2015 30th IEEE/ACM International Conference on Automated Software Engineering

Repairing Programs with Semantic Code Search

Yalin Ke Kathryn T. Stolee
Department of Computer Science
Iowa State University
{yke, kstolee}@iastate.edu

Claire Le Goues
School of Computer Science
Carnegie Mellon University
clegoues@cs.cmu.edu

Yuriy Brun
College of Information and Computer Science
University of Massachusetts, Amherst
brun@cs.umass.edu

***Abstract*—Automated program repair can potentially reduce debugging costs and improve software quality but recent studies have drawn attention to shortcomings in the quality of automatically generated repairs. We propose a new kind of repair that uses the large body of existing open-source code to find potential fixes. The key challenges lie in efficiently finding code semantically similar (but not identical) to defective code and then appropriately integrating that code into a buggy program. We present SearchRepair, a repair technique that addresses these challenges by (1) encoding a large database of human-written code fragments as SMT constraints on input-output behavior, (2) localizing a given defect to likely buggy program fragments and deriving the desired input-output behavior for code to replace those fragments, (3) using state-of-the-art constraint solvers to search the database for fragments that satisfy that desired behavior and replacing the likely buggy code with these potential patches, and (4) validating that the patches repair the bug against program test suites. We find that SearchRepair repairs 150 (19%) of 778 benchmark C defects written by novice students, 20 of which are not repaired by GenProg, TrpAutoRepair, and AE. We compare the quality of the patches generated by the four techniques by measuring how many independent, not-used-during-repair tests they pass, and find that SearchRepair-repaired programs pass 97.3% of the tests, on average, whereas GenProg-, TrpAutoRepair-, and AE-repaired programs pass 68.7%, 72.1%, and 64.2% of the tests, respectively. We conclude that SearchRepair produces higher-quality repairs than GenProg, TrpAutoRepair, and AE, and repairs some defects those tools cannot.**

I. Introduction

Buggy software costs the global economy billions of dollars annually [8], [60]. One major reason software defects are so expensive is that software companies must dedicate considerable developer time [75] to manually finding and fixing bugs in their software. Unfortunately, manual bug repair, the industry standard, is largely unable to keep up with the volume of defects in extant software [2]. Despite their established detrimental impact on a company's bottom line, known defects ship in mature software projects [45], and many defects, including those that are security-critical, remain unaddressed for long periods of time [32].

At the same time, the expansion of the open-source movement has led to many large, publicly accessible source code databases, such as GitHub, BitBucket, and SourceForge. Because many programs include routines, data structures, and designs that have been previously implemented in other software projects [11], [12], [24], we posit that, if a method or component of a software system contains a defect, with high probability, there exists a similar but correct version of that component in some publicly accessible software project. The research challenge lies in how to automatically find and use such implementations to repair bugs.

Our key idea is to use *semantic code search* [68] over existing open-source code to find correct implementations of buggy components and methods, and use the results to *automatically generate patches* for software defects. Semantic search identifies code by what it *does*, rather than by syntactic keywords. We develop SearchRepair, a new technique predicated on our idea. SearchRepair:

1) *Encodes* a large database of human-written code fragments as satisfiability modulo theories (SMT) constraints on their input-output behavior.
2) *Localizes* a defect to likely buggy program fragments.
3) *Constructs*, for each fragment, a lightweight input-output profile that characterizes desired functional behavior as SMT constraints.
4) *Searches* the database, using state-of-the-art constraint solvers, for fragments that satisfy such a profile. These fragments become potential patches when contextualized and inserted into the buggy regions, replacing the original potentially faulty code.
5) *Validates* each potential patch against the program test suite to determine if it indeed repairs the defect in question.

To make SearchRepair possible, we first extend our previous work in semantic code search [68] to C program fragments. Second, we adapt spectrum-based fault localization [36] to identify candidate regions of faulty code and construct input-output profiles to use as input to semantic search. Third, we build the infrastructure to perform semantic code search over the SMT-encoded code database, adapt the returned code fragment to the defective context via variable renaming, and validate against provided test suites.

Our goal with SearchRepair is to produce high quality patches while still addressing a broad range of defects. A key feature of a high quality patch, whether human- or tool-generated, is that it *generalizes* to the full, desired, often unwritten specification of correct program behavior. This is a challenge for automatic repair techniques (e.g., [3], [7], [10], [11], [15], [16], [18], [19], [21], [28], [33], [35], [39], [42], [48], [49], [50], [51], [52], [54], [56], [57], [61], [69], [70], [73], [74], [76]), many of which use test suites to guide and evaluate patching efforts. Modern test-suite guided repair techniques, particularly those following a *generate-and-validate* paradigm (i.e., heuristically constructing and then testing large numbers of candidate repairs), although typically general and scalable, often produce poor-quality patches that overfit to the specification test suites used to guide patch generation [20], [57], [65].

By definition, test suites only encode a partial specification of correct behavior. A patch that is correct according to a given test suite may therefore not be fully correct when evaluated with respect to a hypothetical full correctness specification. This is analogous to the well-known machine learning phenomenon of *overfitting* to an objective function, where the program

978-1-5090-0025-8/15 $31.00 © 2015 IEEE
DOI 10.1109/ASE.2015.60

repair objective is defined as satisfying a given partial correct specification (test suite). Since there are infinitely many programs that satisfy any given partial specification, and a repair technique can produce any one of them, there is reasonably high probability that a resulting repaired program will not adhere to the unwritten, full, desired specification, absent additional efforts to ensure quality control.

To that end, there are two primary ways that SearchRepair differs from previous repair approaches. First, because it uses test cases to guide a semantic search for candidate fix code, it bridges the gap between *correct-by-construction* techniques [35], [49], [50], [52], [70] predicated on program synthesis and *generate-and-validate* techniques that heuristically create and then test large numbers of candidate repairs [1], [11], [12], [15], [18], [39], [47], [54], [57], [63], [69], [73], [74]. Second, although several previous techniques also reuse human-written code from elsewhere in a program, or instantiate human-written templates to effect local changes, SearchRepair identifies larger sections of human-written candidate fix code from other projects, which it uses to replace defective regions wholesale. Because code is repetitive and often reimplements routines [5], [11], [12], [24], it may be possible to find correct alternative versions of a given buggy piece of functionality in other software systems, given a sufficiently large database.

Our core assumption is that a larger block of human-written code, such as a method body, that fits a given partial specification is more likely to satisfy the unwritten specification than a randomly chosen set of smaller edits generated with respect to the same partial specification. Human developers typically possess a notion of desired full correctness that is not necessarily completely captured by a partial test suite, but encoded nonetheless in the resulting functionality. Reusing human-written code at this higher level of granularity is thus more likely to result in patches that capture the underlying human intuition than are smaller edits. Consider an example of a program that has a bug in a subroutine that sorts an array of integers. No finite set of tests can uniquely define sorting the array. Using a test suite, automated program repair is as likely to produce a sorting routine as it is to produce a different routine that works for the example tests but fails on other, unwritten tests. However, in a large body of human-written code, there are many more sorting routines than other routines that satisfy such tests, so searching for such a human-written routine is more likely to generalize to the unwritten specification.

Our evaluation shows that at least in the context of our experiments, this core assumption holds. Simultaneously, while SearchRepair repairs a similar fraction of defects to GenProg [42], [74], TrpAutoRepair [56], and AE [73], SearchRepair can address some defects that none of the other techniques patch. This suggests that SearchRepair is complementary to prior work, able to tackle defects that were previously not amenable to repair via state-of-the-art generate-and-validate techniques.

We evaluate SearchRepair on a benchmark of 778 C defects written by novice students [43]. This benchmark was specifically designed to evaluate program repair techniques, and among other features, each defect in this benchmark has two independent test suites, allowing us to repair the defect with SearchRepair, GenProg, TrpAutoRepair, and AE using one test suite, and then evaluate the quality of the repair on the independent test suite [65]. We find that the quality of SearchRepair's repairs is much higher on average than that of the other techniques. SearchRepair-repaired programs pass 97.3% of the held-out tests, on average, whereas GenProg-, TrpAutoRepair-, and AE-repaired programs pass 68.7%, 72.1%, and 64.2% of the held-out tests, respectively.

