SOSRepair: Expressive Semantic Search for Real-World Program Repair

Afsoon Afzal, Manish Motwani, Kathryn T. Stolee, *Member, IEEE*, Yuriy Brun, *Senior Member, IEEE*, and Claire Le Goues, *Member, IEEE*

Abstract—Automated program repair holds the potential to significantly reduce software maintenance effort and cost. However, recent studies have shown that it often produces low-quality patches that repair some but break other functionality. We hypothesize that producing patches by replacing likely faulty regions of code with semantically-similar code fragments, and doing so at a higher level of granularity than prior approaches can better capture abstraction and the intended specification, and can improve repair quality. We create SOSRepair, an automated program repair technique that uses semantic code search to replace candidate buggy code regions with behaviorally-similar (but not identical) code written by humans. SOSRepair is the first such technique to scale to real-world defects in real-world systems. On a subset of the ManyBugs benchmark of such defects, SOSRepair produces patches for 22 (34%) of the 65 defects, including 3, 5, and 6 defects for which previous state-of-the-art techniques Angelix, Prophet, and GenProg do not, respectively. On these 22 defects, SOSRepair produces more patches (9, 41%) that pass all independent tests than the prior techniques. We demonstrate a relationship between patch granularity and the ability to produce patches that pass all independent tests. We then show that fault localization precision is a key factor in SOSRepair's success. Manually improving fault localization allows SOSRepair to patch 23 (35%) defects, of which 16 (70%) pass all independent tests. We conclude that (1) higher-granularity, semantic-based patches can improve patch quality, (2) semantic search is promising for producing high-quality real-world defect repairs, (3) research in fault localization can significantly improve the quality of program repair techniques, and (4) semi-automated approaches in which developers suggest fix locations may produce high-quality patches.

Index Terms—Automated program repair, semantic code search, patch quality, program repair quality, SOSRepair

1 Introduction

Automated program repair techniques (e.g., [8], [15], [16], [19], [20], [39], [44], [45], [46], [49], [55], [58], [59], [64], [79], [91], [94], [97], [110], [112]) aim to automatically produce software patches that fix defects. For example, Facebook uses two automated program repair tools, SapFix and Getafix, in their production pipeline to suggest bug fixes [60], [83]. The goal of automated program repair techniques is to take a program and a suite of tests, some of which that program passes and some of which it fails, and to produce a patch that makes the program pass all the tests in that suite. Unfortunately, these patches can repair some functionality encoded by the tests, while simultaneously breaking other, undertested functionality [85]. Thus, *quality* of the resulting patches is a critical concern. Recent results suggest that patch overfitting—patches that pass a particular set of test cases supplied to the program repair tool but fail to generalize to the desired specification—is common [47], [57], [76], [85]. The central goal of this work is to improve the ability of automated program repair to produce high-quality patches on real-world defects.

We hypothesize that producing patches by (1) replacing likely faulty regions of code with semantically-similar code fragments, and (2) doing so at a higher level of granularity than prior approaches can improve repair quality. There are two underlying reasons for this hypothesis:

1) The observation that human-written code is highly redundant [4], [13], [14], [25], [61], suggesting that, for many buggy code regions intended to implement some functionality, there exist other code fragments that seek to implement the same functionality, and at least one does so correctly.
2) Replacing code at a high level of granularity (e.g., blocks of 3–7 consecutive lines of code) corresponds to changes at a higher level of abstraction, and is thus more likely to produce patches that correctly capture the implied, unwritten specifications underlying desired behavior than low-level changes to tokens or individual lines of code.

For example, suppose a program has a bug in a loop that is intended to sort an array. First, consider another, semantically similar loop, from either the same project, or some other software project. The second loop is semantically similar to the buggy loop because, like the buggy loop, it sorts some

- A. Afzal and C. Le Goues are with the School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213. E-mail: {afsoona, clegoues}@cs.cmu.edu.
- M. Motwani and Y. Brun are with the College of Information and Computer Sciences, University of Massachusetts Amherst, Amherst, MA 01003-9264. E-mail: {mmotwani, brun}@cs.umass.edu.
- K. T. Stolee is with the Department of Computer Science, North Carolina State University, Raleigh, NC 27695-8206. E-mail: ktstolee@ncsu.edu.

Manuscript received 21 Dec. 2018; revised 21 June 2019; accepted 13 Aug. 2019. Date of publication 0 . 0000; date of current version 0 . 0000.
(Corresponding author: Afsoon Afzal.)
Recommended for acceptance by E. Bodden.
Digital Object Identifier no. 10.1109/TSE.2019.2944914

arrays correctly. At the same time, the second loop may not be semantically identical to the buggy loop, especially on the inputs that the buggy loop mishandles. We may not know a priori if the second, similar loop is correct. However, sorting is a commonly implemented subroutine. If we try to replace the buggy code with several such similar loops, at least one is likely to correctly sort arrays, allowing the program to pass the test cases it previously failed. In fact, the high redundancy present in software source code suggests such commonly implemented subroutines are frequent [4], [13], [14], [25]. Second, we posit that replacing the entire loop with a similar one is more likely to correctly encode sorting than what could be achieved by replacing a + with a -, or inserting a single line of code in the middle of a loop.

Our earlier work on semantic-search-based repair [38] presented one instance that demonstrated that higher-granularity, semantic-based changes can, in fact, improve quality. On short, student-written programs, on average, SearchRepair patches passed 97.3% of independent tests not used during patch construction. Meanwhile, the relatively lower-granularity patches produced by GenProg [49], TrpAutoRepair [75], and AE [107] passed 68.7, 72.1, and 64.2%, respectively [38]. Unfortunately, as we describe next, SearchRepair cannot apply to large, real-world programs.

This paper presents SOSRepair, a novel technique that uses input-output-based semantic code search to automatically find and contextualize patches to fix real-world defects. SOSRepair locates likely buggy code regions, identifies similarly-behaving fragments of human-written code, and then changes the context of those fragments to fit the buggy context and replace the buggy code. Semantic code search techniques [77], [88], [89], [90] find code based on a specification of desired behavior. For example, given a set of input-output pairs, semantic code search looks for code fragments that produce those outputs on those inputs. Constraint-based semantic search [88], [89], [90] can search for partial, non-executable code snippets. It is a good fit for automated program repair because it supports searching for code fragments that show the same behavior as a buggy region on initially passing tests, while looking for one that passes previously-failing tests as well.

While SOSRepair builds on the ideas from SearchRepair [38], to make SOSRepair apply, at scale, to real-world defects, we redesigned the entire approach and developed a conceptually novel method for performing semantic code search. The largest program SearchRepair has repaired is a 24-line C program written by a beginner programmer to find the median of three integers [38]. By contrast, SOSRepair patches defects made by professional developers in real-world, multi-million-line C projects. Since SearchRepair cannot run on these real-world defects, we show that SOSRepair outperforms SearchRepair on the IntroClass benchmark of small programs.

We evaluate SOSRepair on 65 real-world defects of 7 large open-source C projects from the ManyBugs benchmark [48]. SOSRepair produces patches for 22 defects, including 1 that has not been patched by prior techniques (Angelix [64], Prophet [58], and GenProg [49]). We evaluate patch quality using held-out independent test suites [85]. Of the 22 defects for which SOSRepair produces patches, 9 (41%) pass all the held-out tests, which is more than the prior techniques produce for these defects. On small C programs in the IntroClass benchmark [48], SOSRepair generates 346 patches, more than SearchRepair [38], GenProg [49], AE [108], and TrpAutoRepair [75]. Of those patches, 239 pass all held-out tests, again, more than the prior techniques.

To make SOSRepair possible, we make five major contributions to both semantic code search and program repair:

1) *A more-scalable semantic search query encoding.* We develop a novel, efficient, general mechanism for encoding semantic search queries for program repair, inspired by input-output component-based program synthesis [35]. This encoding efficiently maps the candidate fix code to the buggy context using a single query over an arbitrary number of tests. By contrast, SearchRepair [38] required multiple queries to cover all test profiles and failed to scale to large code databases or queries covering many possible permutations of variable mappings. Our new encoding approach provides a significant speedup over the prior approach, and we show that the speedup grows with query complexity.
2) *Expressive encoding capturing real-world program behavior.* To apply semantic search to real-world programs, we extend the state-of-the-art constraint encoding mechanism to handle real-world C language constructs and behavior, including structs, pointers, multiple output variable assignments, console output, loops, and library calls.
3) *Search for patches that insert and delete code.* Prior semantic-search-based repair could only *replace* buggy code with candidate fix code to affect repairs [38]. We extend the search technique to encode deletion and insertion.
4) *Automated, iterative search query refinement encoding negative behavior.*We extend the semantic search approach to include negative behavioral examples, making use of that additional information to refine queries. We also propose a novel, iterative, counterexample-guided search-query refinement approach to repair buggy regions that are not covered by the passing test cases. When our approach encounters candidate fix code that fails to repair the program, it generates new undesired behavior constraints from the new failing executions and refines the search query, reducing the search space. This improves on prior work, which could not repair buggy regions that no passing test cases execute [38].
5) *Evaluation and open-source implementation.* We implement and release SOSRepair (`https://github.com/squaresLab/SOSRepair`), which reifies the above mechanisms. We evaluate SOSRepair on the ManyBugs benchmark [48] commonly used in the assessment of automatic patch generation tools (e.g., [58], [64], [75], [107]). These programs are four orders of magnitude larger than the benchmarks previously used to evaluate semantic-search-based repair [38]. We show that, as compared to previous techniques applied to these benchmarks (Angelix [64], Prophet [58], and GenProg [49]), SOSRepair patches one defect none

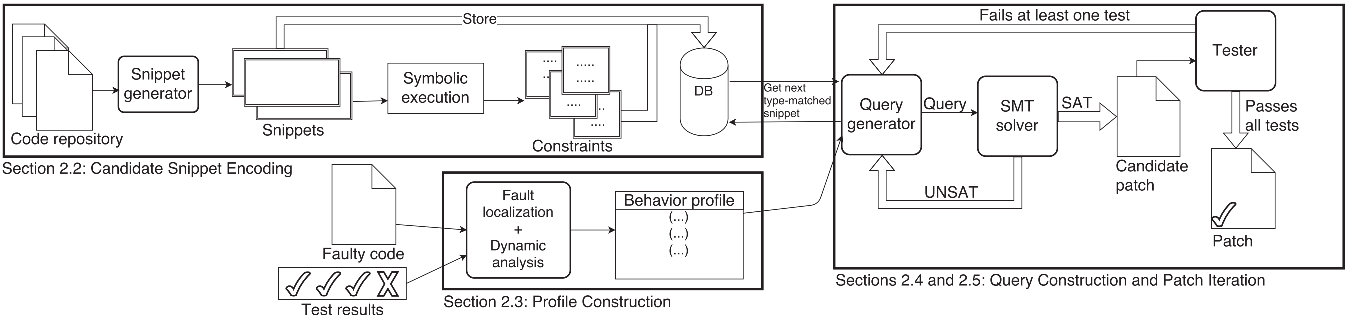

Fig. 1. Overview of the SOSRepair approach.

of those techniques patch, and produces patches of comparable quality to those techniques. We measure quality objectively, using independent test suites held out from patch generation [85]. We therefore also release independently-generated held-out test suites (https://github.com/squaresLab/SOSRepair-Replication-Package) for the defects we use to evaluate SOSRepair.

Based on our experiments, we hypothesize that fault localization's imprecision on real-world defects hampers SOSRepair. We create SOSRepair$^{\oplus}$, a semi-automated version of SOSRepair that is manually given the code location in which a human would repair the defect. SOSRepair$^{\oplus}$ produces patches for 23 defects. For 16 (70%) of the defects, the produced patches pass all independent tests. Thus, SOSRepair$^{\oplus}$ is able to produce high-quality patches for twice the number of defects than SOSRepair produces (16 versus 9). This suggests that semantic code search holds promise for producing high-quality repairs for real-world defects, perhaps in a semi-automated setting in which developers suggest code locations to attempt fixing. Moreover, advances in automated fault localization can directly improve automated repair quality.

To directly test the hypothesis that patch granularity affects the ability to produce high-quality patches, we alter the granularity of code SOSRepair can replace when producing patches, allowing for replacements of 1 to 3 lines, 3 to 7 lines, or 6 to 9 lines of code. On the IntroClass benchmark, using the 3–7-line granularity results in statistically significantly more patches (346 for 3–7-, 188 for 1–3-, and 211 for 6–9-line granularities) and statistically significantly more patches that pass all the held-out tests (239 for 3–7-, 120 for 1–3-, and 125 for 6–9-line granularities).

The rest of this paper is organized as follows. Section 2 describes the SOSRepair approach and Section 3 our implementation of that approach. Section 4 evaluates SOSRepair. Section 5 places our work in the context of related research, and Section 6 summarizes our contributions.

```
1  // len holds current position in stream
2  while (len < maxlen) {
3    php_stream_fill_read_buffer(stream,
4         len + MIN(maxlen- len chunk_size));
5    just_read =
6      (stream->writepos - stream->readpos)-len;
7  - if (just_read < toread) {
8  + if (just_read == 0) {
9      break;
10   } else {
11     len = len + just_read;
12   }
13 }
```

```
1  if (bufflen > 0)
2    mylen += bufflen;
3  else break;
```

Fig. 2. Top: Example code, based on php bug # 60455, in function stream_get_record. The developer patch modifies the condition on line 7, shown on line 8. Bottom: A snippet appearing in the php date module, implementing the same functionality as the developer patch (note that just_read is never negative in this code), with different variable names.

2 THE SOSREPAIR APPROACH

Fig. 1 overviews the SOSRepair approach. Given a program and a set of test cases capturing correct and buggy behavior, SOSRepair generates patches by searching over a database of snippets of human-written code. Unlike keyword or syntactic search (familiar to users of standard search engines), *semantic* search looks for code based on a specification of desired and undesired behavior. SOSRepair uses test cases to construct a behavioral profile of a potentially buggy code region. SOSRepair then searches over a database of snippets for one that implements the inferred desired behavior, adapts a matching snippet to the buggy code's context, and patches the program by replacing the buggy region with patch code, inserting patch code, or deleting the buggy region. Finally, SOSRepair validates the patched program by executing its test cases.

