an extended search space, making it more prone to test-overfitting, so it required new patch prioritisation strategy for its defect class. Finally, its evaluation required the compilation of new datasets, tailored to the generation of side-effected patches. We put TRIDENT's contributions into context below.

Test-driven Program Repair Algorithms: Techniques like Genprog [4], RSRepair [34], AE [35], Astor [36], CapGen [37], SimFix [38] and PraPR [39] are generate-and-validate approaches that, unlike TRIDENT, run tests to validate *each* patch they explore. Genprog [4], RSRepair [34] and AE [35] use search algorithms to find a patch. We used Genprog [4] as one of the tools from this class of program repair algorithms to compare against TRIDENT. Astor [36] provides a framework for generate and validate tools to explore the design space of program repair through extension points. CapGen [37] prioritises patches based on the similarity of the context, whereas PraPR [39] is a bytecode level repair algorithm that mutates JVM bytecode to search for the patch. CapGen [37], PraPR [39], SimFix [38] and Astor [36] fix bugs in Java; TRIDENT fixes bugs in C. Hence, it was not feasible to run them on our datasets. Cure [40] is a representative of recent advances of applying neural machine translation for patch generation. Trident, in contrast, is a constraint-based approach, which has inherent advantages concerning the explainability of its output.

Like TRIDENT, techniques like SemFix [12], Angelix [9], SE-ESOC [1], SPR [2], Nopol [41] and S3 [42] employ program synthesis to generate patches. However, these techniques do not explicitly handle side effects, focusing instead on synthesizing side-effect-free expressions. TRIDENT addresses this important limitation. SOSRepair [27] can synthesize patches with side effects, but, to do so, relies on a database of patterns extracted from the program.

Several existing techniques synthesize memory-manipulating transformations. Kapus *et al.* [43] use symbolic execution to synthesize refactorings for string manipulating loops, but focus only on memoryless loops that do not carry information across iterations. TRIDENT's techniques might support more complex refactoring involving non-memoryless loops. Wolverine [3] generates patches for more complex programs that manipulate dynamic data structures, but requires user interaction and was shown to work only for small student programs. Some techniques leverage deep learning to sort and transform code and, like all of these approaches, they are data hungry and rely on a large amount of training data [44], [45]. TRIDENT scales to real-world programs, is fully-automated, and does not rely on training data.

Test-overfitting in Program Repair: Test-overfitting is a central challenge of test-driven program repair [46]; it affects both generate-and-validate [47] and semantic [48] techniques. Researchers have proposed various approaches to tackle this problem. The first group of approaches uses a pre-defined database of transformations to increase the chance of generating correct patch [5], [10], [49]. The second group generates additional tests [50], [51]. The third group of techniques defines a cost function that assigns lower cost to patches that are more likely to be correct [7], [8], [26], [42], [52]. TRIDENT complements existing techniques by proposing a new cost function specialized for the class of defects that require patches with side effects.

Static Program Repair Algorithms: Two exemplars of this class are an approach by Nguyen *et al.* [53] and Footpatch [54]. The approach of Nguyen *et al.* [53] can synthesize patches with side effects and provide formal correctness guarantees, but requires formal specification, and was only demonstrated to repair small programs. TRIDENT scales to real-world programs, but relies on a test suite, so it is subject to test-overfitting. Footpatch [54] takes advantage of separation logic to repair resource leaks, memory leaks, and null dereferences by copying appropriate code fragments. TRIDENT is mostly orthogonal to Footpatch: we designed TRIDENT to handle arbitrary defect classes by synthesizing new code based on tests; it can, however, potentially benefit from separation logic to reason about aliasing.

Program Repair Benchmarks: For evaluating TRIDENT, we constructed new bug datasets, rather than directly using existing benchmarks, such as ManyBugs [16], IntroClass [16], Codeflaws [17], DBGBench [55]. We did not use ManyBugs, because the majority of its bugs either require side-effect-free patches or complex patches than involve many lines in several files. We did not use DBGBench, because it did not contain enough bugs with side-effect-free modifications, and some of its bugs were not reproducible with KLEE. IntroClass contains only very small program. We reused some defect classes from Codeflaws that involve the insertion of assignments or functions calls.