This paper's main contributions are:

- An extension of state-of-the-art semantic code search to new primitives and functions, and an implementation for the C programming language.
- SearchRepair, a semantic-code-search-based automated program repair approach and its implementation, including an extension of spectrum-based fault localization for use in identifying candidate fragments of defective code: `https://github.com/ProgramRepair/SearchRepair/`.
- An evaluation on 778 defects showing that SearchRepair can repair 150 (19.3%) of them. Of the 310 defects that SearchRepair, GenProg, TrpAutoRepair, or AE repairs, 20 (6.5%) are only repaired by SearchRepair. We also demonstrate that SearchRepair produces significantly higher quality repairs, on average, as compared to the other tools.

The remainder of this paper is structured as follows. Section II provides relevant background on the state-of-the-art in semantic code search, automatic defect repair, and measuring patch quality. Section III details the SearchRepair approach. Section IV evaluates SearchRepair and compares it with three other repair techniques. Section V places our work in the context of related research. Finally, Section VI summarizes our contributions.

II. Background

We begin by introducing background concepts in semantic code search in Section II-A, automated program repair in Section II-B, and limitations of current repair techniques, focusing on output quality, in Section II-C.

A. Semantic code search

In this paper, we extend our prior work on input-output example-based semantic code search [66], [67], [68]. We detail our extensions and illustrate with examples in Section III and focus in this subsection on a high-level overview to ground the subsequent material.

Code search uses a specification to identify code in a repository that matches that specification. *Syntactic* code search uses syntactic features such as keywords and variable names as the specification. For example, a developer trying to find a method for string replacement in C might search for "`c string replace`". By contrast, *semantic* search uses behavioral properties as the specification. For example, the developer could supply several input-output pairs for the desired string replacement function, demonstrating by example the desire for code that *performs replacement*. Semantic code search offers the notable advantage over keyword-based search because a developer can search by example, and need not guess the words that describe the behavior.

All search involves *indexing* and *searching*. Indexing constructs the database of information over which the search will be performed. Searching uses a user- or tool-supplied query to identify potential results, often ranked by predicted relevance, from the indexed database. Our approach to semantic code search (1) indexes code fragments by converting them to symbolic constraints describing their input-output behavior, and (2) converts input-output example queries into additional constraints describing desired behavior and uses off-the-shelf

SMT solvers to identify indexed fragments that satisfy the desired behavior constraints:

Indexing source code as SMT constraints. The indexing step happens offline and produces a database relating code fragments to be searched to sets of SMT constraints over the input-output behavior and the segment's type signature. For each fragment, symbolic execution [13], [14], [40] enumerates the feasible paths through it and our approach converts code constructs and predicates controlling each path's execution into SMT constraints describing variable types and values. The fragment's SMT encoding is the disjunction of the paths' constraints. Section III-B describes this encoding process in more detail.

Searching with an input-output specification. To find code in the database that matches desired behavior expressed as input-output pairs, our approach converts the input-output pairs into SMT constraints over the inputs and outputs. For each fragment in the database that satisfies the required type signature, the approach conjoins the fragment's constraint set with those describing desired behavior and invokes an SMT solver to determine if the overall set of constraints is satisfiable (subject to a mapping constraint between the variables in the input-output pair to those in the fragment). If the overall set is satisfiable, the constraints describing the code fragment satisfy the partial behavioral specification imposed by the input-output pairs, and the fragment is returned as a potential match. Section III-E describes this search process in more detail.

B. Automatic program repair

The high costs of defective software motivates research in automatically and generically repairing bugs. The input to any such technique is typically a program with a defect and a mechanism to validate correct and incorrect behavior in that program. One class of such techniques is *correct-by-construction repair*. These techniques rely on inferred or provided specifications to guide sound patch synthesis [19], [35], [49], [50], [70], [71]. The other primary class of repair approaches is *generate-and-validate repair*, which uses search-based software engineering techniques [31], [57], [69] or predefined repair templates [15], [39], [54] to generate many candidate patches for a bug, and then validate them using indicative workloads or test suites.

In this work, we compare SearchRepair to three prior generate-and-validate techniques: GenProg [42], [44], [72], [74], TrpAutoRepair [56], and AE [73]. (TrpAutoRepair was also published under the name RSRepair in "The strength of random search on automated program repair" by Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang in the 2014 International Conference on Software Engineering; we refer to the original name in this paper.) All three of these techniques are publicly available and open-source, all target C programs, and all are general in the types of defects they repair. Generate-and-validate techniques differ in (1) how they choose which program locations to modify, (2) which modifications they permit, and (3) how they validate candidate patches. These differences encapsulate the three key hurdles that such techniques must overcome to find patches [73]. First, such a technique must *localize* the defect to a set of candidate program locations to be potentially changed. Previous techniques, including the three to which we compare in this work, typically use or adapt prior fault localization work [36], [59] to identify promising locations. Second, such a technique must modify the code in an attempt to repair it. This describes the *fix space* of a particular program repair problem. GenProg, TrpAutoRepair, and AE address this challenge using the observation that programs are often internally repetitive [5], [25] and limit changes to deleting constructs and copying constructs from elsewhere in the same program. TrpAutoRepair and AE restrict attention to single-edit patches. Third, such a technique validates candidate patches and uses the results of that validation to dictate how the space of candidate patches will be traversed. GenProg, TrpAutoRepair, and AE, and most prior techniques, evaluate patches using test cases.

GenProg uses the test results to define an objective function that informs an evolutionary search [22], [41] over the space of candidate patches. TrpAutoRepair samples candidate patches randomly, and gains efficiency by prioritizing the test case execution order, and only runs as many test cases as necessary to find one that fails. AE is deterministic and uses heuristic computation of program equivalence to prune the space of possible repairs, selectively choosing which tests to use to validate intermediate patch candidates. AE uses the same change operators as GenProg and TrpAutoRepair, but rather than using a genetic or randomized search algorithm, AE exhaustively searches through the space of all non-equivalent k-distance edits (with published evaluations using $k = 1$).

C. Measuring patch quality

Generate-and-validate repair has been shown to scale to bugs in large systems, with human-competitive costs [19], [35], [39], [42], [50], [54], [70]. However, such techniques provide no correctness guarantees besides validation against the provided workloads or tests and tend to generate short patches. It may therefore be difficult for them to repair complex defects. Importantly, such techniques may produce lower quality patches than those written by humans [23], [39], and recent studies have identified significant concerns with the functional correctness of automatically-repaired programs [20], [57], [65].

One way to measure the quality of a repair is to obtain a separate, independent test suite that the repair technique cannot access when generating the repair. This test suite offers a second, independent partial specification. Repairs that adhere to the unwritten, full specification will satisfy both the test suites. Meanwhile, repairs that overfit to the test suite used during repair generation may not satisfy the independent test suite. Thus, such a second test suite can be used to estimate repair quality.

In this work, we use a benchmark of 778 defects, each with two such independent test suites, specifically designed for evaluating automated repair [43]. Prior experiments with GenProg, TrpAutoRepair, and AE on this benchmark have shown that, on this dataset, these tools produce repairs that pass only 68.7%, 72.1%, and 64.2%, respectively, of the tests in the independent test suite not used during patch generation [65].

The rest of this paper describes the technique that makes repair via searching for human-written code possible and tests our hypotheses that such human-written code can be used to repair defects, and that the quality of the repairs is higher. Section IV will show that this approach does repair defects, that it repairs some defects that prior techniques cannot repair, and that the resulting repairs pass, on average, 97.3% of the independent tests not used during repair generation.

```
1  int main() {
2   int a, b, c, median = 0;
3   printf("Please enter 3 numbers separated by spaces >");
4   scanf("%d%d%d", &a, &b, &c);
5   if ((a<=b && a>=c) || (a>=b && a<=c))
6      median = a;
7   else if ((b<=a && b>=c) || (b>=a && b<=c))
8      median = b;
9   else if ((c<=b && a>=c) || (c>=b && a<=c))
10     median = c;
11  printf("%d is the median", median);
12  return 0;
13 }
```

Fig. 1: A student-written, buggy program to print the median of three integers. Note that the comparison between variables `a` and `c` on line 9 are flipped, such that `c` will never be identified as the median.

	input	expected output	program output	test result
t_1	9 9 9	`"9 is the median"`	`"9 is the median"`	pass
t_2	0 2 3	`"2 is the median"`	`"2 is the median"`	pass
t_3	0 1 0	`"0 is the median"`	`"0 is the median"`	pass
t_4	0 2 1	`"1 is the median"`	`"0 is the median"`	fail
t_5	8 2 6	`"6 is the median"`	`"0 is the median"`	fail

Fig. 2: Five test cases for the program from Figure 1.