We first describe an illustrative example and define key concepts (Section 2.1). We then detail SOSRepair's approach that (1) uses symbolic execution to produce static behavioral approximations of a set of candidate bug repair snippets (Section 2.2), (2) constructs a dynamic profile of potentially-buggy code regions, which serve as inferred input-output specifications of desired behavior (Section 2.3), (3) constructs an SMT query to identify candidate semantic repairs to be transformed into patches and validated (Section 2.4), and (4) iteratively attempts to produce a patch until timeout occurs (Section 2.5). This section focuses on the conceptual approach; Section 3 will describe implementation details.

2.1 Illustrative Example and Definitions

Consider the example patched code in Fig. 2 (top), which we adapt (with minor edits for clarity and exposition) from php interpreter bug issue #60455, concerning a bug in the streams API.[1] Bug #60455 reports that streams mishandles

1. https://bugs.php.net/bug.php?id=60455

files when the EOF character is on its own line. The fixing commit message elaborates: "stream_get_line misbehaves if EOF is not detected together with the last read." The change forces the loop to continue such that the last EOF character is consumed. The logic that the developer used to fix this bug is not unique to the `stream_get_record` function; indeed, very similar code appears in the `php date` module (bottom of Fig. 2). This is not unusual: there exists considerable redundancy within and across open-source repositories [4], [25], [33], [96].

Let $\mathcal{F}$ refer to a code snippet of 3–7 lines of C code. $\mathcal{F}$ can correspond to either the buggy region to be replaced or a snippet to be inserted as a repair. In our example bug, a candidate buggy to-be-replaced region is lines 7–11 in top of Fig. 2; the snippet in the bottom of Fig. 2 could serve as a repair snippet. We focus on snippets of size 3–7 lines of code because patches at a granularity level greater than single-expression, -statement, or -line may be more likely to capture developer intuition, producing more-correct patches [38], but code redundancy drops off sharply beyond seven lines [25], [33]. We also verify these findings by conducting experiments that use code snippets of varying sizes (Section 4.3).

$\mathcal{F}$'s *input* variables f are those whose values can ever be *used* (in the classic dataflow sense, either in a computation, assignment, or predicate, or as an argument to a function call); $\mathcal{F}$'s *output* variables $\vec{R_f}$ are those whose value may be *defined* with a definition that is not *killed* by the end of the snippet. In the buggy region of Fig. 2, f is $\{\texttt{just\_read}, \texttt{toread}, \texttt{len}\}$; $\vec{R_f}$ is $\{\texttt{len}\}$. $\vec{R_f}$ may be of arbitrary size, and f and $\vec{R_f}$ are not necessarily disjoint, as in our example. $\vec{\mathcal{V}_f}$ is the set of all variables of interest in $\mathcal{F}$: $\vec{\mathcal{V}_f} = f \cup \vec{R_f}$.

To motivate a precise delineation between variable uses and definitions, consider a concrete example that demonstrates correct behavior for the buggy code in Fig. 2: if `just_read` = 5 and `len` = 10 after line 6, at line 12, it should be the case that `just_read` = 5 and `len` = 15. A naive, constraint-based expression of this desired behavior, e.g., $(\texttt{just\_read} = 5) \wedge (\texttt{len} = 10) \wedge (\texttt{just\_read} = 5) \wedge (\texttt{len} = 15)$ is unsatisfiable, because of the conflicting constraints on `len`.

For the purposes of this explanation, we first address the issue by defining a static variable renaming transformation over snippets. Let $U_f(x)$ return all uses of a variable x in $\mathcal{F}$ and $D_f(x)$ return all definitions of x in $\mathcal{F}$ that are not killed. We transform arbitrary $\mathcal{F}$ to enforce separation between inputs and outputs as follows:

$$\mathcal{F}' = F[U_f(x)/x_i] \ s.t. x \in V_f, x_i \in X_{in}, x_i \ fresh$$
$$\mathcal{F}_t = F'[D_f(x)/x_i] \ s.t. x \in R_f, x_i \in X_{out}, x_i \ fresh.$$

All output variables are, by definition, treated also as inputs, and we choose fresh names as necessary. X_{in} and X_{out} refer to the sets of newly-introduced variables.

2.2 Candidate Snippet Encoding

In an offline pre-processing step, we prepare a database of candidate repair snippets of 3–7 lines of C code. This code can be from any source, including the same project, its previous versions, or open-source repositories. A naive lexical approach to dividing code into line-based snippets generates many implausible and syntactically invalid snippets, such as by crossing block boundaries (e.g., lines 10–12 in the top of Fig. 2). Instead, we identify candidate repair snippets from C blocks taken from the code's abstract syntax tree (AST). Blocks of length 3–7 lines are treated as a single snippet. Blocks of length less than 3 lines are grouped with adjacent blocks. We transform all snippets $\mathcal{F}$ into $\mathcal{F}_t$ (Section 2.1). In addition to the code itself (pre- and post- transformation) and the file in which it appears, the database stores two types of information per snippet:

1) *Variable names and types.* Patches are constructed at the AST level, and are thus always syntactically valid. However, they can still lead to compilation errors if they reference out-of-scope variable names, user-defined types, or called functions. We thus identify and store names of user-defined structs and called functions (including the file in which they are defined). We additionally store all variable names from the *original* snippet $\mathcal{F}$ ($\vec{\mathcal{V}_f}$, f, $\vec{R_f}$), as well as their corresponding renamed versions in $\mathcal{F}_t$ (X_{in} and X_{out}).
2) *Static path constraints.* We symbolically execute [12], [40] $\mathcal{F}_t$ to produce a symbolic formula that statically overapproximates its behavior, described as constraints over snippet input and outputs. For example, the fix snippet in Fig. 2 can be described as

$$((\texttt{bufflen}_{in} > 0) \wedge (\texttt{mylen}_{out} = \texttt{mylen}_{in} + \texttt{bufflen}_{in})) \vee$$
$$(\neg(\texttt{bufflen}_{in} > 0) \wedge (\texttt{mylen}_{out} = \texttt{mylen}_{in})).$$

We query an SMT solver to determine whether such constraints match desired inputs and outputs.

The one-time cost of database construction is amortized across many repair efforts.

2.3 Profile Construction

SOSRepair uses spectrum-based fault localization (SBFL) [37] to identify candidate buggy code regions. SBFL uses test cases to rank program entities (e.g., lines) by suspiciousness. We expand single lines identified by SBFL to the enclosing AST block. Candidate buggy regions may be smaller than 3 lines if no region of fewer than 7 lines can be created by combining adjacent blocks.

Given a candidate buggy region $\mathcal{F}$, SOSRepair constructs a *dynamic profile* of its behavior on passing and failing tests. Note that the profile varies by the type of repair, and that SOSRepair can either *delete* the buggy region; *replace* it with a candidate repair snippet; or *insert* a piece of code immediately before it. We discuss how SOSRepair iterates over and chooses between repair strategies in Section 2.5. Here, we describe profile generation for replacement and insertion (the profile is not necessary for deletion).

SOSRepair first statically substitutes $\mathcal{F}_t$ for $\mathcal{F}$ in the buggy program, declaring fresh variables X_{in} and X_{out}. SOSRepair then executes the program on the tests, capturing the values of all local variables before and after the region on all covering test cases. (For simplicity and without loss of generality, this explanation assumes that all test executions cover all input and output variables.) Let T_p be the set of all initially passing tests that cover $\mathcal{F}_t$ and T_n the set of all initially failing tests that do so. If t is a test case covering $\mathcal{F}_t$, let $valIn(t, x)$ be the observed dynamic value of x on test case t before $\mathcal{F}_t$ is executed and $valOut(t,x)$ its dynamic value afterwards. We index each observed value of each variable of interest x by the test execution on which the value is observed, denoted x^t. This allows us to specify desired behavior based on multiple test executions or behavioral examples at once. To illustrate, assume a second passing execution of the buggy region in Fig. 2 on which `len` is 15 on line 6 and 25 on line 12 (ignoring `just_read` for brevity). $\big((\texttt{len}_{\texttt{in}} = 10) \wedge (\texttt{len}_{\texttt{out}} = 15)\big) \wedge \big((\texttt{len}_{\texttt{in}} = 15) \wedge (\texttt{len}_{\texttt{out}} = 25)\big)$ is trivially unsatisfiable; $\big((\texttt{len}_{\texttt{in}}{}^1 = 10) \wedge (\texttt{len}_{\texttt{out}}{}^1 = 15)\big) \wedge \big((\texttt{len}_{\texttt{in}}{}^2 = 15) \wedge (\texttt{len}_{\texttt{out}}{}^2 = 25)\big)$, which indexes the values by the tests on which they were observed, is not. The dynamic profile is then defined as follows:

$$P := P_{in} \wedge P^p_{out} \wedge P^n_{out}.$$

P_{in} encodes bindings of variables to values on entry to the candidate buggy region on all test cases; P^p_{out} enforces the desired behavior of output variables to match that observed on initially passing test cases; P^n_{out} enforces that the output variables should *not* match to those observed on initially failing test cases. P_{in} is the same for both replacement and insertion profiles

$$P_{in} := \bigwedge_{t \in T_p \cup T_n} \bigwedge_{x_i \in X_{in}} x^t_i = valIn(t, x_i).$$

P_{out} combines constraints derived from both passing and failing executions, or $P^p_{out} \wedge P^n_{out}$. For replacement queries

$$P^p_{out} := \bigwedge_{t \in T_p} \bigwedge_{x_i \in X_{out}} x^t_i = valOut(t, x_i)$$
$$P^n_{out} := \bigwedge_{t \in T_n} \neg \left(\bigwedge_{x_i \in X_{out}} x^t_i = valOut(t, x_i) \right).$$

For insertion queries, the output profile specifies that the correct code should simply preserve observed passing behavior while making some observable change to initially failing behavior

$$P^p_{out} := \bigwedge_{t \in T_p} \bigwedge_{x_i \in X_{out}} x^t_i = valIn(t, x_i)$$
$$P^n_{out} := \bigwedge_{t \in T_n} \neg \left(\bigwedge_{x_i \in X_{out}} x^t_i = valIn(t, x_i) \right).$$

Note that we neither know, nor specify, the correct value for these variables on such failing tests, and do not require annotations or developer interaction to provide them such that they may be inferred.

2.4 Query Construction

Assume candidate buggy region $\mathcal{C}$ (a *context* snippet), candidate repair snippet $\mathcal{S}$, and corresponding input variables, output variables, etc. (as described in Section 2.1). Our goal is to determine whether the repair code $\mathcal{S}$ can be used to edit the buggy code, such that doing so will possibly address the buggy behavior without breaking previously-correct behavior. This task is complicated by the fact that candidate repair snippets may implement the desired behavior, but use the wrong variable names for the buggy context (such as in our example in Fig. 2). We solve this problem by constructing a single SMT query for each pair of $\mathcal{C}, \mathcal{S}$, that identifies whether a mapping exists between their variables ($\vec{\mathcal{V}}_c$ and $\vec{\mathcal{V}}_s$) such that the resulting patched code ($\mathcal{S}$ either substituted for or inserted before $\mathcal{C}$) satisfies all the profile constraints P. An important property of this query is that, if satisfiable, the satisfying model provides a variable mapping that can be used to rename $\mathcal{S}$ to fit the buggy context.

The repair search query is thus comprised of three constraint sets: (1) mapping components ψ_{map} and ψ_{conn}, which enforce a valid and meaningful mapping between variables in the candidate repair snippet and those in the buggy context, (2) functionality component ϕ_{func}, which statically captures the behavior of the candidate repair snippet, and (3) the specification of desired behavior, captured in a dynamic profile P (Section 2.3). We now detail the mapping and functionality components, as well as how patches are constructed and validated based on satisfiable semantic search SMT queries.

2.4.1 Mapping Component

Our approach to encoding semantic search queries for program repair takes inspiration from SMT-based input-output-guided component-based program synthesis [35]. The original synthesis goal is to connect a set of components to construct a function f that satisfies a set of input-output pairs $\langle \alpha_i, \beta_i \rangle$ (such that $\forall i, f(\alpha_i) = \beta_i$). This is accomplished by introducing a set of *location variables*, one for each possible component and function input and output variable, that define the order of and connection between components. Programs are synthesized by constructing an SMT query that constrains location variables so that they describe a well-formed program with the desired behavior on the given inputs/outputs. If the query is satisfiable, the satisfying model assigns integers to locations and can be used to construct the desired function. See the prior work by Jha et al. for full details [35].

Mapping Queries for Replacement. We extend the location mechanism to map between the variables used in a candidate repair snippet and those available in the buggy context. We first describe how mapping works for replacement queries, and then the differences required for insertion. We define a set of *locations* as

$$L = \{l_x | x \in \vec{\mathcal{V}}_c \cup \vec{\mathcal{V}}_s\}.$$

The query must constrain locations so that a satisfying assignment tells SOSRepair how to suitably rename variables in $\mathcal{S}$ such that a patch compiles and enforces desired behavior. The variable mapping must be valid: Each variable in $\mathcal{S}$ must uniquely map to some variable in $\mathcal{C}$ (but not vice versa; not all context snippet variables need map to a repair snippet variable). The ψ_{map} constraints therefore define an injective mapping from $\vec{\mathcal{V}}_s$ to $\vec{\mathcal{V}}_c$

$$\psi_{map} := \left(\bigwedge_{x \in \vec{\mathcal{V}}_c \cup \vec{\mathcal{V}}_s} 1 \le l_x \le |\vec{\mathcal{V}}_c| \right) \wedge\ distinct(L, \vec{\mathcal{V}}_c) \wedge\ distinct(L, \vec{\mathcal{V}}_s)$$
$$distinct(L, \vec{V}) := \bigwedge_{x,y \in \vec{V}, x \not\equiv y} l_x \neq l_y.$$

This exposition ignores variable types for simplicity; in practice, we encode them such that matched variables have the same types via constraints on valid locations.