7 CONCLUSION

This paper proposes the first test-driven program repair approach that synthesizes patches with side effects without relying on the plastic surgery hypothesis, a database of patterns, or training data. TRIDENT relies on a tight integration of three techniques: (1) multi-path specification inference that allows to efficiently infer specification for synthesizing patches with side effects, (2) program synthesis encoding that allows to express modifications such as assignments and function calls, and (3) patch prioritization heuristics that helps to alleviate test-overtiffing due to side effects. Our evaluation demonstrates that TRIDENT is able to repair real bugs in open-source projects, and the proposed prioritization strategy indeed helps to increase the chance of generating correct patches.

REFERENCES

[1] S. Mechtaev, A. Griggio, A. Cimatti, and A. Roychoudhury, "Symbolic execution with existential second-order constraints," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 389–399.

[2] F. Long and M. Rinard, "Staged program repair with condition synthesis," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 166–178.

[3] S. Verma and S. Roy, "Synergistic debug-repair of heap manipulations," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 163–173.

[4] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for 8 each," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 3–13.

[5] J. Kim and S. Kim, "Automatic patch generation with context-based change application," *Empirical Software Engineering*, vol. 24, no. 6, pp. 4071–4106, 2019.

[6] Z. Chen, S. J. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, 2019.

[7] S. Mechtaev, J. Yi, and A. Roychoudhury, "Directfix: Looking for simple program repairs," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 448–458.

[8] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise condition synthesis for program repair," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 416–426.

[9] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multi-line program patch synthesis via symbolic analysis," in *Proceedings of the 38th international conference on software engineering*, 2016, pp. 691–701.

[10] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 802–811.

[11] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro, "The plastic surgery hypothesis," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 306–317.

[12] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 772–781.

[13] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 1. IEEE, 2010, pp. 215–224.

[14] R. Alur, R. Bodik, G. Juniwal, M. M. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, *Syntax-guided synthesis*. IEEE, 2013.

[15] S. Anand, P. Godefroid, and N. Tillmann, "Demand-driven compositional symbolic execution," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 367–381.

[16] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer, "The manybugs and introclass benchmarks for automated repair of c programs," *IEEE Transactions on Software Engineering*, vol. 41, no. 12, pp. 1236–1256, 2015.

[17] S. H. Tan, J. Yi, S. Mechtaev, A. Roychoudhury *et al.*, "Codeflaws: a programming competition benchmark for evaluating automated program repair tools," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2017, pp. 180–182.

[18] F. Long and M. Rinard, "An analysis of the search spaces for generate and validate patch generation systems," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 702–713.

[19] D. W. Barron, J. N. Buxton, D. Hartley, E. Nixon, and C. Strachey, "The main features of cpl," *The Computer Journal*, vol. 6, no. 2, pp. 134–143, 1963.

[20] C. Barrett and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Model Checking*. Springer, 2018, pp. 305–343.

[21] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*. IEEE, 2007, pp. 89–98.

[22] "Clang: Llvm's default frontend," https://clang.llvm.org/, accessed: 2021-03-03.

[23] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[24] M. Gario and A. Micheli, "Pysmt: a solver-agnostic library for fast prototyping of smt-based algorithms," in *SMT workshop*, vol. 2015, 2015.

[25] "Docker OS virtualisation," https://www.docker.com/, accessed: 2021-03-03.

[26] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016, pp. 298–312.

[27] A. Afzal, M. Motwani, K. Stolee, Y. Brun, and C. Le Goues, "Sosrepair: Expressive semantic search for real-world program repair," *IEEE Transactions on Software Engineering*, 2019.

[28] M. Monperrus, "A critical review of" automatic patch generation learned from human-written patches": Essay on the problem statement and the evaluation of automatic software repair," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 234–242.

[29] N. Polikarpova and I. Sergey, "Structuring the synthesis of heap-manipulating programs," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–30, 2019.

[30] K. R. M. Leino and A. Milicevic, "Program extrapolation with jennisys," in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, 2012, pp. 411–430.

[31] X. Qiu and A. Solar-Lezama, "Natural synthesis of provably-correct data-structure manipulations," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–28, 2017.

[32] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," *Acm Sigplan Notices*, vol. 47, no. 6, pp. 193–204, 2012.

