Better Code Search and Reuse for Better Program Repair

Qi Xin
Georgia Institute of Technology
Atlanta, GA, USA
qxin6@gatech.edu

Steven P. Reiss
Brown University
Providence, RI, USA
spr@cs.brown.edu

***Abstract*—A branch of automated program repair (APR) techniques look at finding and reusing existing code for bug repair. ssFix is one of such techniques that is syntactic search-based: it searches a code database for code fragments that are syntactically similar to the bug context and reuses such code fragments to produce patches. The keys to its success lie in the approaches it uses for code search and code reuse. We investigated the effectiveness of ssFix using the Defects4J bug dataset and found that its code search and code reuse approaches are not truly effective and can be significantly improved. Motivated by the investigation, we developed a new repair technique sharpFix that follows ssFix's basic idea but differs significantly in the approaches used for code search and code reuse. We compared sharpFix and ssFix on the Defects4J dataset and confirmed through experiments that (1) sharpFix's code search and code reuse approaches are better than ssFix's approaches and (2) sharpFix can do better repair. sharpFix successfully repaired a total of 36 Defects4J bugs and outperformed many existing repair techniques in repairing more bugs. We also compared sharpFix, ssFix, and four other techniques on another dataset Bugs.jar-ELIXIR. Our results show that sharpFix did better than others and repaired the largest number of bugs.**

***Index Terms*—automated program repair, code search, code reuse**

I. Introduction

An automated program repair (APR) technique can significantly save people time and effort by repairing a bug[1] automatically. Taking as input a faulty program and a fault-exposing test suite that the program failed, such a technique automatically modifies the faulty program and can produce a patched program that passes the test suite. A branch of APR techniques [1]–[5] adopt a search-based approach for patch generation: they define a space of patches generated from applying a pre-defined set of modifications on a set of suspicious locations of the faulty program identified by fault localization techniques [6], and then search in the space for a correct patch. The search space is often huge, and finding a correct patch within it is difficult [7].

To address the search space problem, one idea is to reuse existing code from existing programs [8], [9]. Through reusing such code, a repair technique avoids generating a large amount of artificial code to mitigate search space explosion. The recent technique ssFix [10] was built upon the idea. It performs syntactic code search to find existing code fragments (the candidates) that are similar to the bug context (the target) from a code database and reuses those code fragments to produce patches for bug repair. ssFix leverages the syntactic differences between the target and each candidate to produce patches. For a candidate that is similar to the target, the differences are small, and the search space is reduced.

[1]In this paper, we use "bug" and "fault" interchangeably.

The keys to ssFix's success lie in the approaches it uses for code search and code reuse. After investigating their effectiveness using the Defects4J bug dataset [11], we found that its approaches are not truly effective and can be significantly improved. ssFix's code search uses a target that contains the local context of the faulty statement as query (three statements as most) and searches for candidates that are of comparable sizes. It uses the same method for searching candidates from the local program (or the *local search*) and from the external repository (or the *global search*). This type of code search is inflexible: we empirically found that (1) for local search, it needs less context to find candidates related to the faulty statement itself, and for global search, it needs more context to find candidates related to the target method (the enclosing method of the target); and that (2) different methods should be used for the two types of search. For code reuse, ssFix uses three steps: code translation, code matching, and modification to generate patches. Its code translation translates a candidate by finding variable, method, and type identifiers used in the candidate that are related to those used in the target and then renaming the candidate identifiers. Identifying two identifiers as related or not is only based on their usage contexts (the enclosing expressions and statements) which is not enough; Its code matching is based on a tree matching algorithm with a list of matching rules and arbitrary thresholds used and is inflexible. Its modification can lead to a large set of patches generated as it involves modifying not only the faulty statement itself but also its context. Validating those patches however is expensive.

To address these problems, we developed a new APR technique sharpFix which follows ssFix's basic repair idea but uses different approaches for code search and code reuse. sharpFix uses different forms of target and candidate and different search methods for doing local and global search. Its code reuse uses different and improved methods for code translation, code matching, modification, and patch validation to address the problems mentioned above. The technical

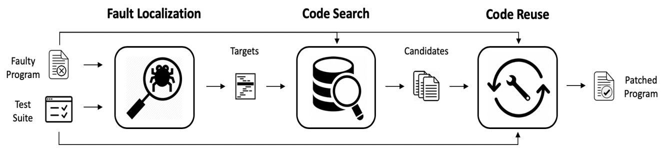

Fig. 1. An Overview of sharpFix

differences are non-trivial and actually lead to better code search, code reuse, and repair abilities that sharpFix has over ssFix. We confirmed these through our experiments using the Defects4J bug dataset. The results show that sharpFix repaired a total of 36 bugs with correct patches generated. We also compared sharpFix, ssFix, and four existing APR techniques: jGenProg [12], jKali [12], Nopol (version 2015) [13], and HDRepair [14] on another dataset: Bugs.jar-ELIXIR. The results show that sharpFix outperformed all these techniques and confirm again that sharpFix is an improvement over ssFix.