Next, ψ_{conn} establishes the connection between location values and variable values as well as between input and output variables s, $\vec{R}_s$ and their freshly-renamed versions in X_{in} and X_{out} across all covering test executions $t \in T_p \cup T_n$. This is important because although the introduced variables eliminate the problem of trivially unsatisfiable constraints over variables used as both inputs and outputs, naive constraints over the fresh variables — e.g., $(\texttt{len}_{\texttt{in}}{}^1 = 10) \wedge (\texttt{len}_{\texttt{out}}{}^1 = 15)$ — are instead trivially satisfiable. Thus

$$\begin{aligned}
\psi_{conn} &:= \psi_{out} \wedge \psi_{in} \\
\psi_{out} &:= \bigwedge_{x \in X^C_{out}, y \in X^S_{out}} l_x = l_y \Rightarrow \\
&\quad \left(\bigwedge_{t=1}^{|T_p \cup T_n|} x^t_{in} = y^t_{in} \wedge x^t_{out} = y^t_{out} \right) \\
\psi_{in} &:= \bigwedge_{x \in X^C_{in}, y \in X^S_{in}} l_x = l_y \Rightarrow \bigwedge_{t=1}^{|T_p \cup T_n|} x^t_{in} = y^t_{in} \Bigg).
\end{aligned}$$

Where X^C_{in} and X^S_{in} refer to the variables in the context and repair snippet respectively and x_{in} refers to the fresh renamed version of variable x, stored in X_{in} (and similarly for output variables).

Insertion. Instead of drawing $\vec{\mathcal{V}}_c$ from the replacement region (a heuristic design choice to enable scalability), insertion queries define $\vec{\mathcal{V}}_c$ as the set of local variables live after the candidate insertion point. They otherwise are encoded as above.

2.4.2 Functionality Component

ϕ_{func} uses the path constraints describing the candidate repair snippet $\mathcal{S}$ such that the query tests whether $\mathcal{S}$ satisfies the constraints on the desired behavior described by the profile constraints P. The only complexity is that we must copy the symbolic formula to query over multiple simultaneous test executions. Let φ_c be the path constraints from symbolic execution. $\varphi_c(i)$ is a copy of φ_c where all variables $x_{in} \in X^S_{in}$ and $x_{out} \in X^S_{out}$ are syntactically replaced with indexed versions of themselves (e.g., x^i_{in} for x_{in}). Then

$$\phi_{func} := \bigwedge_{i=1}^{|T_p \cup T_n|} \varphi_c(i),$$

ϕ_{func} is the same for replacement and insertion queries.

2.4.3 Patch Construction and Validation

The repair query conjoins the above-described constraints

$$\psi_{map} \wedge \psi_{conn} \wedge \phi_{func} \wedge P.$$

Given $\mathcal{S}$ and $\mathcal{C}$ for which a satisfiable repair query has been constructed, the satisfying model assigns values to locations in L and defines a valid mapping between variables in the *original* snippets S and C (rather than their transformed versions). This mapping is used to rename variables in S and integrate it into the buggy context. For *replacement* edits, the renamed snippet replaces the buggy region wholesale; for insertions, the renamed snippet is inserted immediately before the buggy region. It is possible for the semantic search to return satisfying snippets that do not repair the bug when executed, if either the snippet fails to address the bug correctly, or if the symbolic execution is too imprecise in its description of snippet behavior. Thus, SOSRepair validates patches by running the patched program on the provided test cases, reporting the patch as a fix if all test cases pass.

2.5 Patch Iteration

Traversal. SOSRepair iterates over candidate buggy regions and candidate repair strategies, dynamically testing all snippets whose repair query is satisfiable. SOSRepair is parameterized by a fault localization strategy, which returns a weighted list of candidate buggy lines. Such strategies can be imprecise, especially in the absence of high-coverage test suites [87]. To avoid getting stuck trying many patches in the wrong location, SOSRepair traverses candidate buggy regions using breadth-first search. First, it tries deletion at every region. Deletion is necessary to repair certain defects [115], though it can also lead to low-quality patches [76]. However, simply disallowing deletion does not solve the quality problem: even repair techniques that do not formally support deletion can do so by synthesizing tautological `if` conditions [56], [64]. Similarly, SOSRepair can replace a buggy region with a snippet with no effect. Because patches that effectively delete are likely less maintainable and straightforward than those that simply delete, if a patch deletes functionality, it is better to do so explicitly. Thus, SOSRepair tries deleting the candidate buggy region first by replacing it with an empty candidate snippet whose only constraint is *TRUE*. We envision future improvements to SOSRepair that can create and compare multiple patches per region, preferring those that maintain the most functionality. Next, SOSRepair attempts to replace regions with identified fix code, in order of ranked suspiciousness; finally, SOSRepair tries to repair regions by inserting code immediately before them. We favor replacement over insertion because the queries are more constrained. SOSRepair can be configured with various database traversal strategies, such as trying snippets from the same file as the buggy region first, as well as trying up to N returned matching snippets per edit type per region. SOSRepair then cycles through buggy regions and matched snippets N-wise, before moving to the next edit type.

Profile Refinement. Initially-passing test cases partially specify the expected behavior of a buggy code region, thus constraining which candidate snippets quality to be returned by the search. Initially-failing test cases only specify what the behavior *should not* be (e.g., "given input 2, the output should not be 4"). This is significantly less useful in distinguishing between candidate snippets. Previous work in semantic search-based repair disregarded the negative example behavior in generating dynamic profiles [38]. Such an approach might be suitable for small programs with high-coverage test suites. Unfortunately, in real-world programs, buggy regions may only be executed by failing test cases [87]. We observed this behavior in our evaluation on real-world defects.

To address this problem, other tools, such as Angelix [64], require manual specification of the correct values of variables for negative test cases. By contrast, we address this problem

procedure REFINEPROFILE($program, Tests, X_{out}$)
  $constraints \leftarrow \emptyset$
  for all $t \in Tests$ **do** $\triangleright$ all tests $t \in Tests$ failed
    $c \leftarrow \neg(\bigwedge_{x \in X_{out}} x^t = valOut(t, x, program))$
    $constraints \leftarrow constraints \cup c$
  end for
  return $constraints$
end procedure

Fig. 3. Incremental, counter-example profile refinement. REFINEPROFILE receives a $program$ with the candidate snippet incorporated, a set of $Tests$ that fail on $program$, and the set of output variables X_{out}. It computes new $constraints$ to refine the profile by excluding the observed behavior. $valOut(t, x_i, program)$ returns the output value of variable x_i when test t is executed on $program$.

in SOSRepair via a novel *incremental, counter-example-guided profile* refinement for candidate regions that do not have passing executions. Given an initial profile derived from failing test cases (e.g., "given input 2, the output should not be 4"), SOSRepair tries a single candidate replacement snippet $\mathcal{S}$. If unsuccessful, SOSRepair adds the newly discovered unacceptable behavior to the profile (e.g., "given input 2, the output should not be 6"). Fig. 3 details the algorithm for this refinement process. Whenever SOSRepair tries a snippet and observes that all tests fail, it adds one new negative-behavior constraint to the constraint profile for each failing test. Each constraint is the negation of the observed behavior. For example, if SOSRepair observes that test t fails, it computes its output variable values (e.g., $x_1 = 3$, $x_2 = 4$) and adds the constraint $\neg\big((x_1^t = 3) \wedge (x_2^t = 4)\big)$ to the profile, which specifies that the incorrect observed behavior should not take place. Thus, SOSRepair gradually builds a profile based on negative tests without requiring manual effort. SOSRepair continues on trying replacement snippets with queries that are iteratively improved throughout the repair process. Although this is slower than starting with passing test cases, it allows SOSRepair to patch more defects.

3 THE SOSREPAIR IMPLEMENTATION

We implement SOSRepair using KLEE [12], Z3 [21], and the *clang* [17] infrastructure; the latter provides parsing, name and type resolution, and rewriting facilities, among others. Section 3.1 describes the details of our implementation. Section 3.2 summarizes the steps we took to release our implementation and data, and to make our experiments reproducible.

3.1 SOSRepair Implementation Design Choices

In implementing SOSRepair, we made a series of design decisions, which we now describe.

Snippet Database. SOSRepair uses the symbolic execution engine in KLEE [12] to statically encode snippets. SOSRepair uses KLEE's built-in support for loops, using a two-second timeout; KLEE iterates over the loop as many times as possible in the allocated time. We encode user-defined struct types by treating them as arrays of bytes (as KLEE does). SOSRepair further inherits KLEE's built-in mechanisms for handling internal (GNU C) function calls. As KLEE does not symbolically execute external (non GNU C) function calls, SOSRepair makes no assumptions about such functions' side-effects. SOSRepair instead makes a new symbolic variable for each of the arguments and output, which frees these variables from previously generated constraints. These features substantially expand the expressive power of the considered repair code over previous semantic search-based repair. We do sacrifice soundness in the interest of expressiveness by casting floating point variables to integers (this is acceptable because unsoundness can be caught in testing). This still precludes the encoding of snippets that include floating point constants, but future SOSRepair versions can take advantage of KLEE's recently added floating point support.

Overall, we encode snippets by embedding them in a small function, called from `main`, and defining their input variables as symbolic (using `klee_make_symbolic`). We use KLEE off-the-shelf to generate constraints for the snippet-wrapping function, using KLEE's renaming facilities to transform $\mathcal{F}$ into $\mathcal{F}_t$ for snippet encoding. KLEE generates constraints for nearly all compilable snippets. Exceptions are very rare, e.g., KLEE will not generate constraints for code containing function pointers. However, KLEE will sometimes conservatively summarize snippets with single *TRUE* constraints in cases where it can technically reason about code but is still insufficiently expressive to fully capture its semantics.

Console Output. Real-world programs often print meaningful output. Thus, modeling console output in semantic search increases SOSRepair applicability. We thus define a symbolic character array to represent console output in candidate repair snippets. Because symbolic arrays must be of known size, we only model the first 20 characters of output. We transform calls to `printf` and `fprintf` to call `sprintf` with the same arguments. KLEE handles these standard functions natively. We track console output in the profile by logging the start and end of the buggy candidate region, considering anything printed between the log statements as meaningful.

Profile Construction. For consistency with prior work [38], we use Tarantula [37] to rank suspicious source lines. We leave the exploration of other fault localization mechanisms to future work. To focus our study on SOSRepair efficacy (rather than efficiency, an orthogonal concern), we assume the provision of one buggy method to consider for repair, and then apply SBFL to rank lines in the method. Given such a ranked list, SOSRepair expands the identified lines to surrounding regions of 3–7 lines of code, as in the snippet encoding step. The size of the region is selected by conducting an initial experiment on small programs presented in Section 4.3. SOSRepair attempts to repair each corresponding buggy region in rank order, skipping lines that have been subsumed into previously-identified and attempted buggy regions.

Queries and Iteration. Z3 [21] can natively handle integers, booleans, reals, bit vectors, and several other common data types, such as arrays and pairs. To determine whether a candidate struct type is in scope, we match struct names syntactically. For our experiments, we construct snippet databases from the rest of the program under repair, pre-fix, which supports struct matching. Additionally, programs are locally redundant [96], and developers are more often right than not [22], and thus we hypothesize that a defect may be fixable via code elsewhere in the same program. However, this may be unnecessarily restrictive for more broadly-constructed databases. We leave a more flexible matching of struct types to future work. SOSRepair is configured by default to try repair snippets from the same file as a buggy region first, for all candidate considered regions; then the same module; then the same project.

3.2 Open-Source Release and Reproducibility

To support the reproduction of our results and help researchers build on our work, we publicly release our implementation: `https://github.com/squaresLab/SOSRepair`. We also release a replication package that includes all patches our techniques found on the ManyBugs benchmark and the necessary scripts to rerun the experiment discussed in Section 4.4, and all independently generated tests discussed in Section 4.1.2: `https://github.com/squaresLab/SOSRepair-Replication-Package`.

Our implementation includes Docker containers and scripts for reproducing the evaluation results described in Section 4. The containers and scripts use BugZoo [95], a decentralized platform for reproducing and interacting with software bugs. These scripts both generate snippet databases (which our release excludes due to size) and execute SOSRepair.

SOSRepair uses randomness to make two choices during its execution: the order in which to consider equally suspicious regions returned by SOSRepair's fault localization, and the order in which to consider potential snippets returned by the SMT solver that satisfy all the query constraints. SOSRepair's configuration includes a random seed that controls this randomness, making executions deterministic. However, there remain two sources of nondeterminism that SOSRepair cannot control. First, SOSRepair sets a time limit on KLEE's execution on each code snippet (recall Section 3.1). Due to CPU load and other factors, in each invocation, KLEE may be able to execute the code a different number of times in the time limit, and thus generate different constraints. Second, if a code snippet contains uninitialized variables, those variables' values depend on the memory state. Because memory state may differ between executions, SOSRepair may generate different profiles on different executions. As a result of these two sources of nondeterminism, SOSRepair's results may vary between executions. However, in our experiments, we did not observe this nondeterminism affect SOSRepair's ability to find a patch, only its search space and execution time.

4 Evaluation

This section evaluates SOSRepair, answering several research questions. The nature of each research question informs the appropriate dataset used in its answering, as we describe in the context of our experimental methodology (Section 4.1). We begin by using IntroClass [48], a large dataset of small, well-tested programs, to conduct controlled evaluations of:

- Comparison to prior work: How does SOSRepair perform as compared to SearchRepair [38], the prior semantic-based repair approach (Section 4.2)?
- Tuning: What granularity level is best for the purposes of finding high-quality repairs (Section 4.3)?

Next, in Section 4.4, we address our central experimental concern by evaluating SOSRepair on real-world defects taken from the ManyBugs benchmark [48], addressing:

- Expressiveness: How expressive and applicable is SOSRepair in terms of the number and uniqueness of defects it can repair?
- Quality: What is the quality and effectiveness of patches produced by SOSRepair?
- The role of fault localization: What are the limitations and bottlenecks of SOSRepair's performance?

Section 4.5 discusses informative real-world example patches produced by SOSRepair.

Finally, we isolate and evaluate two key SOSRepair features:

- Performance improvements: How much performance improvements does SOSRepair's novel query encoding approach afford (Section 4.6)?
- Profile refinement: How much is the search space reduced by the negative profile refinement approach (Section 4.7)?

Finally, we discuss threats to the validity of our experiments and SOSRepair's limitations in Section 4.8.

4.1 Methodology

We use two datasets to answer the research questions outlined above. SOSRepair aims to scale semantic search repair to defects in large, real-world programs. However, such programs are not suitable for most controlled large-scaled evaluations, necessary for, e.g., feature tuning. Additionally, real-world programs preclude a comparison to previous work that does not scale to handle them. For such questions, we consider the IntroClass benchmark [48] (Section 4.1.1). However, where possible, and particularly in our core experiments, we evaluate SOSRepair on defects from large, real-world programs taken from the ManyBugs [48] benchmark (Section 4.1.2).