III. SearchRepair

We propose SearchRepair, a tool for automated program repair using semantic code search. SearchRepair uses fault localization to identify buggy fragments of code. For each identified candidate buggy fragment, SearchRepair extracts program state in the form of dynamic variable values over the test cases. The program state before and after each fragment on a single test case forms one input-output example. The values for passing test cases form positive input-output examples; the values for failing test cases form negative input-output examples whose corresponding behavior a repaired program should avoid. The full set of these input-output examples is called a *profile*. SearchRepair uses these profiles to search a database of code to find code fragments that can serve as potential patches, replacing the buggy fragments. Once a potential patch is found, SearchRepair renames the variables to fit the context, replaces the buggy code with the patch, and runs all tests. If all tests pass, SearchRepair accepts the patch; otherwise, SearchRepair moves on to another patch. This process continues until a patch is found or all matches from the semantic search are exhausted.

SearchRepair extends our prior work on input-output example-based semantic code search [67], [68], which targeted Java and encoded fewer primitives and functions than SearchRepair. SearchRepair targets C code, supports all primitives and mathematical functions from the Java implementation, and additionally encodes `char*` variables, the modulo operator, and the string library functions `isdigit`, `islower`, `isupper`, `strcmp`, and `strncmp`.

In this section, we describe SearchRepair in detail, starting with an illustrative example.

(a) fully correct code fragment:

```
1  if((x <= y && x >= z) || (x >= y && x <=z))
2      m = x;
3  else if((y <=  x && y >= z) || (y >= x && y <= z))
4      m = y;
5  else
6      m = z;
```

(b) partially correct code fragment:

```
1  if ((a <= b && a >= c) || (a >= b && a <= c))
2    median = a;
3  else if ((b <= a && b >= c) || (b >= a && b <= c))
4    median = b;
5  else if ((c <= b && a <= c) || (c >= b && a <= c))
6    median = c;
```

Fig. 3: Two candidate code fragments to be used to replace lines 5–10 in Figure 1. Code fragment (a) repairs the bug, passing all five tests; meanwhile, (b) only partially repairs the bug, as tests t_1, t_2, t_3, and t_5 pass, but test t_4 still fails.

A. Illustrative example

We use a short program (Figure 1) and a test suite (Figure 2) to illustrate the concepts underlying SearchRepair.

Prior to initiating any particular repair effort, SearchRepair constructs a database of candidate repair code fragments over which the semantic search for repairs will be conducted. These fragments can come from code in open-source repositories, large in-house code bases, or any other source of code fragments. As Section III-B describes, indexing will create a database relating the code fragments to their type signatures and sets of constraints on their input-output behavior. For introducing the example program, we will assume this database already exists.

Consider the code in Figure 1, adapted from one of the student programming assignments we use to evaluate SearchRepair in Section IV. The `main` function takes three integers from the user, and identifies and prints their median. This code is partially incorrect, as the test suite in Figure 2 exposes. Of the five test cases t_1–t_5 in Figure 2, t_4 and t_5 fail on this code. This is because when `c` is the median and $a \neq c$, the code always outputs zero (the default value of `median`), rather than the value of `c`, because the `if` predicate on line 9 is incorrect.

SearchRepair uses tests and dynamic fault localization to identify likely buggy regions. Assume fault localization correctly identifies line 9 in Figure 1 as potentially buggy. SearchRepair attempts to replace the entire block of control flow (the full `if-then-else` block in lines 5–10), as Section III-C explains in detail.

Next, SearchRepair observes the test case executions around this code fragment to construct input and output states in terms of the observed runtime values of the local variables (Section III-D). The passing test cases characterize correct behavior: Executing t_1, the input program state (state before line 5) is $\{$`a = 9, b = 9, c = 9, median = 0`$\}$. The output program state (state after line 10) is $\{$`a = 9, b = 9, c = 9, median = 9`$\}$. Similarly, the failing test cases characterize behavior to be avoided: Executing t_4, the input and output program states are both $\{$`a = 0, b = 2, c = 1, median = 0`. The overall input-output profile, generated from the five test cases, consists one input-output state pair per test case, annotated with whether the behavior is correct. Figure 4 shows a full profile for the example.

test	input	input state	output state	test result
t_1	9 9 9	a:9:int b:9:int c:9:int median:0:int	a:9:int b:9:int c:9:int median:9:int	pass
t_2	0 2 3	a:0:int b:2:int c:3:int median:0:int	a:0:int b:2:int c:3:int median:2:int	pass
t_3	0 1 0	a:0:int b:1:int c:0:int median:0:int	a:0:int b:1:int c:0:int median:0:int	pass
t_4	2 0 1	a:0:int b:2:int c:1:int median:0:int	a:0:int b:2:int c:1:int median:0:int	fail
t_5	2 8 6	a:2:int b:8:int c:6:int median:0:int	a:2:int b:8:int c:6:int median:0:int	fail

Fig. 4: An input-output profile for the program from Figure 1, constructed using the test suite from Figure 2.

A profile, such as the one in Figure 4, serves as input to the semantic search engine described in Section III-E, which searches the database of code fragments constructed by the indexing step. SearchRepair considers each code fragment the search finds as a potential patch (subject to variable renaming), integrates it into the program, and runs the test suites to check if it repairs the defect (Section III-F). For example, the code fragment in Figure 3(a) can be used to construct a repair for lines 5–10 in the program in Figure 1 such that it passes all the test cases in Figure 2. Note, however, that the variable names Figure 3(a) do not match those used in the original program. SearchRepair modifies such fragments by mapping the variables in the original code context to those in the candidate fix fragment. For this example, one of the several valid mappings is `x` $\mapsto$ `a`, `y` $\mapsto$ `c`, `z` $\mapsto$ `b`, and `m` $\mapsto$ `median`. The resulting modified program passes all test cases, as desired.

B. Indexing for semantic code search

As Section II-A summarized, SearchRepair indexes a set of code fragments to create a searchable database. This section illustrates indexing with an example code fragment from Figure 3(a). Indexing consist of:

Collecting candidate source code fragments. SearchRepair collects entire blocks of statements surrounded by predicates (e.g., for statements inside `if-then-else` conditions, SearchRepair captures the entire `if-then-else` block), and sequences of 1 to 5 statements of code. This captures our intuition that higher-granularity patches are more likely to lead to higher-quality patches than lower-granularity patches, while still enabling sufficiently expressive repair of defects. SearchRepair does not include in its indexed database code fragments with loops, but SearchRepair can repair code constructs that contain loops, as Section III-C further explains. We leave encoding fragments with loops to future work.

Statically enumerating execution paths in each fragment. SearchRepair translates all fragments into static single assignment (SSA) format and then uses symbolic execution [13], [14], [40] to enumerate statically feasible intra-procedural program paths. These paths consist of *variable declarations* that contain the names and types of the variables used in the fragment, *assumptions* (path conditions) that represent predicates controlling path execution (e.g., the expression controlling a conditional branch execution), and *statements* that capture relevant code constructs along a path, such as assignments and function calls. Figure 5 shows the four local variables (*vars*) and three paths (p_1, p_2, and p_3) for the C fragment from Figure 3(a). For multi-path fragments, each feasible path is translated into a separate set of constraints (described next) and the whole fragment encoding is a disjunction of the constraints of all the paths.

```
vars: LOCAL(int x, int y, int z, int m)
p1:   ASSUME[(x <= y && x >= z)|| (x >= y && x <=z)]
      STMT[m = x]
p2:   ASSUME[not((x <= y && x >= z)|| (x >= y && x <=z))
           && ((y <= x && y >= z)|| (y >= x && y <= z))]
      STMT[m = y]
p3:   ASSUME[not((x <= y && x >= z)|| (x >= y && x <=z))
           && not((y <= x && y >= z)|| (y >= x && y <= z))]
      STMT[m = z]
```

Fig. 5: The local variables and the three paths that describe the potential behavior of the C code fragment in Figure 3(a).