In this paper, we make the following contributions:

- A new APR technique sharpFix that finds and reuses existing code for automatic bug repair.
- An implementation of sharpFix (code available at *https://github.com/sharpFix18/sharpFix*).
- An evaluation of sharpFix on two bug datasets that shows sharpFix is an improvement over ssFix and outperforms four other repair techniques.

II. Overview

Figure 1 shows an overview of sharpFix's repair process. sharpFix takes in a faulty program and a fault-exposing test suite, and automatically generates as output a patched program that passes the test suite (or nothing if it cannot find one). The repair process is done in three stages: (1) ***fault localization***, (2) ***code search***, and (3) ***code reuse***. For fault localization, it uses ssFix's approach that leverages the spectrum-based technique GZoltar [15] to identify a list of suspicious statements that are likely to be buggy. Each statement is associated with a score ranged from 0 (non-suspicious) to 1 (highest-suspicious) that represents the likelihood of being buggy. The statements are ranked by scores from high to low. sharpFix next looks at each suspicious statement independently to produce patches for it. With a suspicious statement as the *target*, in the second stage, sharpFix does code search to find statements as the *candidate*s from both the local faulty program and an external code repository. These candidates are ranked based on their syntactic similarities with the target and the syntactic similarities between their contexts and the target's context. sharpFix looks at each candidate independently to produce patches for the target. In the third stage, sharpFix translates the candidate and its context by renaming the used variable, type, and method identifiers, matches code between the target and the translated candidate, produces patches for the target using the translated code, and validates the patches using the test suite. It reports as output the first validated patch whose corresponding patched program passes the test suite.

We will use the Defects4J bug M69 (Figure 2) as an example to explain how sharpFix works. For this bug, the developer

```
1 double r=correlationMatrix.getEntry(i, j);
2 double t=Math.abs(r * Math.sqrt((nObs - 2)/(1 - r * r)));
3 - out[i][j]=2 * (1 - tDistribution.cumulativeProbability(t));//target
4 + out[i][j]=2 * tDistribution.cumulativeProbability(-t);
```

Fig. 2. The M69 Bug and its Developer Patch

```
degreesOfFreedom=df(v1, v2, n1, n2);
distribution.setDegreesOfFreedom(degreesOfFreedom);
return 2.0 * distribution.cumulativeProbability(-t); //candidate
//return 2.0 * tDistribution.cumulativeProbability(-t); //translated
```

Fig. 3. A Code Fragment from the Bug's Local Program

patch changed the statement from line #3 to line #4 for calculating the correct matrix of p-values associated with a null hypothesis for Pearsons Correlation. For this bug, sharpFix's fault localization identified the statement at line #3 as the target for repair. The statement is ranked as No. 10.

III. Methodology

In this section, we first elaborate on sharpFix's code search and code reuse approaches and then summarize the technical differences between sharpFix and ssFix.

A. Code Search

In the code search stage, sharpFix takes in the target and does code search to generate as output a list of candidates. It uses different search methods to do local search and global search and merges the results.

For local search, sharpFix uses the target as the query code chunk. It does not use a larger code chunk as we found that a larger chunk that contains more context is more likely to be unique in the local program. For local search, sharpFix compares the target with every single statement as candidate in the faulty program. It calculates a score by comparing the tokens extracted from the names of variables, types, and methods used in the two statements to measure their syntactic similarity. To extract tokens from a statement, sharpFix first creates a list containing the original names of the variables, types, and methods used in the statement. It next looks at each name and splits it by camel-cases, underscores, and numbers. It further does stemming on these splitted tokens using the Porter Stemming algorithm [16]. Finally, it transforms each token in the list into lower-case. We call the tokens extracted as such the ***search*** tokens[2]. As an example, we show below the search tokens (in angle brackets) extracted from the target in Figure 2.

```
<out>,<i>,<j>,<tdistribution>,<t>,<distribution>
<cumulativeprobability>,<cumulative>,<probability>,<t>
```

With the lists of search tokens extracted from the target and the candidate, sharpFix calculates the Dice Similarity[3] of them as the score. sharpFix finally ranks the candidates by the calculated scores as the local search result.