We run all experiments on a server running Ubuntu 16.04 LTS, consisting of 16 Intel(R) Xeon(R) 2.30 GHz CPU E5-2699 v3s processors and 64 GB RAM.

4.1.1 Small, Well-Tested Programs

The IntroClass benchmark [48] consists of 998 small defective C programs (maximum 25 lines of code) with multiple test suites, intended for evaluating automatic program repair tools. Because the programs are small, it is computationally feasible to run SOSRepair on all defects multiple times, for experiments that require several rounds of execution on the whole benchmark. Since our main focus is applicability to real-world defects, we use the IntroClass benchmark for tuning experiments, and to compare with prior work that cannot scale to real-world defects.

Defects. The IntroClass benchmark consists of 998 defects from solutions submitted by undergraduate students to six small C programming assignments in an introductory C programming course. Each problem class (assignment) is associated with two independent test suites: One that is written by the instructor of the course (the black-box test

program	kLOC	tests	defects	patched
`gmp`	145	146	2	0
`gzip`	491	12	4	0
`libtiff`	77	78	9	8
`lighttpd`	62	295	5	1
`php`	1,099	8,471	39	9
`python`	407	355	4	2
`wireshark`	2,814	63	2	2
total	5,095	9,420	65	22

Fig. 4. Subject programs and defects in our study, and the number of each for which SOSRepair generates a patch.

suite), and one that is automatically generated by KLEE [12], a symbolic execution tool that automatically generates tests (the white-box test suite). Fig. 6 shows the number of defects in each program assignment group that fail at least one test case from the *black-box* test suite. The total number of such defects is 778.

Patch Quality. For all repair experiments on IntroClass, we provide the black-box tests to the repair technique to guide the search for a patch. We then use the white-box test suite to measure patch quality, in terms of the percent of held-out tests the patched program passes (higher is better).

4.1.2 Large, Real-World Programs

The ManyBugs [48] benchmark consists of 185 defects taken from nine large, open-source C projects, commonly used to evaluate automatic program repair tools (e.g., [58], [64], [75], [107]).

Defects. The first four columns of Fig. 4 show the project, size of source code, number of developer-written tests, and the number of defective versions of the ManyBugs programs we use to evaluate SOSRepair. Prior work [68] argues for explicitly defining *defect classes*(the types of defects that can be fixed by a given repair method) while evaluating repair tools, to allow for fair comparison of tools on comparable classes. For instance, Angelix [64] cannot fix the defects that require adding a new statement or variable, and therefore all defects that require such modification are excluded from its defect class. For SOSRepair, we define a more general defect class that includes all the defects that can be fixed by editing one or more consecutive lines of code in one location, and are supported by BugZoo (version 2.1.29) [95]. As mentioned in Section 3.2, we use Docker containers managed by BugZoo to run experiments in a reproducible fashion. BugZoo supports ManyBugs scenarios that can be configured on a modern, 64-bit Linux system; we therefore exclude 18 defects from `valgrind` and `fbc`, which require the 32-bit Fedora 13 virtual machine image originally released with ManyBugs. Further, automatically fixing defects that require editing multiple files or multiple locations within a file is beyond SOSRepair's current capabilities. We therefore limit the scope of SOSRepair's applicability only to the defects that require developers to edit one or more consecutive lines of code in a single location. In theory, SOSRepair can be used to find multi-location patches, but considering multiple locations increases the search space and is beyond the scope of this paper.

SOSRepair's defect class includes 65 of the 185 ManyBugs defects. We use method-level fault localization by limiting SOSRepair's fault localization to the method edited by the developer's patch, which is sometimes hundreds of lines long. We construct a single snippet database (recall Section 3) per project from the oldest version of the buggy code among all the considered defects. Therefore, the snippet database contains none of the developer-written patches.

Fig. 5 shows, for each ManyBugs program, the mean and median snippet size, the number of variables in code snippets, the number of functions called within the snippets, the number of constraints for the code snippets stored in the database, and the time spent on building the database. For each program, SOSRepair generates thousands of snippets, and for each snippet, on average, KLEE generates tens of SMT constraints. SOSRepair generated a total of 145,639 snippets, with means of 140 characters, 4 variables, 1 function call, and 13 SMT constraints. The database generation is SOSRepair's most time-consuming step, which only needs to happen once per project. The actual time to generate the database varies based on the size of the project. It takes from 2.3 hours for `gzip` up to 115 hours for `wireshark`, which is the largest program in the ManyBugs benchmark. On average, it takes 8.2 seconds to generate each snippet. However, we collected these numbers using a single thread. This step is easily parallelizable, representing a significant performance opportunity in generating the database. We set the snippet granularity to 3–7 lines of code, following the results of our granularity experiments (Section 4.3) and previous work on code redundancy [25].

program	snippets	snippet size (# characters)		variables		# of functions called in the snippet		constraints per snippet		time to build the DB (h)
		mean	median	mean	median	mean	median	mean	median	
`gmp`	6,003	95.4	88	4.0	4	0.9	0	32.7	3	26.3
`gzip`	2,028	103.2	93	2.6	2	1.1	1	25.4	2	2.3
`libtiff`	3,010	114.8	108	3.0	3	1.2	1	29.9	2	5.8
`lighttpd`	797	90.6	82	2.0	2	1.4	1	24.8	2	2.3
`php`	22,423	113.5	100	2.7	2	1.4	1	19.8	2	51.6
`python`	20,960	116.1	108	2.4	2	1.0	1	26.9	1	41.9
`wireshark`	90,418	157.7	145	4.3	4	1.6	1	6.4	2	115.1

Fig. 5. The code snippet database SOSRepair generates for each of the ManyBugs programs. SOSRepair generated a total of 145,639 snippets, with means of 140 characters, 4 variables, 1 function call, and 13 SMT constraints. On average, SOSRepair builds the database in 35 hours, using a single thread.

problem class	defects	SearchRepair	SOSRepair
checksum	29	0	3
digits	91	0	24
grade	226	2	37
median	168	68	87
smallest	155	73	120
syllables	109	4	75
total	778	150	346
mean quality		97.3%	91.5%

Fig. 6. Number of defects repaired by SearchRepair and SOSRepair on IntroClass dataset. “Mean quality” denotes the mean percent of the associated held-out test suite passed by each patched programs.

Patch Quality. A key concern in automated program repair research is the *quality* of the produced repairs [76], [85]. One mechanism for objectively evaluating patch quality is via independent test suites, held out from patch generation. The defects in ManyBugs are released with developer-produced test suites of varying quality, often with low coverage of modified methods. Therefore, we construct additional held-out test suites to evaluate the quality of generated patches. For a given defect, we automatically generate unit tests for all methods modified by either the project’s developer or by at least one of the automated repair techniques in our evaluation. We do this by constructing small driver programs that invoke the modified methods:

- Methods implemented as part of an *extension* or *module* can be directly invoked from a driver’s `main` function (e.g., the `substr_compare`method of `php string` module.)
- Methods implemented within *internal libraries* are invoked indirectly by using other functionality. For example, the method `do_inheritance_check_on_method`of `zend_compile` library in `php` is invoked by creating and executing `php`programs that implement inheritance. For such methods, the driver’s `main`function sets the values of requisite global variables and then calls the functionality that invokes the desired method.

We automatically generate random test inputs for the driver programs that then invoke modified methods. We generate inputs until either the tests fully cover the target method or until adding new test inputs no longer significantly increases statement coverage. For four `php` and two `lighttpd` scenarios for which randomly generated test inputs were unable to achieve high coverage, we manually added new tests to that effect. For `libtiff` methods requiring tiff images as input, we use 7,214 tiff images randomly generated and released by the AFL fuzz tester [2]. We use the developer-patched behavior to construct test oracles, recording logged, printed, and returned values and exit codes as ground truth behavior. If the developer-patched program crashes on an input, we treat the crash as the expected behavior.

We release these generated test suites (along with all source code, data, and experimental results) to support future evaluations of automated repair quality on ManyBugs. All materials may be downloaded from `https://github.com/squaresLab/SOSRepair-Replication-Package`. This release is the first set of independently-generated quality-evaluation test suites for ManyBugs.

Baseline Approaches. We compare to three previous repair techniques that have been evaluated on (subsets) of ManyBugs, relying on their public data releases. Angelix [64] is a state-of-the-art semantic program repair approach; Prophet [58] is a more recent heuristic technique that instantiates templated repairs [56], informed by machine learning; and GenProg [49] uses genetic programming to combine statement-level program changes in a repair search. GenProg has been evaluated on all 185 ManyBugs defects; Angelix, on 82 of the 185 defects; Prophet, on 105 of 185. Of the 65 defects that satisfy SOSRepair’s defect class, GenProg is evaluated on all 65 defects, Angelix on 30 defects, and Prophet on 39 defects.

4.2 Comparison to SearchRepair

First, to substantiate SOSRepair’s improvement over previous work in semantic search-based repair, we empirically compare SOSRepair’s performance to SearchRepair [38]. Because SearchRepair does not scale to the ManyBugs programs, we conduct this experiment on the IntroClass dataset (Section 4.1.1). We use the black-box tests to guide the search for repair, and the white-box tests to evaluate the quality of the produced repair.

Fig. 6 shows the number of defects patched by each technique. SOSRepair patches more than twice as many defects as SearchRepair (346 versus 150, out of the 778 total repairs attempted). This difference is statistically significant based on Fisher’s exact test ($p < 10^{-15}$). The bottom row shows the mean percent of the associated held-out test suite passed by each patched program. Note that SOSRepair’s average patch quality is slightly lower than SearchRepair’s (91.5 versus 97.3%). However, 239 of the 346 total SOSRepair patches pass 100% of the held-out tests, constituting substantially more very high-quality patches than SearchRepair finds total (150). Overall, however, semantic search-based patch quality is quite high, especially as compared to patches produced by prior techniques as evaluated in the prior work: AE [107] finds patches for 159 defects with average quality of 64.2%, TrpAutoRepair [75] finds 247 patches with 72.1% quality, and GenProg [108] finds 287 patches with average quality of 68.7% [38]. Overall, SOSRepair outperforms these prior techniques in expressive power (number of defects repaired, at 346 of 778), and those patches are of measurably higher quality.

4.3 Snippet Granularity

Snippet granularity informs the size and preparation of the candidate snippet database, as well as SOSRepair’s expressiveness. Low granularity snippets may produce prohibitively large databases and influence patch quality. High granularity (i.e., larger) snippets lower the available redundancy (previous work suggests that the highest code redundancy is found in snippets of 1–7 lines of code [25]) and may reduce the probability of finding fixes. Both for tuning purposes and to assess one of our underlying hypotheses, we evaluate the effect of granularity on repair success and

	patches			patches passing all held-out tests			mean % of held-out tests passing		
program	1–3	3–7	6–9	1–3	3–7	6–9	1–3	3–7	6–9
checksum	0	3	8	0	3	8	—	100.0%	100.0%
digits	26	24	17	14	9	5	91.5%	89.5%	92.9%
grade	1	37	2	1	37	2	100.0%	100.0%	100.0%
median	14	87	52	1	63	44	84.5%	95.0%	95.5%
smallest	60	120	132	27	57	54	80.4%	82.2%	78.5%
syllables	87	75	17	77	70	12	97.0%	98.6%	97.0%
Total	188	346	211	120	239	125			

Fig. 7. A comparison of applying SOSRepair to IntroClass defects with three different levels of granularity: 1–3, 3–7, and 6–9 lines of code.

patch quality by systematically altering the granularity level of both the code snippets in the SOSRepair database and the buggy snippet to be repaired. Because this requires a large number of runs on many defects to support statistically significant results, and to reduce the confounds introduced by real-world programs, we conduct this experiment on the IntroClass dataset, and use SOSRepair to try to repair all defects in the dataset using granularity level configuration of 1–3 lines, 3–7 lines, and 6–9 lines of code.

Fig. 7 shows the number of produced patches, the number of those patches that pass *all* the held-out tests, and the mean percent of held-out test cases that the patches pass, by granularity of the snippets in the SOSRepair database. The granularity of 3–7 lines of code produces the most patches (346 versus 188 and 211 with other granularities), and the most patches that pass all the held-out tests (239 versus 120 and 125 with other granularities). Fisher's exact test confirms that these differences are statistically significant (all $p < 10^{-70}$).

While the number of patches that pass all defects is significantly higher for the 3–7 granularity, and the fraction of patches that pass all held-out tests is higher for that granularity (69.1% for 3–7, 63.8% for 1–3, and 59.2% for 6–9), the mean patch quality is similar for all the three levels of granularity. We hypothesize that this observation may be a side-effect of the small size of the programs in the IntroClass benchmark and the high redundancy induced by many defective programs in that benchmark attempting to satisfy the same specification. We suspect this observation will not extend to benchmarks with more diversity and program complexity, and thus make no claims about the effect of granularity on average quality.

We configure our database in subsequent experiments to use snippets of 3–7 lines, as these results suggest that doing so may provide a benefit in terms of expressive power. The results of this study may not immediately extend to large, real-world programs; we leave further studies exploring repair granularity for large programs to future work.

4.4 Repair of Large, Real-World Programs

A key contribution of our work is a technique for semantic search-based repair that scales to real-world programs; we therefore evaluate SOSRepair on defects from ManyBugs that fall into its defect class (as described in Section 4.1.2). The "patched" column in Fig. 4 summarizes SOSRepair's ability to generate patches. Fig. 8 presents repair effectiveness and quality for all considered defects in the class, comparing them with patches produced by previous evaluations of Angelix, Prophet, and GenProg. Fig. 8 enumerates defects for readability and maps each "program ID" to a revision pair of the defect and developer-written repair.

4.4.1 Repair Expressiveness and Applicability

SOSRepair patches 22 of the 65 defects that involved modifying consecutive lines by the developer to fix those defects. The Angelix, Prophet, and GenProg columns in Fig. 8 indicate which approaches succeed on patching those defects (✗ for not patched, and NA for not attempted, corresponding to defects outside the defined defect class for a technique). There are 5 defects that all four techniques patch. SOSRepair is the only technique that repaired `libtiff-4`. SOSRepair produces patches for 3 defects that Angelix cannot patch, 5 defects that Prophet cannot patch, and 6 defects that GenProg cannot patch. These observations corroborate results from prior work on small programs, which showed that semantic search-based repair could target and repair defects that other techniques cannot [38].