Translating paths into SMT constraints over variables and control flow. There are two types of constraints representing a fragment's behavior: those concerning variable declaration and types (`LOCAL` in Figure 5), and those concerning variable values (`ASSUME` and `STMT` in Figure 5). Variable types are constrained to their statically declared types in the program fragment. For example, the *vars* in Figure 5 are associated with the following environment constraint describing in-scope variables:

$$\exists x, y, z, m : \texttt{int} \qquad (\gamma)$$

Constraints on variable values are translated from the predicates that control path execution and explicit statements along the statically-enumerated path. For example, p_1's execution assumes the predicate in the first `if` statement, constraining the implicated variables accordingly; p_2 assumes that the first `if` predicate evaluates to false (is negated) but the second (in the `else if` on line 3 of Figure 3(a)) is true; etc. These predicates are captured in the `ASSUME` statements from Figure 5. Any remaining statements in a path capture variable manipulation. Overall, for the fragment from Figure 3(a):

$$(m = x) \ \wedge \ ((z \leq x \leq y) \vee (y \leq x \leq z)) \qquad (\phi_1)$$

$$(m = y) \ \wedge \ (\neg((z \leq x \leq y) \vee (y \leq x \leq z)) \wedge ((z \leq y \leq x) \vee (x \leq y \leq z))) \qquad (\phi_2)$$

$$(m = z) \ \wedge \ (\neg((z \leq x \leq y) \vee (y \leq x \leq z)) \wedge \neg((z \leq y \leq x) \vee (x \leq y \leq z))) \qquad (\phi_3)$$

Each path is thus encoded as a conjunction of the variable value and type constraints, and the entire fragment is encoded as a disjunction of all n paths in the fragment:

$$\bigvee_{1}^{n} \gamma \wedge \phi_n$$

SearchRepair ultimately translates all constraints into a format suitable for the Z3 SMT solver and stores them in a database. SearchRepair could instead use other SMT solvers with nonlinear integer arithmetic with uninterpreted function symbols (UFNIA) theory. Integers and booleans are built-in types for this SMT theory; strings and characters are not. We treat characters as integers, translating into associated integer values when known, and encode strings via constraints on string length and location and value of each individual character. Our translation has special handling for common arithmetic operators and functions, and for string library functions in C; these translations are novel with respect to our previous work on semantic search for Java fragments [68].

SearchRepair stores each translated code fragment in a relational database relating path constraints, original source code, and type signature of the fragment.

C. Fault localization

We adapt the Tarantula fault localization technique [36] to identify code fragments that SearchRepair attempts to replace with patch code. Tarantula is a well-known and foundational example of a class of techniques that implement spectrum-based fault localization. Tarantula uses coverage information provided by a set of passing and failing test cases to compute suspiciousness scores that characterize the likelihood that a given statement of code is responsible for a failing test case. Given a program and a test suite, Tarantula executes each test in the suite and records each statement the test executes, and if the test passes or fails. It uses the number of passing and failing test cases on each executed statement s to compute a *suspiciousness* score:

$$suspiciousness(s) = \sqrt{\left(\frac{failed(s)}{total\_failed}\right)\left(\frac{failed(s)}{failed(s)+passed(s)}\right)}$$

where $total\_failed$ denotes the total number of failed test cases, and $failed(s)$ and $passed(s)$ denote, respectively, the number of failing and passing test cases that execute s. A high $suspiciousness(s)$ value suggests that s is more likely to be buggy. Prior work has varied in the weighting of the two factors in the formula; SearchRepair weights them equally.

Given the buggy program and the passing and failing test cases, SearchRepair computes the Tarantula suspiciousness score for fragments in the following way:

1) **Compute initial suspiciousness.** SearchRepair uses the test suite to compute the suspiciousness score for each statement in the program. For the example in Figure 1, this initial computation assigns the maximal score of 1 to the `if` condition on line 9, because it is only executed by the failing test cases. The preceding predicate (line 7) in the `if-else` sequence has the score of 0.71; it is executed by the failing test cases and some of the passing test cases. Lines 2–5 and 11–12 are executed by all tests, and have the score of 0.63. Lines not executed by any failing tests (lines 6, 8, and 10) receive a score of 0.
2) **Identify the most suspicious statement(s).** SearchRepair identifies the line with the highest suspiciousness score and denotes it *pivot*. In our example, this is line 9. If multiple statements share the highest suspiciousness, SearchRepair designates multiple pivots and identifies suspicious fragments for each, treating each independently.
3) **Select fragments for replacement.** If pivot corresponds to a guarded statement, such as the predicate that controls a block or loop, SearchRepair will attempt to replace the entire guarded block as well as the preceding related control flow (in the case of multiple `else-if` constructs). In Figure 1, the pivot statement at line 9 identifies a candidate replacement fragment corresponding to the statements on lines 5–10.
   If the pivot does not identify such a predicate, SearchRepair will attempt to replace a set of fragments around the pivot. The size of the set is defined by a window size. For example, in our experiments in Section IV, we heuristically set the window size to no more than five lines, meaning that SearchRepair attempted to replace up to five different fragments: the pivot, the pivot with the following line, the pivot with the following two lines, and so on, until the window size of five. For cases in which the enclosing C code block was smaller than the window size, SearchRepair did not go beyond the code block size.
   There to-be-replaceD fragments preserve the granularity of the fragments encoded in the indexed repository, resulting in higher-granularity repairs than produced by prior repair techniques. We hypothesize this increase in granularity leads to higher-quality patches.

Our experimental results in Section IV show that this approach can often sufficiently accurately locate buggy fragments for the defects in our dataset.

D. Obtaining input-output profiles

The fault localization described in Section III-C identifies candidate code fragments as sites for repair. For each such identified code fragment, the next step is to extract variables and their values dynamically from the source code and its execution on the test cases. The goal is to collect, for each local variable, its *name*, *type*, and *value* both before and after the execution of the buggy code fragment. We leave consideration of global variables for future work. In the program from Figure 1, this requires identifying the values of `a`, `b`, `c`, and `median` before line 5 (input state) and after line 10 (output state). SearchRepair uses an unoptimized logging procedure to acquire these values in our experiments.

Figure 4 shows the profile for our example program from Figure 1 as executed on the test suite from Figure 2. Since the test cases t_1, t_2, and t_3 pass on the buggy program, the associated profile has three positive examples, one per test. The other test cases, t_4 and t_5, fail, so the associated input-output examples are negative.

E. Semantic search with an input-output profile

For each candidate buggy code fragment, SearchRepair uses the associated input-output profile to search the indexed repository of code fragments (recall Section III-B) for replacement fragments. The profiles characterize the desired behavior of the fragment; at a high level, the semantic search identifies functionally similar code, ideally without the defect in question. To find code that matches a specified profile, SearchRepair encodes each input-output example in the profile as SMT constraints and then uses the Z3 SMT solver [17] to find fragments in the database that satisfy them.

The first step in searching with an input-output profile is thus to transform input-output information into SMT constraints. Similar to the encoding used to construct the database of candidate replacement fragments (Section III-B), an input-output profile is described by constraints over the types and values of implicated variables.

To illustrate, consider a single input-output example that could describe code that computes the median of three numbers: {`int i = 2, int j = 3, int k = 4`} (*input*) and {`int med = 3`} (*output*). Such an input-output query can be translated into constraints as:

$$(\exists i,j,k,med : \texttt{int}) \wedge \\ (i=2) \wedge (j=3) \wedge (k=4) \wedge (med=3) \qquad (Q_{io})$$

Because we cannot assume consistent variable names, for each considered candidate database fragment, SearchRepair includes constraints encoding all possible mappings between the inputs and outputs of the profile and the candidate. Consider the example in Figure 5, whose input variables are `x`, `y`, and `z`. In some instances, including in this example, we can preemptively identify the fragment's output variable, if it is assigned last on every path (`m`, in this example; were this not the case, the number of possible mappings, and thus the mapping constraint, would be larger). The mapping constraints between the example input-output pair and this candidate fragment is:

$$(med=m) \wedge \\ (((i=x) \wedge (j=y) \wedge (k=z)) \vee ((i=x) \wedge (j=z) \wedge (k=y)) \vee \\ ((i=y) \wedge (j=x) \wedge (k=z)) \vee ((i=y) \wedge (j=z) \wedge (k=x)) \vee \\ ((i=z) \wedge (j=x) \wedge (k=y)) \vee ((i=z) \wedge (j=y) \wedge (k=x))) \qquad (\mathcal{M}_{io})$$

Thus, for each input-output pair *io*, and for each path *n*, in the fragment, the query to the SMT solver is:

$$\gamma \wedge \phi_n \wedge Q_{io} \wedge \mathcal{M}_{io} \qquad (search)$$

If at least one these queries is satisfiable for a particular fragment, the associated path *n* satisfies the constraints imposed by the input-output pair in question. Only one path per input-output pair needs to be satisfiable for the entire fragment to be considered a candidate patch.[1] When the query is satisfiable, the SMT solver produces a satisfiable model, which provides a suitable binding between input-output and fragment variables consistent with the constraint $\mathcal{M}_{io}$.