For global search, sharpFix uses a different search method. It first finds Java methods from the code repository that are similar to the enclosing Java method of the target, or

[2]We compared in total three types of tokens in [17] (Section IV-B1).

[3]The original measure is used for sets. We slightly changed it to be used for lists.

the target method, using a search method similar to ssFix's. The differences are that here sharpFix uses Java methods as the query code chunks and uses three as the k-value for generating k-grams[4]. For each retrieved Java method m, sharpFix calculates a score s for it. sharpFix does not simply report these Java methods as the search results: a method can be big and reusing it can lead to too many patches generated. So instead, sharpFix uses its local search method to identify as candidates statements in m that are most similar to the target (it currently identifies two of such statements from m). Each such statement is associated with the retrieval score s. sharpFix finally ranks the candidates as the global result.

sharpFix merges the search results by first normalizing the scores of the candidates retrieved by local search and by global search separately and then ranking them all together. It selects the top-200 candidates as the code search output. For the target at line #3 of Figure 2, sharpFix did code search and retrieved a candidate from the local faulty program shown at line #3 of Figure 3. The rank of the candidate is No. 2.

B. Code Reuse

In this stage, sharpFix reuses each candidate retrieved in the previous stage independently to repair the target. This is done in four steps: *code translation*, *code matching*, *modification*, and *patch validation*.

1) Code Translation: As the first step, sharpFix translates the candidate by renaming variable, type, and method identifiers used in the candidate and its context, i.e., the enclosing method. Without doing so, it would often fail to directly transfer code from the candidate and its context to the target program for repair as there can be undeclared identifiers. The translation is done in three steps: (1) collecting identifiers in the candidate program and those in the target program, (2) identifying candidate and target identifiers that are related, and (3) renaming candidate identifiers as their related target identifiers.

For (1), sharpFix collects a list of candidate identifiers, or cids, as the variable, type, and method identifiers used in the candidate's enclosing method. An identifier we mention here is actually an identifier binding that represents for example a variable declaration and its use. sharpFix collects all identifiers in the candidate method scope since it may reuse code in that scope to produce patches. sharpFix collects a list of target identifiers, or tids, as the variable, type, and method identifiers used in the target's enclosing method and other identifiers that are accessible in the method: the declared field and method names in the target's enclosing class and the class name.

For (2), sharpFix creates a mapping that maps each cid to a tid identified as related. This is done in four steps: (a) it first maps cids to tids that share the same names that are not too short, i.e., with at least three characters; (b) if the candidate's enclosing method name cid is used in the method body and is unmapped, sharpFix maps it to the target's enclosing method name tid; (c) if the candidate's enclosing class name cid is used in the method body and is unmapped, sharpFix maps it to the target's enclosing class name tid; (d) sharpFix maps an unmapped cid to the tid that has most similar usage contexts; and (e) sharpFix maps an unmapped cid to the tid that shares the largest number of conceptual tokens extracted from their names measured by the Dice Similarity. For (c), an identifier's usage contexts are its parental expressions and statements in the AST structure. sharpFix compares the parameterized strings of two identifiers' usage contexts using ssFix's method (III-A(2) of [10]). The only difference is here sharpFix takes into account the results from (a) and (b): it does not parameterize the mapped identifiers from (a) and (b) for generating a usage context's parameterized string. As an example, one usage context of `tDistribution` is the method call at line #3, and the parameterized string sharpFix generates is `$v$.cumulativeProbability($v$)`. Note the method name is mapped in (a) and is not parameterized. For (d), to extract conceptual tokens, sharpFix first generates the search tokens (from Section III-A), and then filters away tokens that are Java keywords, stop words, too-short, and too-long (less than three and greater than 32 characters). Note that sharpFix checks the compatibility of two identifiers to make sure a variable cid is not mapped to a method tid for example. After a mapping is created, for (3), sharpFix simply renames a cid to its mapped tid.

For our example, sharpFix renames `distribution` (Figure 3, line #3) as `tDistribution` (Figure 2, line #3). The former is mapped to the latter due to a common conceptual token *distribution* they share.

2) Code Matching: sharpFix does not arbitrarily transfer code from candidate to target to produce patches: it does code matching in this step to match related statements and expressions from the target and the translated candidate, and in the next step it performs modifications based on the matched statements and expressions to produce patches.

sharpFix's code matching is based on comparing the search tokens (defined in Section III-A) and symbols (e.g., +) that it extracts from statements and expressions. We call the search tokens and symbols together the ***match*** tokens. We call the target $tchunk$ and the translated candidate $cchunk$. For code matching, sharpFix accepts $tchunk$ and $cchunk$ as input. As output, it produces a code mapping that maps each statement/expression in $tchunk$ to its matched statement/expression in $cchunk$. To create such a mapping, sharpFix starts by collecting two lists of statements and expressions $tses$ and $cses$ from $tchunk$ and $cchunk$ respectively (by visiting the ASTs). The collected expressions are non-trivial and do not include identifiers, number constants, or any of the four types of literals: *boolean*, *null*, *character*, and *string*. For each statement/expression tse in $tses$, sharpFix finds a cse in $cses$ that is compatible and shares the most match tokens with tse (measured by the Dice Similarity) and maps tse to cse.