Even though efficiency is not a focus of SOSRepair's design, we measured the amount of time required to generate a patch with SOSRepair. On average, it took SOSRepair 5.25 hours to generate patches reported in Fig. 8. Efficiency is separate from, and secondary to the ability to produce patches and can be improved by taking advantage of parallelism and multithreading in SOSRepair's implementation. On average, 57.6% of the snippets in the database (satisfying type constraints) matched the SMT query described in Section 2.4. Of the repaired defects, seven involve insertion, seven involve replacement, and eight involve deletion.

4.4.2 Repair Effectiveness and Quality

Fig. 8 shows the percent of evaluation tests passed by the SOSRepair, Angelix, Prophet, and GenProg patches. "Coverage" is the average statement-level coverage of the generated tests on the methods modified by either the developer or by at least one automated repair technique in our evaluation. SOSRepair produces more patches (9, 41%) that pass all independent tests than Angelix (4), Prophet (5) and, GenProg (4). For the defects patched in-common by SOSRepair and other techniques, Angelix and SOSRepair patch 9 of the same defects; both SOSRepair and Angelix produce 4 patches that pass all evaluation tests on this set. Prophet and

program ID:	revision pair	coverage	Angelix	Prophet	GenProg	SOSRepair	SOSRepair$^{\oplus}$
gmp-2:	14166-14167	—	✔	✘	✔	✘	✘
gzip-2:	3fe0caeada-39a362ae9d	—	✘	✔	✘	✘	✘
gzip-3:	1a085b1446-118a107f2d	—	✘	✘	✔	✘	✘
gzip-4:	3eb6091d69-884ef6d16c	—	✔	✘	✘	✘	✘
libtiff-1:	3b848a7-3edb9cd	90%	✔ 99%	✔ 99%	✔ 97%	✔ 97% ✂	✘
libtiff-2:	a72cf60-0a36d7f	76%	✔ 100%	✘	✔ 100%	✔ 100% ✂	✔ 100% ✂
libtiff-3:	eec4c06-ee65c74	73%	✔ 96%	✔ 96%	✔ 96%	✔ 96% ✂	✔ 96% ✂
libtiff-4:	09e8220-f2d989d	96%	✘	✘	✘	✔ 59% ⇝	✔ 100% ✍
libtiff-5:	371336d-865f7b2	50%	✔ 100%	✘	✔ 98%	✔ 99% ⇝	✘
libtiff-6:	764dbba-2e42d63	73%	✔ 92%	✔ 92%	✔ 28%	✔ 99% ✂	✔ 99% ⇝
libtiff-7:	e8a47d4-023b6df	82%	✔ 100%	✘	✔ 0%	✔ 100% ✂	✔ 100% ⇝
libtiff-8:	eb326f9-eec7ec0	90%	✘	✔ 100%	✔ 100%	✔ 60% ✍	✔ 100% ✍
libtiff-9:	b2ce5d8-207c78a	—	✘	✘	✘	✘	✘
lighttpd-1:	2661-2662	50%	NA	✔ 100%	✔ 100%	✔ 100% ✍	✔ 100% ✍
lighttpd-3:	2254-2259	—	NA	✘	✘	✘	✘
lighttpd-4:	2785-2786	—	NA	✘	✘	✘	✘
lighttpd-5:	1948-1949	—	NA	✔	✘	✘	✘
php-1:	74343ca506-52c36e60c4	89%	NA	NA	✔ 17%	✔ 100% ⇝	✔ 100% ⇝
php-2:	70075bc84c-5a8c917c37	86%	✔ 100%	✔ 100%	✔ 0%	✔ 100% ✂	✔ 100% ✂
php-3:	8138f7de40-3acdca4703	63%	‡	✔ 100%	✘	✔ 100% ✍	✘
php-4:	1e6a82a1cf-dfa08dc325	100%	NA	NA	✘	✔ 53% ✍	✔ 53% ✍
php-5:	ff63c09e6f-6672171672	79%	NA	NA	✔ 80%	✔ 90% ⇝	✔ 50% ✍
php-6:	eeba0b5681-d3b20b4058	40%	NA	NA	✔ 0%	✔ 0% ✍	✔ 100% ⇝
php-7:	77ed819430-efcb9a71cd	70%	✘	✔ 100%	✔ 50%	✔ 100% ✍	✘
php-8:	01745fa657-1f49902999	100%	✔ 67%	✘	✔ 100%	✔ 67% ⇝	✘
php-9:	7aefbf70a8-efc94f3115	79%	NA	NA	✘	✔ 91% ⇝	✘
php-14:	0a1cc5f01c-05c5c8958e	—	NA	NA	✔	✘	✘
php-15:	5bb0a44e06-1e91069eb4	—	✘	✔	✘	✘	✘
php-16:	fefe9fc5c7-0927309852	—	✘	✔	✔	✘	✘
php-17:	e65d361fde-1d984a7ffd	—	✔	✔	✘	✘	✘
php-18:	5d0c948296-8deb11c0c3	—	✘	✘	✘	✘	✘
php-19:	63673a533f-2adf58cfcf	—	✔	✔	✘	✘	✘
php-20:	187eb235fe-2e25ec9eb7	—	✔	✔	✔	✘	✘
php-21:	db01e840c2-09b990f499	—	✘	✘	✘	✘	✘
php-22:	453c954f8a-daecb2c0f4	—	✘	✘	✔	✘	✘
php-23:	b60f6774dc-1056c57fa9	—	✔	✔	✘	✘	✘
php-24:	1f78177e2b-d4ae4e79db	—	NA	NA	✘	✘	✘
php-25:	2e5d5e5ac6-b5f15ef561	—	NA	NA	✔	✘	✘
php-26:	c4eb5f2387-2e5d5e5ac6	—	NA	NA	✔	✘	✘
php-27:	ceac9dc490-9b0d73af1d	—	NA	NA	✔	✘	✘
php-28:	fcbfbea8d2-c1e510aea8	—	NA	NA	✘	✘	✘
php-29:	236120d80e-fb37f3b20d	—	NA	NA	✔	✘	✘
php-30:	55acfdf7bd-3c7a573a2c	—	NA	NA	✔	✘	✘
php-31:	ecc6c335c5-b548293b99	—	NA	NA	✔	✘	✘
php-32:	eca88d3064-db0888dfc1	—	NA	NA	✔	✘	✘
php-33:	544e36dfff-acaf9c5227	—	NA	NA	✘	✘	✘
php-34:	9de5b6dc7c-4dc8b1ad11	—	NA	NA	✔	✘	✘
php-35:	c1322d2505-cfa9c90b20	—	NA	NA	✔	✘	✘
php-36:	60dfd64bf2-34fe62619d	—	NA	NA	✔	✘	✘
php-37:	0169020e49-cdc512afb3	—	NA	NA	✘	✘	✘
php-38:	3954743813-d4f05fbffc	—	NA	NA	✘	✘	✘
php-39:	438a30f1e7-7337a901b7	—	NA	NA	✘	✘	✘
python-1:	69223-69224	100%	NA	✔ 33%	✘	✔ 76% ✍	✔ 76% ✍
python-2:	69368-69372	72%	NA	✔ 54%	✘	✔ 50% ⇝	✔ 50% ✍
python-3:	70098-70101	—	NA	✔	✘	✘	✘
python-4:	70056-70059	—	NA	✘	✔	✘	✘
wireshark-1:	37112-37111	100%	✔ 87%	✔ 87%	✔ 87%	✔ 100% ✂	✔ 100% ✂
wireshark-2:	37122-37123	100%	NA	NA	✔ 87%	✔ 100% ✂	✔ 100% ✂
↓ additional defects patched by SOS$^{\oplus}$ ↓							
gmp-1:	13420-13421	97%	✔ 99%	✔ 99%	✘	✘	✔ 100% ✍
gzip-1:	a1d3d4019d-f17cbd13a1	79%	‡	✔ 100%	✘	✘	✔ 100% ✍
lighttpd-2:	1913-1914	56%	NA	✔ 100%	✘	✘	✔ 100% ⇝
php-10:	51a4ae6576-bc810a443d	90%	NA	NA	✔ 92%	✘	✔ 92% ✍
php-11:	d890ece3fc-6e74d95f34	72%	‡	✔ 100%	✘	✘	✔ 100% ✍
php-12:	eeba0b5681-f330c8ab4e	42%	NA	NA	✔ 0%	✘	✔ 100% ✍
php-13:	80dd931d40-7c3177e5ab	71%	NA	NA	✘	✘	✔ 100% ✍

Fig. 8. SOSRepair patches 22 of the 65 considered defects, 9 (41%) of which pass all of the independent tests. When SOSRepair is manually provided a fault location (SOSRepair$^{\oplus}$), it patches 23 defects, 16 (70%) of which pass all of the independent tests. All defects repaired by either SOSRepair or SOSRepair$^{\oplus}$ (shaded in gray) have a generated test suite for patch quality assessment. **Coverage** is the mean statement-level coverage of that test suite on the patch-modified methods. ✔ indicates that a technique produced a patch, ✘ indicates that a technique did not produce a patch, and NA indicates that the defect was not attempted by a technique (for Angelix, this defect was outside its defect class; for Prophet this defect was not available because Prophet was evaluated on an older version of ManyBugs). Three of the released Angelix patches [64] (denoted ‡) do not automatically apply to the buggy code. Each SOSRepair and SOSRepair$^{\oplus}$ patch is either a replacement (⇝), an insertion (✍), or a deletion (✂).

SOSRepair patch 11 of the same defects; both SOSRepair and Prophet produce 5 patches that pass all evaluation tests on this set. GenProg and SOSRepair patch 16 of the same defects; 4 out of these 16 GenProg patches and 8 SOSRepair patches pass all evaluation tests. Thus, SOSRepair produces more patches that pass all independent tests than GenProg, and as many such patches as Angelix and Prophet. This suggests that semantic code search is a promising approach to generate high-quality repairs for real defects, and that it has potential to repair defects that are outside the scope of other, complementary repair techniques.

4.4.3 Improving Patch Quality through Fault Localization

Although these baseline results are promising, most of the patches previous semantic search-based repair produced on small program defects passed all held-out tests [38]. We investigated why SOSRepair patch quality is lower than this high bar. We hypothesized that two possible reasons are that real-world buggy programs do not contain code that can express the needed patch, or that fault localization imprecision hampers SOSRepair success. Encouragingly, anecdotally, we found that many buggy programs do contain code that can express the developer patch. However, fault localization is the more likely culprit. For example, for `gmp-1`, fault localization reports 59 lines as equally-highly suspicious, including the line modified by the developer, but as part of its breadth-first strategy, SOSRepair only tries 10 of these 59.

We further observed that in some cases, more than one mapping between variables satisfies the query, but only one results in a successful patch. Since trying all possible mappings is not scalable, SOSRepair only tries the first mapping selected by the solver. Including more variables in the mapping query increases the number of patch possibilities, but also the complexity of the query.

We created SOSRepair$^{\oplus}$, a semi-automated version of SOSRepair that can take hints from the developer regarding fault location and variables of interest. SOSRepair$^{\oplus}$ differs from SOSRepair in the following two ways:

1) SOSRepair uses spectrum-based fault localization [37] to identify candidate buggy code regions. SOSRepair$^{\oplus}$ uses a manually-specified candidate buggy code region. In our experiments, SOSRepair$^{\oplus}$ uses the location of the code the developer modified to patch the defect as its candidate buggy code region, simulating the developer suggesting where the repair technique should try to repair a defect.
2) SOSRepair considers all live variables after the insertion line in its query. While multiple mappings may exist that satisfy the constraints, not all such mappings may pass all the tests. SOSRepair uses the one mapping the SMT solver returns. SOSRepair$^{\oplus}$ can be told which variables not to consider, simulating the developer suggesting to the repair technique which variables likely matter for a particular defect. A smaller set of variables of interest increases the chance that the mapping the SMT solver returns and SOSRepair$^{\oplus}$ tries is a correct one. We found that for 6 defects (`gzip-1`, `libtiff-4`, `libtiff-8`, `php-10`, `php-12`, and `gmp-1`), SOSRepair failed to produce a patch because it attempted an incorrect mapping. For these 6 defects, we instructed SOSRepair$^{\oplus}$ to reduce the variables of interest to just those variables used in the developer-written patch.

On our benchmark, SOSRepair$^{\oplus}$ patches 23 defects and 16 (70%) of them pass all independent tests. While it is unsound to compare SOSRepair$^{\oplus}$ to prior, fully-automated techniques, our conclusions are drawn only from the comparison to SOSRepair; the quality results for the SOSRepair$^{\oplus}$-patched defects for the prior tools in Fig. 8 are only for reference.

Our experiments show that precise fault localization allows SOSRepair$^{\oplus}$ to patch 7 additional defects SOSRepair could not (bottom of Fig. 8), and to improve the quality of 3 of SOSRepair's patches. Overall, 9 new patches pass 100% of the independent tests.

SOSRepair and SOSRepair$^{\oplus}$ sometimes attempt to patch defects at different locations: SOSRepair using spectrum-based fault localization and SOSRepair$^{\oplus}$ at the location where the developer patched the defect. For 6 defects, SOSRepair finds a patch, but SOSRepair$^{\oplus}$ does not. Note that defects can often be patched at multiple locations, and developers do not always agree on a single location to patch a particular defect [10]. Thus, the localization hint SOSRepair$^{\oplus}$ receives is a heuristic, and may be neither unique nor optimal. In each of these 6 cases, the patch SOSRepair finds it at an alternate location than where the developer patched the defect.

Because SOSRepair and SOSRepair$^{\oplus}$ sometimes patch at different locations, the patches they produce sometimes differ, and accordingly, so does the quality of those patches. In our experiments, in all but one case (`php-5`) SOSRepair$^{\oplus}$ patches were at least as high, or higher quality than SOSRepair patches for the same defect.

We conclude that research advancements that produce more accurate fault localization or elicit guidance from developers in a lightweight manner are likely to dramatically improve SOSRepair performance. Additionally, input (or heuristics) on which variables are likely related to the buggy functionality (and are thus appropriate to consider) could limit the search to a smaller but more expressive domain, further improving SOSRepair.