Extending this procedure to the multiple examples included in the profile of a candidate buggy region requires the separate encoding of each input-output pair. We define a code fragment as a *match*, or potential patch for a candidate faulty code region, if for each input state and output state pair corresponding to passing test cases, at least one path satisfies the specification (Section III-F describes other types of matches, such as partial matches). SearchRepair currently queries the SMT solver once per input-output example in a profile and requires variable mappings to be consistent between each example for a satisfying fragment to be considered a match.

SearchRepair currently identifies patches using exclusively the positive input-output examples because positive examples are more restrictive than negative examples. Intuitively, while relatively few code fragments satisfy the overall *search* query above (conceptually encoding "output must be the median of the three given inputs"), many more code fragments would satisfy a negative constraint such as "output may be anything except the observed incorrect output." In the absence of positive examples, the search is likely to return an intractable number of results. Instead, SearchRepair excludes patches that produce faulty behavior in the patch evaluation step, described next.

[1]Note that this is equivalent to a single disjunctive query that considers all paths in the candidate fragment, but supports precise functionality slicing when only one path in a candidate implements correct behavior, a full investigation of which we leave to future work.

F. Evaluating patches

For each *match* fragment returned by the search, SearchRepair constructs a patch that replaces the buggy fragment in the program with the matching fragment (the candidate fix code). The SMT solver produces a satisfying model (recall Section III-E) detailing how the variables in the input-output profile from the program state map to the variables in the patch. Using this mapping, SearchRepair performs variable renaming on the patch fragment to match the environment of in-scope variables for the code to be replaced.

SearchRepair applies each such patch to the buggy input program and reruns the full test suite, classifying the candidate patch as a *full repair*, a *partial repair*, or a *non-repair*:

Full repair: The patched program passes all of test cases. For example, the fragment in Figure 3(a) is a full repair for the program in Figure 1, since all test cases in Figure 2 pass when lines 5–10 are replaced with the repair.

Partial repair: The patched program passes all of the previously passing test cases, and passes a portion of previously failing test cases. For example, the fragment in Figure 3(b) is a *partial repair* for program in Figure 1 because when lines 5–10 are replaced with the repair, while t_1, t_2, and t_3 from Figure 2 continue to pass, and t_5 now passes, t_4 still fails.

Non-repair: The patched program fails all of the previously failing test cases, or fails at least one of the previously passing test cases.

The process of evaluating patches continues until either a full repair is found or all candidate repairs are evaluated. If a full repair is not found, SearchRepair can return a partial repair, as Section IV-B reports.

IV. Evaluation

This section describes SearchRepair's evaluation, including a comparison to three prior tools, GenProg [74], TrpAutoRepair [56], and AE [73]. Section IV-A describes our experimental set up. Sections IV-B and IV-C evaluate SearchRepair's effectiveness at producing repairs and the quality of those repairs, respectively. Section IV-D presents preliminary results into a fully automated approach for repairing defects without requiring the developer to write tests. Finally, Section IV-E discusses the threats to our experimental validity.

A. Experimental setup

We base our evaluation on the IntroClass benchmark. IntroClass is intended specifically for evaluating automatic program repair research [43]. IntroClass is available for download: `http://repairbenchmarks.cs.umass.edu/`. IntroClass consists of 998 defective versions of C programs submitted by around 200 students for six small C programming assignments in an introductory undergraduate course. All defects are made by the students as part of normal development practices while attempting to complete their assignments. These 998 versions are broken into two (overlapping) sets of defects, 778 that fail tests in instructor-written test suites, and 845 that fail tests in automatically-generated test suites. Our evaluation uses the set of 778 defects, described in Figure 6.

Each assignment asks students to write a C program according to a detailed specification. The students may, at any time and as often as they wish, commit a version of their code to a `git` repository. This saves a potentially defective version and runs, on a remote server, a set of instructor-written test cases for the assignment. The testing harness then reports to the student the number of test cases the program passed and

program	defects	instructor tests	KLEE tests	description
checksum	29	6	10	check sum of a string
digits	91	6	10	digits of a number
grade	226	9	9	grade from score
median	168	7	6	median of three numbers
smallest	155	8	8	smallest of four numbers
syllables	109	6	10	count vowels of a string
total	778	42	53	

Fig. 6: The IntroClass dataset, including the number of buggy versions of each assignment and the associated test suite sizes.

program	SearchRepair	AE	GenProg	TrpAutoRepair	total
checksum	0	0	8	0	29
digits	0	17	30	19	91
grade	5	2	2	2	226
median	68	58	108	93	168
smallest	73	71	120	119	155
syllables	4	11	19	14	109
total repaired	150	159	287	247	778

Fig. 7: Number of defects repaired by each technique. The total *column* specifies the total number of defects, and the total *row* specifies the total number of repaired defects.

failed, without providing details about the inputs and expected outputs of those programs. The students also may query an oracle implementation for the correct output on inputs they provide. The 778 defects are collected from these submitted versions by selecting versions that pass at least one and fail at least one instructor-written test.

The instructor-written test suite is constructed via careful consideration of the requirements and the input space. IntroClass also includes, for each program, a test suite automatically generated using KLEE [9] to achieve full branch coverage on an instructor-written oracle program. These additional test suites support independent high-coverage validation of program and patch correctness.

Our evaluation attempts to repair the 778 buggy versions identified by the instructor-written test suite, giving the repair tools access to that test suite. Test suites are known to be imperfect [60], and so a repair technique that uses them exclusively to evaluate partial correctness risks breaking undertested functionality if the resulting patches are insufficiently general. We use the KLEE test suite to independently validate patch correctness, evaluating the degree to which patches *overfit* to the input test suites, possibly breaking undertested functionality.

Prior work has released baseline experimental results for GenProg, AE, and TrpAutoRepair on *IntroClass* is to support comparative evaluations for new techniques [43], [65]. We use these experimental results in our comparative evaluation.

We indexed several repositories of code fragments for SearchRepair. Our overall experimental goal is to evaluate the feasibility of semantic-search-based program repair. For most experiments, we index the repository for a given defective student assignment out of the other students' buggy assignment submissions. We explicitly do not include correct submissions that pass all instructor-written tests. We also investigate a repair scenario with a database constructed of fragments from the Linux kernel source code (see Section IV-D). Each database has at minimum 2,000 fragments to consider for repair.

B. Repair effectiveness

Figure 7 shows how effective each of the four repair techniques, SearchRepair, GenProg, TrpAutoRepair, and AE, are at producing patches that pass all of the tests supplied to the repair technique. (We defer the evaluation of the quality of these patches until Section IV-C.) SearchRepair repairs 150 (19.3%) of the 778 defective student programs, compared to GenProg's 287 (36.9%), TrpAutoRepair's 247 (31.7%), and AE's 159 (20.4%).

SearchRepair did as well, and as well as the other techniques on `median` and `smallest`, outperformed the other techniques on `grade`, and produced some but not many repairs on `syllables`. There are two assignments for which SearchRepair was unable to produce any repairs, `checksum` and `digits`. The `checksum` assignment is challenging for all the techniques except GenProg. GenProg has the ability to combine multiple repairs over the course of an evolutionary search, and `checksum` functionality may require multi-edit repairs beyond what can be provided even at SearchRepair's higher granularity. However, Section IV-D describes how using a database of code fragments from the Linux kernel, SearchRepair was able to repair 18 of the `checksum` defects, 17 of which were not be repaired by any of the other three repair tools. SearchRepair performed poorly on the `digits` assignment because this assignment requires modeling of I/O operation beyond the capability of our constraint encoder. Extending the semantic search technique to encode and model such operations will increase SearchRepair's ability to handle a wider array of program constructs and thus, defects. We are encouraged by SearchRepair's success given the subset of C language constructs and operations it currently supports.