[33] K. Sen, G. Necula, L. Gong, and W. Choi, "Multise: Multi-path symbolic execution using value summaries," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 842–853.

[34] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 254–265.

[35] W. Weimer, Z. P. Fry, and S. Forrest, "Leveraging program equivalence for adaptive program repair: Models and first results," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 356–366.

[36] M. Martinez and M. Monperrus, "Astor: Exploring the design space of generate-and-validate program repair beyond genprog," *Journal of Systems and Software*, vol. 151, pp. 65–80, 2019.

[37] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 1–11.

[38] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*, 2018, pp. 298–309.

[39] A. Ghanbari and L. Zhang, "Prapr: Practical program repair via bytecode mutation," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1118–1121.

[40] N. Jiang, T. Lutellier, and L. Tan, "Cure: Code-aware neural machine translation for automatic program repair," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1161–1173.

[41] J. Xuan, M. Martinez, F. Demarco, M. Clement, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 34–55, 2016.

[42] X.-B. D. Le, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser, "S3: syntax-and semantic-guided repair synthesis via programming by examples," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 593–604.

[43] T. Kapus, O. Ish-Shalom, S. Itzhaky, N. Rinetzky, and C. Cadar, "Computing summaries of string loops in c for better testing and refactoring," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 874–888.

[44] M. White, M. Tufano, M. Martinez, M. Monperrus, and D. Poshyvanyk, "Sorting and transforming program repair ingredients via deep learning code similarities," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 479–490.

[45] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," *arXiv preprint arXiv:2103.06333*, 2021.

[46] C. L. Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Communications of the ACM*, vol. 62, no. 12, pp. 56–65, 2019.

[47] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, "Is the cure worse than the disease? overfitting in automated program repair," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 532–543.

[48] X. B. D. Le, F. Thung, D. Lo, and C. Le Goues, "Overfitting in semantics-based automated program repair," *Empirical Software Engineering*, vol. 23, no. 5, pp. 3007–3033, 2018.

[49] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. Le Traon, "Fixminer: Mining relevant fix patterns for automated program repair," *Empirical Software Engineering*, pp. 1–45, 2020.

[50] J. Yang, A. Zhikhartsev, Y. Liu, and L. Tan, "Better test cases for better automated program repair," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 831–841.

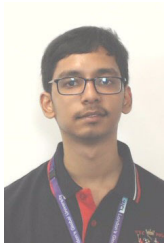
[51] X. Gao, S. Mechtaev, and A. Roychoudhury, "Crash-avoiding program repair," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 8–18.

[52] L. D'Antoni, R. Samanta, and R. Singh, "Qlose: Program repair with quantitative objectives," in *International Conference on Computer Aided Verification*. Springer, 2016, pp. 383–401.

[53] T.-T. Nguyen, Q.-T. Ta, I. Sergey, and W.-N. Chin, "Automated repair of heap-manipulating programs using deductive synthesis," 2021.

[54] R. van Tonder and C. L. Goues, "Static automated program repair for heap properties," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 151–162.

[55] M. Böhme, E. O. Soremekun, S. Chattopadhyay, E. Ugherughe, and A. Zeller, "Where is the bug and how is it fixed? an experiment with practitioners," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 117–128.



Nikhil Parasaram received his BTech in computer science and engineering from the International Institute of Information Technology Hyderabad. He is currently working towards his PhD at the University College London under the supervision of Prof Sergey Mechtaev and Prof Earl Barr. His PhD work focuses on automated synthesis of memory manipulating patches.

Earl T. Barr received the PhD degree from the University of California, Davis. He is a Professor at the University College London. Earl works on testing and program analysis, software engineering, debugging, big code, and cybersecurity. Earl has won three ACM distinguished paper awards; his paper entitled “The Naturalness of Software” was a research highlight in the May 2016 Communications of the ACM. Earl dodges vans and taxis on his bike commute in London. Web page: http://earlbarr.com.

Sergey Mechtaev is a Lecturer at University College London. Previously, he obtained a PhD degree from the National University of Singapore. His research interests include automated program repair, program synthesis and symbolic execution. He has received ACM SIGSOFT Outstanding Dissertation Award for his PhD thesis on semantic program repair. He has been a programme committee member of ASE, and a reviewer of TSE, TOSEM and EMSE. Web page: http://mechtaev.com