4.5 Example Patches

In this section, we present several SOSRepair patches produced on the ManyBugs defects (Section 4.4), comparing them to developer patches and those produced by other tools. Our goal is not to be comprehensive, but rather to present patches highlighting various design decisions.

Example 1 python-1. The `python` interpreter at revision #69223 fails a test case concerning a variable that should never be negative. The developer patch is as follows:

```
 }
+if (timeout < 0) {
+  PyErr_SetString(PyExc_ValueError,
+    "timeout must be non-negative");
+  return NULL;
+}
 seconds = (long)timeout;
```

Fault localization correctly identifies the developer's insertion point for repair. Several snippets in the python project perform similar functionality to the fix, including the following, from the IO module:

```
if (n < 0) {
 PyErr_SetString(PyExc_ValueError,
      "invalid key number");
 return NULL;
}
```

SOSRepair correctly maps variable n to timeout and inserts the code to repair the defect. Although the error message is not identical, the functionality is, and suitable to satisfy the developer tests. However, unlike the developer tests, the generated tests do consider the error message, explaining the patch's relatively low success on the held-out tests. Synthesizing good error messages is an open problem; such a semantically meaningful patch could still assist developers in more quickly addressing the underlying defect [106].

GenProg did not patch this defect; Angelix was not attempted on it, as the defect is outside its defect class. The Prophet patch modifies an if-check elsewhere in the code to include a tautological condition:

```
- if ((!rv)) {
+ if((!rv) && !(1)) {
  if (set_add_entry((PySetObject *)...
```

This demonstrates how techniques that do not delete directly can still do so, motivating our explicit inclusion of deletion.

Example 2 php-2. We demonstrate the utility of explicit deletion with our second example, from php-2 (recall Fig. 8). At the buggy revision, php fails two test cases because of an incorrect value modification in its string module. Both the developer and SOSRepair delete the undesired functionality:

```
- if (len > s1_len - offset) {
-  len = s1_len - offset;
- }
```

Angelix and Prophet correctly eliminate the same functionality by modifying the if condition such that it always evaluates to false. GenProg inserts a return; statement in a different method.

Example 3 php-1. Finally, we show a SOSRepair patch that captures a desired semantic effect while syntactically different from the human repair. Revision 74343ca506 of php-1 (recall Fig. 8) fails 3 test cases due to an incorrect condition around a loop break, which the developer modifies:

```
- if (just_read < toread) {
+ if (just_read == 0) {
   break;
  }
```

This defect inspired our illustrative example (Section 2.1). Using default settings, SOSRepair first finds a patch identical to the developer fix. To illustrate, we present a different but similar fix that SOSRepair finds if run beyond the first repair:

```
if ((int)box_length <= 0) {
   break;
}
```

SOSRepair maps box_length to just_read, and replaces the buggy code. In this code, just_read is only ever greater than or equal to zero, such that this patch is acceptable. Angelix and Prophet were not attempted on this defect; GenProg deletes other functionality.

4.6 Query Encoding Performance

To answer our final two research questions, we isolate and evaluate two key novel features of SOSRepair. First, this section evaluates the performance improvements gained by SOSRepair's novel query encoding approach. Second, Section 4.7 evaluates the effects of SOSRepair's negative profile refinement approach on reducing the search space.

In the repair search problem, query complexity is a function of the number of test inputs through a region and the number of possible mappings between a buggy region and the repair context. To understand the differences between SOSRepair's and the old approach's encodings, consider a buggy snippet $\mathcal{C}$ with two input variables a and b and a single output variable c. Suppose $\mathcal{C}$ is executed by two tests, t_1 and t_2. And suppose $\mathcal{S}$ is a candidate repair snippet with two input variables x and y, a single output variable z, and path constraints φ_c generated by the symbolic execution engine. SOSRepair's encoding uses *location variables* to discover a valid mapping between variables a, b and x, y that satisfy φ_c constraints for both test cases t_1 and t_2, with a single query (recall Section 2.4.1). Meanwhile, the prior approach [38] traverses all possible mappings between variables $(m_1 : (a = x) \wedge (b = y) \wedge (c = z)$ and $m_2 : (a = y) \wedge (b = x) \wedge (c = z))$, and creates a query for every test case, for every possible variable mapping. A satisfiable query implies its mapping is valid for that particular test. For example, to show that mapping m_1 is a valid mapping, two queries are required (one for t_1 and one for t_2), and only if both are satisfiable is m_1 considered valid. The number of queries required for this approach grows exponentially in the number of variables, as there is an exponential number of mappings (permutation) of the variables. In our example, there are two possible mappings and two tests, so four queries are required, unlike SOSRepair's one.

To evaluate the performance impact of SOSRepair's new encoding, we reimplement the previous encoding approach [38]. We then compare SMT solver speed on the same repair questions using each encoding. Running on two randomly-selected ManyBugs defects, we measured the response time of the solver on more than 10,000 queries for both versions of encoding techniques. Fig. 9 shows the speed up using the new encoding as compared to the old encoding, as a function of query complexity (number of tests times the number of variable permutations). The new encoding approach delivers a significant speed up over the previous approach, and the speed up increases linearly with query complexity ($R^2 = 0.982$).

Looking at the two approaches individually, query time increases linearly with query complexity (growing slowly

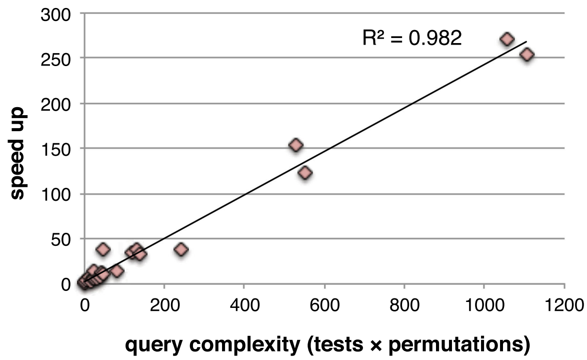

Fig. 9. The speedup of the new encoding approach over the previous approach grows with query complexity.

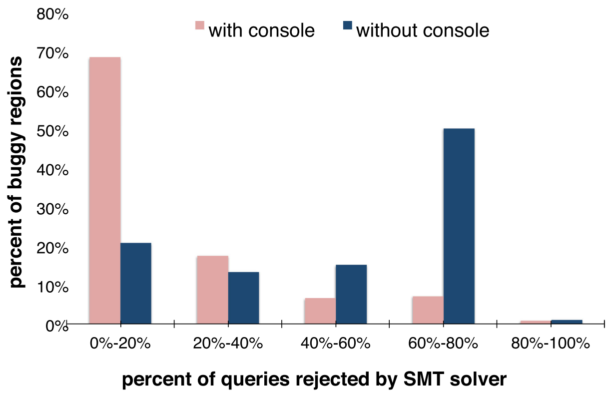

Fig. 10. Fraction of defects that can reject fractions of the search space (measured via SMT queries) using only iteratively-constructed negative examples. Profile refinement improves scalability by reducing the number of candidate snippets to consider. Console output that relies on symbolic values affects this performance.

slope-wise, but with a very high $R^2 = 0.993$) with the previous encoding, and is significantly more variable with the new encoding and does not appear linearly related to query complexity ($R^2 = 0.008$). Overall, Fig. 9 shows the speed up achieved with the new encoding, and its linear increase as query complexity grows.

4.7 Profile Refinement Performance

The profile refinement approach (recall Section 2.5) uses negative tests to iteratively improve a query, reduce the number of attempted candidate snippets, and repair defects without covering passing test cases. By default, SOSRepair uses the automated, iterative query refinement on all defects whenever at least one faulty region under consideration is covered only by negative test cases. In our experiments, for 2 ManyBugs defects (`libtiff-8` and `lighttpd-2`), the patches SOSRepair and SOSRepair$^{\oplus}$ produce cover a region only covered by negative test cases, though SOSRepair and SOSRepair$^{\oplus}$ use the refinement process while attempting to patch other defects as well.

In this experiment, we evaluate the effect of iterative profile refinement using negative examples on the size of the considered SMT search space. We conduct this experiment on a subset of the IntroClass dataset to control for the effect of symbolic execution performance (which is highly variable on the real-world programs in ManyBugs). We ran SOSRepair on all the defects in the `median`, `smallest`, and `grade` programs, only using the initially failing test cases, with profile refinement, for repair. For every buggy line selected by the fault localization and expanded into a region with granularity of 3–7 lines of code, we measured the number of candidate snippets in the database that can be rejected by the SMT-solver (meaning the patch need not be dynamically tested to be rejected, saving time) using only negative queries.

Fig. 10 shows the percent of the search space excluded after multiple iterations for all buggy regions. For example, the first bar shows that on 68% of buggy regions tried, fewer than 20% of candidate snippets were eliminated by the solver when only negative tests are available, leaving more than 80% of possible candidates for dynamic evaluation. We find that approach effectiveness depends on the nature of the defect and snippets. In particular, the approach performs poorly when desired snippet behavior involves console output that depends on a symbolic variable. This makes sense: KLEE produces random output in the face of symbolic console output, and such output is uninformative in specifying undesired behavior. Our results show that on 14% of the defects (that are dependent on console output), more than 40% of database snippets can be rejected using only the test cases that the program initially failed. We also transformed the defects in the dataset to capture console output by variable assignments, treating those variables as the output (rather than the console printout); Fig. 10 also shows the results of running the same study on the modified programs. More than 40% of the possible snippets can be eliminated for 66% of the preprocessed programs. Overall, profile refinement can importantly eliminate large amounts of the search space, but its success depends on the characteristics of the code under repair.

4.8 Threats and Limitations

Even though SOSRepair works on defects that require developers to modify a single (potentially multi-line) location in the source code, we ensure that it generalizes to all kinds of defects belonging to large unrelated projects by evaluating SOSRepair on a subset of the ManyBugs benchmark [48], which consists of real-world, real-developer defects, and is used extensively by prior program repair evaluations [48], [58], [64], [70], [75], [107]. The defects in our evaluation also cover the novel aspects of our approach, e.g., defects with only negative profiles, console output, and various edit types.

Our work inherits KLEE's limitations: SOSRepair cannot identify snippets that KLEE cannot symbolically execute, impacting patch expressiveness nevertheless, the modified buggy code can include KLEE-unsupported constructs, such as function pointers. Note that this limitation of KLEE is orthogonal to our repair approach. As KLEE improves in its handling of more complex code, so will SOSRepair. Our discussion of other factors influencing SOSRepair success (recall Section 4.4) suggests directions for improving applicability and quality.

Our experiments limit the database of code snippets to those found in the same project, based on observations of high within-project redundancy [4]. Anecdotally, we have observed SOSRepair failing to produce a patch when using snippets only from the same project, but succeeding with a correct patch when using snippets from other projects. For example, for `gzip-1` defect, the code in `gzip` lacks the necessary snippet to produce a patch, but that snippet appears in the python code. Extending SOSRepair to use snippets from other projects could potentially improve SOSRepair's effectiveness, but also creates new scalability challenges, including handling code snippets that include custom-defined, project-specific types and structures.

Precisely assessing patch quality is an unsolved problem. As with other repair techniques guided by tests, we use tests, a partial specification, to evaluate the quality of SOSRepair's patches. Held-out, independently generated or written test suites represent the state-of-the-art of patch quality evaluation [85], along with manual inspection [58], [76]. Although developer patches (which we use as a functional oracle) may contain bugs, in the absence of a better specification, evaluations such as ours must rely on the developers.

We conduct several experiments (e.g., Sections 4.3 and 4.7) on small programs from the IntroClass benchmark [48], since these experiments require controlled, large-scale executions of SOSRepair. Even though these experiments provide valuable insights, their results may not immediately extend to large, real-world programs.

We publicly release our code, results, and new test suites to support future evaluation, reproduction, and extension, mitigating the risk of errors in our implementation or setup. All materials may be downloaded from `https://github.com/squaresLab/SOSRepair` (SOSRepair's implementation), and `https://github.com/squaresLab/SOSRepair-Replication-Package` (SOSRepair's replication package).

5 Related Work

We place our work in the context of related research in two areas, code search and automated program repair.

5.1 Code Search

Execution-based semantic code search [77] executes code to find matches with queries as test cases, signature, and keywords [77]. Meanwhile constraint-satisfaction-based search [88], [89], [90] matches input-output examples to code fragments via symbolic execution. SOSRepair builds on this prior work. Synthesis can adapt code-search results to a desired context [93], [105]. The prior approaches had humans directly or indirectly write queries. By contrast, SOSRepair automatically extracts search queries from program state and execution, and uses the query results to map snippets to a new context. Other code search work synthesizes Java directly from free-form queries [32], [86] or based on crash reports [27]. While effective at repairing Java expressions that use wrong syntax or are missing arguments [32], this type of repair does not target semantic errors and requires an approximate Java-like expression as part of the query (and is thus similar to synthesis by sketching [86]).

5.2 Program Repair

There are two general classes of approaches to repairing defects using failing tests to identify faulty behavior and passing tests to demonstrate acceptable program behavior: *generate-and-validate* or *heuristic* repair and *semantic-based* repair. The former uses search-based techniques or predefined templates to generate many syntactic candidate patches, validating them against the tests (e.g., GenProg [49], Prophet [58], AE [107], HDRepair [46], ErrDoc [94], JAID [15], Qlose [19], and Par [39], among others). Techniques such as DeepFix [31] and ELIXIR [80] use learned models to predict erroneous program locations along with patches. ssFix [110] uses existing code that is syntactically related to the context of a bug to produce patches. CapGen [109] works at the AST node level (token-level) and uses context and dependency similarity (instead of semantic similarity) between the suspicious code fragment and the candidate code snippets to produce patches. To manage the large search space of candidates created because of using finer-level granularity, it extracts context information from candidate code snippets and prioritizes the mutation operators considering the extracted context information. SimFix [36] considers the variable name and method name similarity in addition to the structural similarity between the suspicious code and candidate code snippets. Similar to CapGen, it prioritizes the candidate modifications by removing the ones that are found less frequently in existing patches. Hercules [81] generalizes single-location program repair techniques to defects that require similar edits be made in multiple locations. Enforcing that a patch keeps a program semantically similar to the buggy version by ensuring that user-specified correct traces execute properly on the patched version can repair reactive programs with linear temporal logic specifications [98]. Several repair approaches have aimed to reduce syntactic or semantic differences between the buggy and patched program [19], [36], [38], [45], [63], [98], [109], with a goal of improving patch quality. For example, Qlose [19] minimizes a combination of syntactic and semantic differences between the buggy and patched programs while generating candidate patches. SketchFix [34] optimizes the candidate patch generation and evaluation by translating faulty programs to sketches (partial programs with holes) and lazily initializing the candidates of the sketches while validating them against the test execution. SOFix [50] uses 13 predefined repair templates to generate candidate patches. These repair templates are created based on the repair patterns mined from StackOverflow posts by comparing code samples in questions and answers for fine-grained modifications. SapFix [60] and Getafix [83], two tools deployed on production code at Facebook, efficiently produce repairs for large real-world programs. SapFix [60] uses prioritized repair strategies, including pre-defined fix templates, mutation operators, and bug-triggering change reverting, to produce repairs in realtime. Getafix [83] learns fix patterns from past code changes to suggest repairs for bugs that are found by Infer, Facebook's in-house static analysis tool.