SearchRepair is complementary to the other repair techniques. It can repair 20 of the defects that the other three techniques do not repair. Figure 8 shows a Venn diagram describing the breakdown of which techniques repaired which defects. There are 310 total unique defects the tools were able to repair. Of these, 20 (6.5%) are unique to SearchRepair, 160 (51.6%) can be repaired by at least one other technique but not by SearchRepair, and 130 (41.9%) can be repaired by SearchRepair and at least one other technique.

These results suggest that SearchRepair may be complementary to these other previously-proposed generate-and-validate techniques. The fact that SearchRepair repairs 20 defects that the other tools do not suggests that, although SearchRepair does not repair *more* defects than the previous tools, it repairs at least some *different* defects.

SearchRepair can also automatically identify *partial repairs* that address some but not all of the defective behavior (recall Section III-F). In addition to the defects it repaired fully, SearchRepair also identified partial repairs for four of the `grade` programs.

C. Repair quality

Producing patches that address undesirable behavior is important, but does not fully address the issue of repair *quality*.

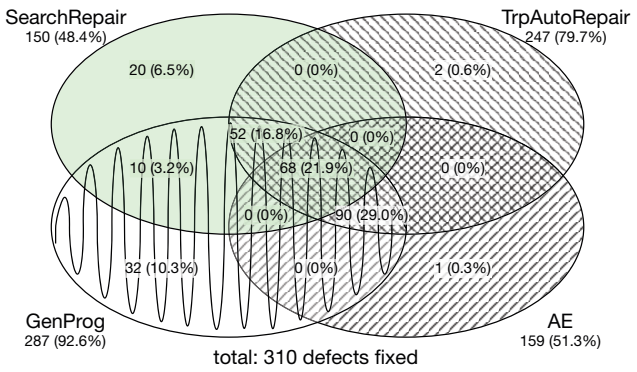

Fig. 8: SearchRepair is the only tool that can repair 20 (6.5%) of the 310 defects repaired by the four repair techniques. The other three repair tools can together repair 160 (51.6%) defects that SearchRepair cannot. The remaining 130 (41.9%) of the defects can be repaired by SearchRepair and at least one other tool. (Not shown in the diagram is that 35 (11.3%) of the defects can be repaired by both GenProg and TrpAutoRepair, and that 0 (0%) of the defects can be repaired by both SearchRepair and AE.)

One of our main hypotheses in developing SearchRepair is that program repair via search for human-written routines is likely to produce higher-quality repairs than prior work has been able to. We thus use the second, independent test suite that is not used by the repair process to assess and compare repair quality between techniques. If a repair passes more of the independent tests, then it generalizes better to the full specification of the program, and is thus of higher quality (recall Section II-C).

Figure 9 shows that the quality of the patches produced by SearchRepair is much higher than that of the other techniques. SearchRepair-produced patches pass 97.3% of the independent tests, whereas GenProg-, TrpAutoRepair-, and AE-produced patches pass only 68.7%, 72.1%, and 64.2% of those tests, respectively. This supports our hypothesis that using human-written code at a higher level of granularity than previously explored produces higher-quality, better-generalizing patches than do prior techniques.

D. Fully automated repair

Today, automated program repair requires the user to write a specification, such as a test suite. Because real-life test-suites are incomplete and imperfect [60], and poor-quality test suites pose a challenge to program repair [65].

Automated test input generation can perhaps alleviate the burden of writing extensive tests and may aid automated program repair, although this requires solving the problem of producing a test oracle, which we do not address here. We designed and ran a preliminary experiment to test if SearchRepair can be used in an automated manner to repair defects. This experiment had SearchRepair (1) use the KLEE-generated tests for the defects to simulate a scenario in which the user had written no tests, and (2) created and used an indexed database of a random subset of the Linux kernel code to use as candidate repairs. In this scenario, SearchRepair was able to repair 18 `checksum` defects, 17 of which, none of the other three tools repaired. This result suggests that SearchRepair is a powerful tool that can be used to search for repairs in large, open-source repositories and be made even more automated with the use of automated test generation.

SearchRepair	GenProg	TrpAutoRepair	AE
97.3%	68.7%	72.1%	64.2%

Fig. 9: The quality of the patches produced by the four repair techniques, as measured by the number of independent (not used for patch generation) tests the patched programs pass.

E. Threats to validity

IntroClass is composed of short, simple programs and our results may not generalize to more complex programs. We mitigate this threat to validity by evaluating on a large number of such programs, tested by systematically-designed test suites, programmed by actual novice developers making real mistakes. The experimental setting is appropriate for the constraints of our semantic search engine research prototype for C. The implementation of new constraint encodings for C language constructs is labor-intensive, and we are confident that adding new such constructs will improve SearchRepair's expressive power. For example, approximation and unrolling can both extend support to more powerfully encoding loops.

For most of our experiments, the repository is constructed of closely related programs, in the interest of supporting a scalable evaluation. For each student-written defect, we excluded from the repository that student's assignment solution to avoid giving the search unfairly correct code. Still, the artificial nature of the repository construction is a threat to the generality of our results. Our success on the one case study assignment using fragments scraped from the Linux kernel supports our hypothesis that the approach can generalize to broader datasets.

SearchRepair's repairs may be hard to maintain. Our over-fitting evaluation suggests that they more generally encode the desired requirements than the patches produced by competing techniques, but this measure is only a proxy for quality.

We assume the test cases are sound and complete. That is, we assume the tests fail when there is a bug and pass when there is not one. While, in general, this assumption is flawed, for the scope of the evaluation, it is reasonable. We mitigate the threat it poses by using two independent test suites to evaluate SearchRepair output. Applying SearchRepair to a broader context will likely require us to consider partial fixes when complete fixes cannot be found. In that case, the partial fix may be the product of the test suite, and not the code repository being searched over.

V. Related work

Code redundancy. Prior work has found that much of software is both syntactically and semantically redundant [5], [11], [12], [24]. A study of 6,000 software projects (over 420 million lines of code) found that large portions of most software projects are syntactically redundant [24]. Semantically, many methods can be reconstructed by composing other methods in the same project [5], [11], [12]. These findings are consistent with researchers' observations of code clones [37]. Intuitively, software projects repeatedly reuse the same common building block data structures and methods as other projects, and often reimplement this functionality. This suggests that if a project contains a method with a bug, other software projects

likely implement a bug-free version of the same or similar functionality that may be used to produce a repair.

Semantic Code Search. Recently keyword-based searches have begun to incorporate semantic information for finding working code examples from the Web [38] or reformulating queries for concept localization [30]. The search approach we use for program repair depends on a more structured specification. Prior semantic code search has used formal specifications [27], [53], [77] and test cases [55], [58]. Formal specifications allow precise and sound matching but must be written by hand, which is difficult and error-prone. Test cases are more lightweight but, prior to our work, required the code to be executed and could not identify partial matches.

Code Synthesis. Code can be synthesized using input-output examples, written in a domain-specific language [29], using predefined components [34], or based on a high-level behavioral description [4]. Recent approaches use context from a debugger to show where in a program synthesis should occur [26]. While effective in that domain, synthesis-based approaches are limited by the solver's ability to enumerate and test all possible combinations of program constructs. In our use of solvers for semantic search, we instead encode, search over, and return existing code.

Code transfer. SearchFix uses textual replacement with variable renaming to insert candidate patches into buggy programs. This relatively simple procedure is sufficient for the IntroClass dataset, but more complex programs will likely require deeper code transfer techniques. Recent advances in code transfer [6], [62], [63] are complementary to our work and we expect future versions of SearchRepair to use them.