SOSRepair's approach to using existing code to inform repair is reminiscent of Prophet [58], Par [39], IntPTI [16], and HDRepair [46] that use models of existing code to create or evaluate patches. SOSRepair does not use patterns, but rather considers a database of code snippets for candidate patches, using a constraint solver and existing test cases to assess them. The latter class of approaches use semantic reasoning to synthesize patches to satisfy an inferred specification (e.g., Nopol [112], Semfix [73], DirectFix [63], Angelix [64], S3 [45], JFIX [44]). SemGraft [62] infers specifications by symbolically analyzing a correct reference implementation (as opposed to using test cases), but unlike

SOSRepair, requires that reference implementation. Genesis [55], Refazer [79], NoFAQ [20], Sarfgen [100], and Clara [30] process correct patches to automatically infer code transformations to generate patches, a problem conceptually related to our challenge in integrating repair snippets to a new context.

SearchRepair [38] combines those classes, using a constraint solver to identify existing code to construct repairs. SOSRepair builds on SearchRepair, fundamentally improving the approach in several important ways. It is significantly more expressive (handling code constructs used in real code and reasoning about snippets that can affect multiple variables as output) and scalable (SearchRepair could only handle small, student-written C programs), supports deletion and insertion, uses failing test cases to restrict the search space, repairs code without passing examples, and its encoding of the repair query is significantly more expressive and efficient.

The location mechanism we adapt to repair queries was previously proposed for program synthesis [35] and adapted to semantic-based program repair [63], [64], [73]. Despite underlying conceptual similarities, SOSRepair differs from these approaches in key ways. Instead of replacing buggy expressions in if conditions or assignments with synthesized expressions, SOSRepair uses the constraint solver to *identify existing code* to use as patches, at a higher level of granularity than in prior work. Like SOSRepair, semantic-based approaches constrain desired behavior with failing test cases to guide patch synthesis. Critically, however, prior techniques require that the expected output on failing test cases be explicitly stated, typically through annotation. See, for example, `https://github.com/mechtaev/angelix/blob/master/doc/Manual.md`. SOSRepair automatically infers and uses the negative behavior extracted from the program state with no additional annotation burden.

Like SOSRepair, approaches that aim to generate higher-quality patches using a test suite are complementary to attempts to generate oracles to improve the test suite. For example, Swami processes natural-language specifications to generate precise oracles and tests, improving on both developer-written and other automatically-generated tests [69]. Similarly, Toradacu [29] and Jdoctor [9] generate oracles from Javadoc comments, and @tComment [92] generates preconditions related to nullness of parameters, each of which can lead to better tests. Regression test generation tools, e.g., EvoSuite [23] and Randoop [74], can help ensure patches do not alter otherwise-undertested functionality. UnsatGuided [114] generates regression tests using EvoSuite to constrain the repair process and produce fewer low-quality patches. However, automatically-generated tests often differ in quality from manually-written ones [84], [101], and have different effects on patch quality [85]. Specification mining uses execution data to infer (typically) FSM-based specifications [1], [5], [6], [7], [28], [41], [42], [43], [51], [52], [53], [54], [78], [82]. TAUTOKO uses such specifications to generate tests, e.g., of sequences of method invocations on a data structure [18], then iteratively improving the inferred model [18], [99]. Patch quality can also potentially improve using generated tests for non-functional properties, such as software fairness, which rely on observed behavior, e.g., by asserting that the behavior on inputs differing in a controlled way should be sufficiently similar [3], [11], [26]. Meanwhile, assertions on system data can also act as oracles [71], [72], and inferred causal relationships in data management systems [24], [65], [66] can help explain query results, debug errors [102], [103], [104], and suggest oracles for systems that rely on data management systems [67].

Our central goal is to improve the ability of program repair to produce correct patches. Recent work has argued for evaluating patch correctness using independent tests [47], [85], [111], [113], which is the approach we follow, as opposed to manual examination [57], [76]. Of the 22 defects for which SOSRepair produces patches, 9 pass all the independent tests, more than prior techniques. Improving fault localization, 16 of the patches SOSRepair$^{\oplus}$ produces pass all independent tests. This suggests that high-granularity, semantic-search-based repair can produce more high-quality patches, and that better fault localization can play an important role in improving repair quality.

6 Contributions

Automated program repair may reduce software production costs and improve software quality, but only if it produces high-quality patches. While semantic code search can produce high-quality patches [38], such an approach has never been demonstrated on real-world programs. In this paper, we have designed SOSRepair, a novel approach to using semantic code search to repair programs, focusing on extending expressiveness to that of real-world C programs and improving the search mechanism's scalability. We evaluate SOSRepair on 65 defects in large, real-world C programs, such as `php` and `python`. SOSRepair produces patches for 22 (34%) of the defects, and 9 (41%) of those patches pass 100% of independently-generated, held-out tests. SOSRepair repairs a defect no prior techniques have, and produces higher-quality patches. In a semi-automated approach that manually specifies the fault's location, SOSRepair patches 23 defects, of which 16 (70%) pass all independent tests. Our results suggest semantic code search is a promising approach for automatically repairing real-world defects.

Acknowledgments

This work is supported by the National Science Foundation under grants no. CCF-1453474, CCF-1563797, CCF-1564162, CCF-1645136, and CCF-1763423.

References

[1] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie, "Fault-scalable Byzantine fault-tolerant services," in *Proc. ACM Symp. Operating Syst. Principles*, 2005, pp. 59–74.

[2] afl fuzz, "American fuzzy lop," 2018. [Online]. Available: http://lcamtuf.coredump.cx/afl/

[3] R. Angell, B. Johnson, Y. Brun, and A. Meliou, "Themis: Automatically testing software for discrimination," in *Proc. Eur. Softw. Eng. Conf. and ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, Nov. 2018, pp. 871–875.

[4] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro, "The plastic surgery hypothesis," in *Proc. ACM SIGSOFT Symp. Found. Softw. Eng.*, Nov. 2014, pp. 306–317.

[5] I. Beschastnikh, Y. Brun, J. Abrahamson, M. D. Ernst, and A. Krishnamurthy, "Using declarative specification to improve the understanding, extensibility, and comparison of model-inference algorithms," *IEEE Trans. Softw. Eng.*, vol. 41, no. 4, pp. 408–428, Apr. 2015.

[6] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy, "Inferring models of concurrent systems from logs of their behavior with csight," in *Proc. IEEE/ACM Int. Conf. Softw. Eng.*, Jun. 2014, pp. 468–479.

[7] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst, "Leveraging existing instrumentation to automatically infer invariant-constrained models," in *Proc. Eur. Softw. Eng. Conf. and ACM SIGSOFT Symp. Found. Softw. Eng.*, Sep. 2011, pp. 267–277.

[8] S. Bhatia, P. Kohli, and R. Singh, "Neuro-symbolic program corrector for introductory programming assignments," in *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, May 2018, pp. 60–70.

[9] A. Blasi, A. Goffi, K. Kuznetsov, A. Gorla, M. D. Ernst, M. Pezzè, and S. D. Castellanos, "Translating code comments to procedure specifications," in *Proc. Int. Symp. Softw. Testing Anal.*, 2018, pp. 242–253.

[10] 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 *Proc. Eur. Softw. Eng. Conf. and ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, Sep. 2017, pp. 117–128.

[11] Y. Brun and A. Meliou, "Software fairness," in *Proc. Eur. Softw. Eng. Conf. and ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, Nov. 2018, pp. 754–759.

[12] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. USENIX Conf. Operating Syst. Des. Implementation*, 2008, pp. 209–224.

[13] A. Carzaniga, A. Gorla, A. Mattavelli, N. Perino, and M. Pezzè, "Automatic recovery from runtime failures," in *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, 2013, pp. 782–791.

[14] A. Carzaniga, A. Gorla, N. Perino, and M. Pezzè, "Automatic workarounds for web applications," in *Proc. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2010, pp. 237–246.

[15] L. Chen, Y. Pei, and C. A. Furia, "Contract-based program repair without the contracts," in *Proc. IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2017, pp. 637–647.

[16] X. Cheng, M. Zhou, X. Song, M. Gu, and J. Sun, "IntPTI: Automatic integer error repair with proper-type inference," in *Proc. IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2017, pp. 996–1001.

[17] C. Lattner, "Clang: A C language family frontend for LLVM," 2019. [Online]. Available: https://clang.llvm.org/

[18] V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller, "Generating test cases for specification mining," in *Proc. Int. Symp. Softw. Testing Anal.*, 2010, pp. 85–96.

[19] L. D'Antoni, R. Samanta, and R. Singh, "Qlose: Program repair with quantitative objectives," in *Proc. Int. Conf. Comput. Aided Verification*, Jul. 2016, pp. 383–401.

[20] L. D'Antoni, R. Singh, and M. Vaughn, "NoFAQ: Synthesizing command repairs from examples," in *Proc. Eur. Softw. Eng. Conf. and ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2017, pp. 582–592.

[21] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Proc. Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2008, pp. 337–340.

[22] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: A general approach to inferring errors in systems code," *ACM SIGOPS Operating Syst. Rev.*, vol. 35, no. 5, pp. 57–72, 2001.

[23] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Trans. Softw. Eng.*, vol. 39, no. 2, pp. 276–291, Feb. 2013.

[24] C. Freire, W. Gatterbauer, N. Immerman, and A. Meliou, "A characterization of the complexity of resilience and responsibility for self-join-free conjunctive queries," *Proc. VLDB Endowment*, vol. 9, no. 3, pp. 180–191, 2015.

[25] M. Gabel and Z. Su, "A study of the uniqueness of source code," in *Proc. SIGSOFT Int. Symp. Found. Softw. Eng.*, 2010, pp. 147–156.

[26] S. Galhotra, Y. Brun, and A. Meliou, "Fairness testing: Testing software for discrimination," in *Proc. Eur. Softw. Eng. Conf. and ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, Sep. 2017, pp. 498–510.

[27] Q. Gao, H. Zhang, J. Wang, Y. Xiong, L. Zhang, and H. Mei, "Fixing recurring crash bugs via analyzing Q&A sites," in *Proc. 30th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, Nov. 2015, pp. 307–318.

[28] C. Ghezzi, M. Pezzè, M. Sama, and G. Tamburrelli, "Mining behavior models from user-intensive web applications," in *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, 2014, pp. 277–287.

[29] A. Goffi, A. Gorla, M. D. Ernst, and M. Pezzè, "Automatic generation of oracles for exceptional behaviors," in *Proc. Int. Symp. Softw. Testing Anal.*, Jul. 2016, pp. 213–224.

[30] S. Gulwani, I. Radiček, and F. Zuleger, "Automated clustering and program repair for introductory programming assignments," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2018, pp. 465–480.

[31] R. Gupta, S. Pal, A. Kanade, and S. K. Shevade, "DeepFix: Fixing common C language errors by deep learning," in *Proc. Conf. Artif. Intell.*, 2017, pp. 1345–1351.

[32] T. Gvero and V. Kuncak, "Synthesizing Java expressions from free-form queries," in *Proc. ACM Int. Conf. Object Oriented Program. Syst. Lang. Appl.*, 2015, pp. 416–432.

[33] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, 2012, pp. 837–847.

[34] J. Hua, M. Zhang, K. Wang, and S. Khurshid, "Towards practical program repair with on-demand candidate generation," in *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, 2018, pp. 12–23.

[35] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," in *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, 2010, pp. 215–224.

[36] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *Proc. Int. Symp. Softw. Testing Anal.*, 2018, pp. 298–309.

[37] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proc. Int. Conf. Softw. Eng.*, 2002, pp. 467–477.

[38] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun, "Repairing programs with semantic code search," in *Proc. IEEE/ACM Int. Conf. Autom. Softw. Eng.*, Nov. 2015, pp. 295–306.

[39] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, 2013, pp. 802–811.

[40] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976.

[41] I. Krka, Y. Brun, G. Edwards, and N. Medvidovic, "Synthesizing partial component-level behavior models from system specifications," in *Proc. Eur. Softw. Eng. Conf. and ACM SIGSOFT Symp. Found. Softw. Eng.*, Aug. 2009, pp. 305–314.

[42] T.-D. B. Le, X. B. D. Le, D. Lo, and I. Beschastnikh, "Synergizing specification miners through model fissions and fusions," in *Proc. 30th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, Nov. 2015, pp. 115–125.

[43] T.-D. B. Le and D. Lo, "Beyond support and confidence: Exploring interestingness measures for rule-based specification mining," in *Proc. Int. Conf. Softw. Anal., Evolution, Reengineering*, 2015, pp. 331–340.

[44] X.-B. D. Le, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser, "JFIX: Semantics-based repair of Java programs via symbolic PathFinder," in *Proc. ACM Int. Symp. Softw. Testing Anal.*, 2017, pp. 376–379.

[45] 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 *Proc. Eur. Softw. Eng. Conf. and ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2017, pp. 593–604.

[46] X.-B. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *Proc. Int. Conf. Softw. Anal., Evolution, Reengineering*, Mar. 2016, vol. 1, pp. 213–224.

[47] X.-B. D. Le, F. Thung, D. Lo, and C. L. Goues, "Overfitting in semantics-based automated program repair," in *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, 2018, pp. 163–163.

[48] 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 Trans. Softw. Eng.*, vol. 41, no. 12, pp. 1236–1256, Dec. 2015.

[49] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A generic method for automatic software repair," *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, pp. 54–72, Jan./Feb. 2012.

[50] X. Liu and H. Zhong, "Mining StackOverflow for program repair," in *Proc. Int. Conf. Softw. Anal., Evolution, Reengineering*, 2018, pp. 118–129.

[51] D. Lo and S.-C. Khoo, "QUARK: Empirical assessment of automaton-based specification miners," in *Proc. Work. Conf. Reverse Eng.*, 2006, pp. 51–60.

[52] D. Lo and S.-C. Khoo, "SMArTIC: Towards building an accurate, robust and scalable specification miner," in *Proc. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2006, pp. 265–275.

[53] D. Lo and S. Maoz, "Scenario-based and value-based specification mining: Better together," in *Proc. IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2010, pp. 387–396.

[54] D. Lo, L. Mariani, and M. Pezzè, "Automatic steering of behavioral model inference," in *Proc. Eur. Softw. Eng. Conf. and ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2009, pp. 345–354.

[55] F. Long, P. Amidon, and M. Rinard, "Automatic inference of code transforms for patch generation," in *Proc. Eur. Softw. Eng. Conf. and ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2017, pp. 727–739.

[56] F. Long and M. Rinard, "Staged program repair with condition synthesis," in *Proc. Eur. Softw. Eng. Conf. and ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2015, pp. 166–178.

[57] F. Long and M. Rinard, "An analysis of the search spaces for generate and validate patch generation systems," in *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, 2016, pp. 702–713.

[58] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proc. ACM SIGPLAN-SIGACT Symp. Principles Program. Lang.*, 2016, pp. 298–312.

[59] C. Macho, S. McIntosh, and M. Pinzger, "Automatically repairing dependency-related build breakage," in *Proc. Int. Conf. Softw. Anal., Evolution, Reengineering*, 2018, pp. 106–117.

[60] A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, and A. Scott, "SapFix: Automated end-to-end repair at scale," in *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, May 2019, pp. 269–278.

[61] M. Martinez, W. Weimer, and M. Monperrus, "Do the fix ingredients already exist? An empirical inquiry into the redundancy assumptions of program repair approaches," in *Proc. ACM/IEEE Int. Conf. Softw. Eng. New Ideas Emerging Results Track*, 2014, pp. 492–495.

[62] S. Mechtaev, M.-D. Nguyen, Y. Noller, L. Grunske, and A. Roychoudhury, "Semantic program repair using a reference implementation," in *Proc. Int. Conf. Softw. Eng.*, 2018, pp. 129–139.

[63] S. Mechtaev, J. Yi, and A. Roychoudhury, "DirectFix: Looking for simple program repairs," in *Proc. Int. Conf. Softw. Eng.*, May 2015, pp. 448–458.

[64] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng.*, 2016, pp. 691–701.

[65] A. Meliou, W. Gatterbauer, J. Y. Halpern, C. Koch, K. F. Moore, and D. Suciu, "Causality in databases," *IEEE Data Eng. Bull.*, vol. 33, no. 3, pp. 59–67, Sep. 2010.

[66] A. Meliou, W. Gatterbauer, K. F. Moore, and D. Suciu, "The complexity of causality and responsibility for query answers and non-answers," *Proc. VLDB Endowment*, vol. 4, no. 1, pp. 34–45, 2010.

[67] A. Meliou, S. Roy, and D. Suciu, "Causality and explanations in databases," *Proc. VLDB Endowment Tut.*, vol. 7, no. 13, pp. 1715–1716, 2014.

[68] 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 *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, Jun. 2014, pp. 234–242.

[69] M. Motwani and Y. Brun, "Automatically generating precise oracles from structured natural language specifications," in *Proc. IEEE/ACM Int. Conf. Softw. Eng.*, May 2019, pp. 188–199.

[70] M. Motwani, S. Sankaranarayanan, R. Just, and Y. Brun, "Do automated program repair techniques repair hard and important bugs?" *Empirical Softw. Eng.*, vol. 23, no. 5, pp. 2901–2947, Oct. 2018.

[71] K. Muşlu, Y. Brun, and A. Meliou, "Data debugging with continuous testing," in *Proc. Eur. Softw. Eng. Conf. and ACM SIGSOFT Int. Symp. Found. Softw. Eng. New Ideas Track*, Aug. 2013, pp. 631–634.

[72] K. Muşlu, Y. Brun, and A. Meliou, "Preventing data errors with continuous testing," in *Proc. ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, Jul. 2015, pp. 373–384.

[73] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "SemFix: Program repair via semantic analysis," in *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, 2013, pp. 772–781.

[74] C. Pacheco and M. D. Ernst, "Randoop: Feedback-directed random testing for Java," in *Proc. Conf. Object-Oriented Program. Syst. Appl.*, 2007, pp. 815–816.

[75] Y. Qi, X. Mao, and Y. Lei, "Efficient automated program repair through fault-recorded testing prioritization," in *Proc. Int. Conf. Softw. Maintenance*, Sep. 2013, pp. 180–189.

[76] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proc. Int. Symp. Softw. Testing Anal.*, 2015, pp. 24–36.

[77] S. P. Reiss, "Semantics-based code search," in *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, 2009, pp. 243–253.

[78] S. P. Reiss and M. Renieris, "Encoding program executions," in *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, 2001, pp. 221–230.

[79] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann, "Learning syntactic program transformations from examples," in *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, 2017, pp. 404–415.

[80] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "ELIXIR: Effective object oriented program repair," in *Proc. IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2017, pp. 648–659.

[81] S. Saha, R. K. Saha, and M. R. Prasad, "Harnessing evolution for multi-hunk program repair," in *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, May 2019, pp. 13–24.

[82] M. Schur, A. Roth, and A. Zeller, "Mining behavior models from enterprise web applications," in *Proc. Eur. Softw. Eng. Conf. and ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2013, pp. 422–432.

[83] J. Bader, A. Scott, M. Pradel, and S. Chandra, "Getafix: Learning to fix bugs automatically," *Proc. ACM Program. Languages*, Object-Oriented Programming, Systems, Languages, and Applications, vol. 3, Oct. 2019.

[84] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do automatically generated unit tests find real faults? An empirical study of effectiveness and challenges," in *Proc. Int. Conf. Autom. Softw. Eng.*, Nov. 2015, pp. 201–211.

[85] E. K. Smith, E. Barr, C. Le Goues, and Y. Brun, "Is the cure worse than the disease? Overfitting in automated program repair," in *Proc. Eur. Softw. Eng. Conf. and ACM SIGSOFT Symp. Found. Softw. Eng.*, Sep. 2015, pp. 532–543.

[86] A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioğlu, "Programming by sketching for bit-streaming programs," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2005, pp. 281–294.

[87] F. Steimann, M. Frenkel, and R. Abreu, "Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators," in *Proc. Int. Symp. Softw. Testing Anal.*, 2013, pp. 314–324.

[88] K. T. Stolee and S. Elbaum, "Toward semantic search via SMT solver," in *Proc. ACM SIGSOFT Int. Symp. Found. Softw. Eng. New Ideas and Emerging Results Track*, 2012, pp. 25:1–25:4.

[89] K. T. Stolee, S. Elbaum, and D. Dobos, "Solving the search for source code," *ACM Trans. Softw. Eng. Methodology*, vol. 23, no. 3, pp. 26:1–26:45, May 2014.

[90] K. T. Stolee, S. Elbaum, and M. B. Dwyer, "Code search with input/output queries: Generalizing, ranking, and assessment," *J. Syst. Softw.*, vol. 116, pp. 35–48, 2016.

[91] S. H. Tan, Z. Dong, X. Gao, and A. Roychoudhury, "Repairing crashes in Android apps," in *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, 2018, pp. 187–198.

[92] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens, "@tComment: Testing Javadoc comments to detect comment-code inconsistencies," in *Proc. Int. Conf. Softw. Testing, Verification, Validation*, 2012, pp. 260–269.

[93] V. Terragni, Y. Liu, and S.-C. Cheung, "CSNIPPEX: Automated synthesis of compilable code snippets from Q&A sites," in *Proc. ACM Int. Symp. Softw. Testing Anal.*, 2016, pp. 118–129.

[94] Y. Tian and B. Ray, "Automatically diagnosing and repairing error handling bugs in C," in *Proc. Eur. Softw. Eng. Conf. and ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2017, pp. 752–762.

[95] C. Timperley, S. Stepney, and C. Le Goues, "Poster: BugZoo—A platform for studying software bugs," in *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, May 2018, pp. 446–447.

[96] Z. Tu, Z. Su, and P. Devanbu, "On the localness of software," in *Proc. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2014, pp. 269–280.

[97] R. van Tonder and C. L. Goues, "Static automated program repair for heap properties," in *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, 2018, pp. 151–162.

[98] C. von Essenz and B. Jobstmann, "Program repair without regret," *Formal Methods Syst. Des.*, vol. 47, no. 1, pp. 26–50, 2015.

[99] R. J. Walls, Y. Brun, M. Liberatore, and B. N. Levine, "Discovering specification violations in networked software systems," in *Proc. IEEE Int. Symp. Softw. Rel. Eng.*, Nov. 2015, pp. 496–506.

[100] K. Wang, R. Singh, and Z. Su, "Search, align, and repair: Data-driven feedback generation for introductory programming exercises," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2018, pp. 481–495.

[101] Q. Wang, Y. Brun, and A. Orso, "Behavioral execution comparison: Are tests representative of field behavior?" in *Proc. IEEE Int. Conf. Softw. Testing, Verification, Validation*, Mar. 2017, pp. 321–332.

[102] X. Wang, X. L. Dong, and A. Meliou, “Data X-Ray: A diagnostic tool for data errors,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2015, pp. 1231–1245.

[103] X. Wang, A. Meliou, and E. Wu, “QFix: Demonstrating error diagnosis in query histories,” in *Proc. Int. Conf. Manage. Data*, 2016, pp. 2177–2180.

[104] X. Wang, A. Meliou, and E. Wu, “QFix: Diagnosing errors through query histories,” in *Proc. ACM Int. Conf. Manage. Data*, 2017, pp. 1369–1384.

[105] Y. Wang, Y. Feng, R. Martins, A. Kaushik, I. Dillig, and S. P. Reiss, “Hunter: Next-generation code reuse for Java,” in *Proc. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2016, pp. 1028–1032.

[106] W. Weimer, “Patches as better bug reports,” in *Proc. Int. Conf. Generative Program. Component Eng.*, 2006, pp. 181–190.

[107] W. Weimer, Z. P. Fry, and S. Forrest, “Leveraging program equivalence for adaptive program repair: Models and first results,” in *Proc. IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2013, pp. 356–366.

[108] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, “Automatically finding patches using genetic programming,” in *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, 2009, pp. 364–374.

[109] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, “Context-aware patch generation for better automated program repair,” in *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, 2018, pp. 1–11.

[110] Q. Xin and S. P. Reiss, “Leveraging syntax-related code for automated program repair,” in *Proc. IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2017, pp. 660–670.

[111] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang, “Identifying patch correctness in test-based program repair,” in *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, 2018, pp. 789–799.

[112] J. Xuan, M. Martinez, F. Demarco, M. Clement, S. R. Lamelas Marcote, T. Durieux, D. Le Berre, and M. Monperrus, “Nopol: Automatic repair of conditional statement bugs in Java programs,” *IEEE Trans. Softw. Eng.*, vol. 43, no. 1, pp. 34–55, Jan. 2017.

[113] J. Yang, A. Zhikhartsev, Y. Liu, and L. Tan, “Better test cases for better automated program repair,” in *Proc. Eur. Softw. Eng. Conf. and ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2017, pp. 831–841.

[114] Z. Yu, M. Martinez, B. Danglot, T. Durieux, and M. Monperrus, “Alleviating patch overfitting with automatic test generation: A study of feasibility and effectiveness for the Nopol repair system,” *Empirical Softw. Eng.*, vol. 24, no. 1, pp. 33–67, Feb. 2019.

[115] H. Zhong and Z. Su, “An empirical study on real bug fixes,” in *Proc. Int. Conf. Softw. Eng.*, May 2015, pp. 913–923.

Afsoon Afzal received the MS degree in software engineering from the School of Computer Science, Carnegie Mellon University, in 2019. She is working toward the PhD degree in the School of Computer Science, Carnegie Mellon University. She is interested in applying automated quality assurance methods, including automated testing and repair to evolving and autonomous systems. More information is available at: http://www.cs.cmu.edu/~afsoona.

Manish Motwani received the MS degree from the College of Information and Computer Sciences, University of Massachusetts Amherst, in 2018. He is working toward the PhD degree in the College of Information and Computer Sciences, University of Massachusetts Amherst. His research involves studying large software repositories to learn interesting phenomena in software development and maintenance, and to use that knowledge to design novel automation techniques for testing and program repair. More information is available at: http://people.cs.umass.edu/~mmotwani/.

Kathryn T. Stolee received the BS, MS, and PhD degrees from the University of Nebraska-Lincoln. She is an assistant professor with the Department of Computer Science, North Carolina State University. She received an NSF CAREER award. Her research interests include program analysis, code search, and empirical studies. She is a member of the IEEE. More information is available at: http://people.engr.ncsu.edu/ktstolee/.

Yuriy Brun received the PhD degree from the University of Southern California, in 2008 and completed his postdoctoral work with the University of Washington, in 2012. He is an associate professor with the College of Information and Computer Sciences, University of Massachusetts Amherst. His research focuses on software engineering, self-adaptive systems, and testing software for fairness. He received an NSF CAREER award and an IEEE TCSC Young Achiever in Scalable Computing Award. He is a senior member of the IEEE and a distinguished member of the ACM. More information is available at: http://www.cs.umass.edu/~brun/.

Claire Le Goues received the BA degree in computer science from Harvard University and the MS and PhD degrees from the University of Virginia. She is an associate professor with the School of Computer Science, Carnegie Mellon University, where she is primarily affiliated with the Institute for Software Research. She received an NSF CAREER award. She is interested in constructing high-quality systems in the face of continuous software evolution, with a particular interest in automatic error repair. She is a member of the IEEE. More information is available at: http://www.cs.cmu.edu/~clegoues.