Program repair. Automated program repair is concerned with automatically bringing an implementation more in line with its specification, typically by producing a patch that addresses a defect as exposed by a specification or a test case. Interest in this field has expanded substantially over the past decade to include at least twenty projects since 2009 that involve some form of repair (e.g., AE [73], AFix [35], ARC [7], Arcuri and Yao [3], ARMOR [11], and AutoFix-E [52], [70], Axis [48], BugFix [33], CASC [76], ClearView [54], Coker and Hafiz [15], Debroy and Wong [18], Demsky and Rinard [19], DirectFix [49], FINCH [51], GenProg [42], [74], Gopinath et al. [28], Jolt [10], Juzi [21], Kali [57], PACHIKA [16], PAR [39], *relifix* [69], SemFix [50], Sidiroglou and Keromytis [61], TrpAutoRepair [56], etc.). Section II-B has discussed the differences between generate-and-validate and correct-by-construction repair. The approach behind SearchRepair bridges the gap between these two repair approaches, in a similar vein as SemFix [50], which uses learned constraints to guide component-based synthesis of repair code. SearchRepair is more general in the types of defects it can repair than SemFix, which specifically targets defective predicates and defective assignments. SearchRepair can also use databases of human-written code, broadening the granularity of the changes that can be found and applied to beyond one line. This may impact readability, maintainability, or generalizability of the resulting code, as we observed in our overfitting metrics.

Another way to distinguish between prior work in automated repair is by the generality of the proposed techniques. Some approaches target a particular class of bugs, such as buffer overruns [64], unsafe integer use in C programs [15], single-variable atomicity violations [35], deadlock and livelock defects [46], concurrency errors [47], and data input errors [1]. Other techniques tackle generic bugs. For example, the ARMOR tool replaces buggy library calls with different calls that achieve the same behavior [11], and *relifix* uses a set of templates mined from regression fixes to automatically patch generic regression bugs [69]. Our evaluation has focused on tools that fix generic bugs, but our methodology can be applied to focused repair as well. Regardless, such techniques require a mechanism to specify desired behavior, whether those specifications are implicit (integer vulnerability patterns), explicit in the form of formal specifications or annotations (as in AutoFix-E [52], [70]), test cases (as in GenProg [42], TrpAutoRepair [56], and other test-case-based generate-and-validate approaches), or hybrids (as in SemFix [50]). SearchRepair falls into the latter category, learning specifications over test case behavior and using that observed behavior to guide repair. We have demonstrated that its particular approach is orthogonal to other prior approaches in this space, and that it may provide unique benefits in terms of the generality of its resulting repairs.

VI. Contributions

The high source code redundancy in modern software development practice means that it is very likely that a piece of code with a defect has been reimplemented correctly elsewhere. We proposed to take advantage of this observation by using advances in semantic code search — searching for code based on a behavioral specification as opposed to a keyword description — to automatically repair defective programs. We implemented our approach in SearchRepair, a technique that uses static analysis to build a searchable database of open-source code fragments that describes fragment behavior as a set of SMT constraints, and dynamic analysis to identify candidate faulty regions in a program and construct characteristic input-output behavior profiles. SearchRepair uses an SMT constraint solver to search over the database of code fragments for potential repairs, maps candidate repair fragments to a buggy context, and validates the candidate repair using test cases.

We hypothesized that using human-written code to construct patches and performing repair at a higher granularity level than prior work offers a unique advantage in creating repairs that generalize beyond the partial correctness specification encoded in a test suite. Our evaluation on 778 defects written by novice developers confirms our hypothesis. SearchRepair-repaired programs pass, on average, 97.3% of independent tests not used to construct the repair, whereas GenProg-, TrpAutoRepair-, and AE-repaired programs pass 68.7%, 72.1%, and 64.2% of the tests, respectively. Our results suggest that SearchRepair is complementary to prior approaches, repairing some defects those approaches cannot, and producing higher-quality repairs.

Overall, SearchRepair's results strongly suggest more research is warranted in semantic-search-based repair and that such approaches may produce patches that drastically outperform its counterpart techniques in terms of generalizing to program specifications.

Acknowledgments

This work is supported by the National Science Foundation under grants no. CCF-1446683, CCF-1446932, CCF-1446966, and CCF-1453508.

References

[1] Muath Alkhalaf, Abdulbaki Aydin, and Tevfik Bultan. Semantic differential repair for input validation and sanitization. In *International*

Symposium on Software Testing and Analysis (ISSTA), pages 225–236, San Jose, CA, USA, July 2014.

[2] John Anvik, Lyndon Hiew, and Gail C. Murphy. Coping with an open bug repository. In *Workshop on Eclipse Technology eXchange*, pages 35–39, San Diego, California, 2005.

[3] Andrea Arcuri and Xin Yao. A novel co-evolutionary approach to automatic software bug fixing. In *Congress on Evolutionary Computation*, pages 162–168, 2008.

[4] Marco Autili, Paola Inverardi, Alfredo Navarra, and Massimo Tivoli. SYNTHESIS: A tool for automatically assembling correct and distributed component-based systems. In *ACM/IEEE International Conference on Software Engineering (ICSE) Formal Demonstrations track*, pages 784–787, 2007.

[5] Earl T. Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. The plastic surgery hypothesis. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 306–317, Hong Kong, China, November 2014.

[6] Earl T. Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. Automated software transplantation. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 373–384, Baltimore, MD, USA, July 2015.

[7] Jeremy S. Bradbury and Kevin Jalbert. Automatic repair of concurrency bugs. In Massimiliano Di Penta, Simon Poulding, Lionel Briand, and John Clark, editors, *International Symposium on Search Based Software Engineering (SSBSE) fast abstract*, Benevento, Italy, September 2010.

[8] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen. Reversible debugging software. Technical report, University of Cambridge, Judge Business School, 2013.

[9] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 209–224, San Diego, CA, USA, 2008.

[10] Michael Carbin, Sasa Misailovic, Michael Kling, and Martin C. Rinard. Detecting and escaping infinite loops with Jolt. In *European Conference on Object Oriented Programming (ECOOP)*, Lancaster, England, UK, July 2011.

[11] Antonio Carzaniga, Alessandra Gorla, Andrea Mattavelli, Nicolò Perino, and Mauro Pezzè. Automatic recovery from runtime failures. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 782–791, San Francisco, CA, USA, 2013.

[12] Antonio Carzaniga, Alessandra Gorla, Nicolò Perino, and Mauro Pezzè. Automatic workarounds for web applications. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 237–246, Santa Fe, New Mexico, USA, 2010.

[13] Lori A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering (TSE)*, SE-2(3):215–222, September 1976.

[14] Lori A. Clarke and Debra J. Richardson. Applications of symbolic evaluation. *Journal of Systems and Software (JSS)*, 5(1):15–35, February 1985.

[15] Zack Coker and Munawar Hafiz. Program transformations to fix C integers. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 792–801, San Francisco, CA, USA, 2013.

[16] Valentin Dallmeier, Andreas Zeller, and Bertrand Meyer. Generating fixes from object behavior anomalies. In *IEEE/ACM International Conference on Automated Software Engineering (ASE) short paper track*, pages 550–554, Auckland, New Zealand, November 2009.

[17] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. 2008.

[18] Vidroha Debroy and W. Eric Wong. Using mutation to automatically suggest fixes for faulty programs. In *International Conference on Software Testing, Verification, and Validation*, pages 65–74, Paris, France, 2010.

[19] Brian Demsky, Michael D. Ernst, Philip J. Guo, Stephen McCamant, Jeff H. Perkins, and Martin Rinard. Inference and enforcement of data structure consistency specifications. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 233–243, Portland, ME, USA, July 2006.

[20] Thomas Durieux, Matias Martinez, Martin Monperrus, Romain Sommerard, and Jifeng Xuan. Automatic repair of real bugs: An experience report on the Defects4J dataset. *CoRR*, abs/1505.07002, 2015.

[21] Bassem Elkarablieh and Sarfraz Khurshid. Juzi: A tool for repairing complex data structures. In *ACM/IEEE International Conference on Software Engineering (ICSE) Formal Demonstration track*, pages 855–858, Leipzig, Germany, 2008.

[22] Stephanie Forrest. Genetic algorithms: Principles of natural selection applied to computation. *Science*, 261:872–878, August 1993.

[23] Zachary P. Fry, Bryan Landau, and Westley Weimer. A human study of patch maintainability. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 177–187, Minneapolis, MN, USA, July 2012.

[24] Mark Gabel and Zhendong Su. A study of the uniqueness of source code. In *Proceedings of the SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 147–156, Santa Fe, NM, USA, 2010.

[25] Mark Gabel and Zhendong Su. Testing mined specifications. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, Cary, NC, USA, 2012.

[26] Joel Galenson, Philip Reames, Rastislav Bodik, Björn Hartmann, and Koushik Sen. CodeHint: Dynamic and interactive synthesis of code snippets. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 653–663, Hyderabad, India, 2014.

[27] Carlo Ghezzi and Andrea Mocci. Behavior model based component search: An initial assessment. In *Workshop on Search-driven Development: Users, Infrastructure, Tools, and Evaluation (SUITE)*, pages 9–12, Cape Town, South Africa, 2010.

[28] Divya Gopinath, Muhammad Zubair Malik, and Sarfraz Khurshid. Specification-based program repair using SAT. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 173–188, Saarbrücken, Germany, March 2011.

[29] Sumit Gulwani, Vijay Anand Korthikanti, and Ashish Tiwari. Synthesizing geometry constructions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 50–61, San Jose, CA, USA, 2011.

[30] Sonia Haiduc, Giuseppe De Rosa, Gabriele Bavota, Rocco Oliveto, Andrea De Lucia, and Andrian Marcus. Query quality prediction and reformulation for source code search: The refoqus tool. In *International Conference on Software Engineering (ICSE) Formal Demonstrations Track*, pages 1307–1310, San Francisco, CA, USA, 2013.

[31] Mark Harman. The current state and future of search based software engineering. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 342–357, 2007.

[32] Pieter Hooimeijer and Westley Weimer. Modeling bug report quality. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 34–43, 2007.

[33] Dennis Jeffrey, Min Feng, Neelam Gupta, and Rajiv Gupta. BugFix: A learning-based tool to assist developers in fixing bugs. In *International Conference on Program Comprehension (ICPC)*, pages 70–79, Vancouver, BC, Canada, May 2009.

[34] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 215–224, Cape Town, South Africa, 2010.

[35] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. Automated atomicity-violation fixing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 389–400, San Jose, CA, USA, 2011.

[36] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *International Conference on Software Engineering (ICSE)*, pages 467–477, Orlando, FL, USA, 2002.

[37] Cory J. Kapser and Michael W. Godfrey. "cloning considered harmful" considered harmful: Patterns of cloning in software. *Empirical Software Engineering*, 13(6):645–692, December 2008.

[38] Iman Keivanloo, Juergen Rilling, and Ying Zou. Spotting working code examples. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 664–675, Hyderabad, India, 2014.

[39] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 802–811, San Francisco, CA, USA, 2013.

[40] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[41] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[42] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *AMC/IEEE International Conference on Software Engineering (ICSE)*, pages 3–13, Zurich, Switzerland, 2012.

[43] Claire Le Goues, Neal Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. The ManyBugs and IntroClass benchmarks for automated repair of C programs. *IEEE Transactions on Software Engineering (TSE)*, 2015.

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

[45] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 141–154, San Diego, CA, USA, 2003.

[46] Yiyan Lin and Sandeep S. Kulkarni. Automatic repair for multi-threaded programs with deadlock/livelock using maximum satisfiability. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 237–247, San Jose, CA, USA, July 2014.

[47] Peng Liu, Omer Tripp, and Charles Zhang. Grail: Context-aware fixing of concurrency bugs. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 318–329, Hong Kong, China, November 2014.

[48] Peng Liu and Charles Zhang. Axis: Automatically fixing atomicity violations through solving control constraints. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 299–309, Zurich, Switzerland, 2012.

[49] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. DirectFix: Looking for simple program repairs. In *International Conference on Software Engineering (ICSE)*, Florence, Italy, May 2015.

[50] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. SemFix: Program repair via semantic analysis. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 772–781, San Francisco, CA, USA, 2013.

[51] Michael Orlov and Moshe Sipper. Flight of the FINCH through the Java wilderness. *IEEE Transactions on Evolutionary Computation*, 15(2):166–182, April 2011.

[52] Yu Pei, Carlo A. Furia, Martin Nordio, Yi Wei, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. *IEEE Transactions on Software Engineering (TSE)*, 40(5):427–449, 2014.

[53] John Penix and Perry Alexander. Efficient specification-based component retrieval. *Automated Software Engineering*, 6:139–170, April 1999.

[54] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically patching errors in deployed software. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 87–102, Big Sky, MT, USA, October 12–14, 2009.

[55] Andy Podgurski and Lynn Pierce. Retrieving reusable software by sampling behavior. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2:286–303, July 1993.

[56] Yuhua Qi, Xiaoguang Mao, and Yan Lei. Efficient automated program repair through fault-recorded testing prioritization. In *International Conference on Software Maintenance (ICSM)*, pages 180–189, Eindhoven, The Netherlands, September 2013.

[57] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 24–36, Baltimore, MD, USA, 2015.

[58] Steven P. Reiss. Semantics-based code search. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 243–253, Vancouver, BC, Canada, 2009.

[59] Manos Renieris and Steven Reiss. Fault localization with nearest neighbor queries. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 30–39, Montreal, Québec, Canada, October 2003.

[60] Research Triangle Institute. The economic impacts of inadequate infrastructure for software testing. NIST Planning Report 02-3, May 2002.

[61] Stelios Sidiroglou and Angelos D. Keromytis. Countering network worms through automatic patch generation. *IEEE Security and Privacy*, 3(6):41–49, November 2005.

[62] Stelios Sidiroglou-Douskos, Eli Davis, and Martin Rinard. Horizontal code transfer via program fracture and recombination. Technical Report MIT-CSAIL-TR-2015-012, MIT Computer Science and Artificial Intelligence Laboratory, 2015.

[63] Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. Automatic error elimination by horizontal code transfer across multiple applications. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 43–54, Portland, OR, USA, 2015.

[64] Alexey Smirnov and Tzi cker Chiueh. Dira: Automatic detection, identification and repair of control-hijacking attacks. In *Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, February 2005.

[65] Edward K. Smith, Earl Barr, Claire Le Goues, and Yuriy Brun. Is the cure worse than the disease? Overfitting in automated program repair. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, Bergamo, Italy, September 2015.

[66] Kathryn T. Stolee. *Solving the Search for Source Code*. PhD thesis, University of Nebraska, Lincoln, Lincoln, NE, USA, August 2013.

[67] Kathryn T. Stolee and Sebastian Elbaum. Toward semantic search via SMT solver. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE) New Ideas and Emerging Results Track*, pages 25:1–25:4, Cary, NC, USA, 2012.

[68] Kathryn T. Stolee, Sebastian Elbaum, and Daniel Dobos. Solving the search for source code. *ACM Transactions on Software Engineering Methodology*, 23(3):26:1–26:45, May 2014.

[69] Shin Hwei Tan and Abhik Roychoudhury. relifix: Automated repair of software regressions. In *International Conference on Software Engineering (ICSE)*, Florence, Italy, 2015.

[70] Yi Wei, Yu Pei, Carlo A. Furia, Lucas S. Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 61–72, Trento, Italy, 2010.

[71] Westley Weimer. Patches as better bug reports. In *International Conference on Generative Programming and Component Engineering (GPCE)*, pages 181–190, Portland, OR, USA, 2006.

[72] Westley Weimer, Stephanie Forrest, Claire Le Goues, and ThanhVu Nguyen. Automatic program repair with evolutionary computation. *Communications of the ACM Research Highlight*, 53(5):109–116, May 2010.

[73] Westley Weimer, Zachary P. Fry, and Stephanie Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Palo Alto, CA, USA, 2013.

[74] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 364–374, Vancouver, BC, Canada, 2009.

[75] Cathrin Weiss, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. How long will it take to fix this bug? In *International Workshop on Mining Software Repositories*, Minneapolis, MN, USA, 2007.

[76] Josh L. Wilkerson, Daniel R. Tauritz, and James M. Bridges. Multi-objective coevolutionary automated software correction. In *Conference on Genetic and Evolutionary Computation (GECCO)*, pages 1229–1236, Philadelphia, PA, USA, 2012.

[77] Amy Moormann Zaremski and Jeannette M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6:333–369, October 1997.