When two ses (statements/expressions) are both statements, they are compatible if they are both loops. Otherwise, they

[4]To determine this, we compared several search methods in [17] (Section IV-B1).

need to have the same statement type[5] (e.g., both *return* statements) to be compatible. When two ses are both expressions, they are compatible if their expression types are equal. When one se is a statement and the other is an expression, they are only compatible if the statement's type is *VariableDeclarationStatement* and the expression's type is either *Assignment* or *VariableDeclarationExpression*.

For the bug example, sharpFix maps the right-hand side of the target to the returned expression of the translated candidate. The extracted match tokens and the similarity calculation are shown below.

```
Matched tokens from target (16 in total):
<2> <*> <(> <1> <-> <tdistribution> <t> <distribution> <.>
<cumulativeprobability> <cumulative> <probability> <(> <t> <)> <)>

Matched tokens from translated candidate (13 in total):
<2.0> <*> <tdistribution> <t> <distribution> <.>
<cumulativeprobability> <cumulative> <probability> <(> <-> <t> <)>

Overlapped tokens (11 in total):
<*> <tdistribution> <t> <distribution> <.> <cumulativeprobability>
<cumulative> <probability> <(> <t> <)>

Dice Similarity: (2*11)/(16+13)=0.759
```

3) Modification: sharpFix uses four modifications: *statement/expression replacement*, *method replacement*, *statement insertion*, and *adding if-guard* to transfer code from the candidate and its context to the target and its context to produce patches. It leverages ssFix's method for doing statement/expression replacement based on the matching result yielded in the previous step. If a statement/expression se from the target is mapped to a statement/expression se' from the candidate, sharpFix replaces se with se' to yield a patch. Note that it can yield more patches by replacing the components of se with those of se'. For method replacement, sharpFix replaces the target's enclosing method with the translated candidate's enclosing method to support making multiple changes within the method scope. For insertion, sharpFix looks at the translated candidate s' to which the target s is mapped, identifies the two neighboring statements of s' in its block: s'_0 and s'_1 that are before and after s', and inserts s'_0 before s and s'_1 after s to yield two patches. To produce patches using *adding if-guard*, sharpFix looks at the target s and its mapped candidate s'. If the parent of s' is an if-statement with a condition e', sharpFix creates new if-statements using the condition e' to guard s and other statements. Currently, sharpFix selects two sets of statements to be guarded: (1) s itself and (2) s plus the following statements its block.

For the bug example, sharpFix replaced the right-hand side of the target with the returned expression of the translated candidate to produce the correct patch.

4) Patch Validation: In the previous step, sharpFix does modification to generate patches. In this step, it validates the generated patches. To do this, sharpFix first removes patches that are syntactically duplicated and have already been validated before (from using other candidates). It next follows ssFix's approach to sort the patches by their sizes to possibly avoid reporting an overfitting patch [19], [31]. sharpFix next validates each sorted patch: It first applies the patch on the faulty program to generate a patched program; then does static analysis using S[6]'s method [20] to check whether the patched program can be resolved (to see for example whether it uses undeclared variables); then compiles the resolved program; and finally runs it against the test suite. sharpFix reports the first validated patch whose patched program passes the test suite. Such a patch is called a *plausible* patch [23]. For our example, sharpFix reported as output the correct patch generated in the previous step.

[5]The type of a statement/expression is its node type in the abstract syntax tree that sharpFix builds using the Eclipse JDT library [18].

C. Technical Differences Between sharpFix and ssFix

We next summarize the technical differences between the two techniques.

Code Search: ssFix uses the same form of code chunks for local and global search. The used code chunks contain at most three statements. sharpFix uses different forms of code chunks. For local search, it uses code chunks containing single statements, and for global search, it uses code chunks containing Java methods. ssFix uses the same search method for local and global search. The method sharpFix uses for local search is based on the extracted search tokens and is significantly different from ssFix's method. For global search, it reuses ssFix's method for finding Java methods from the code repository. It further uses its local search method for finding statements within the retrieved Java methods.

Code Translation: the main differences lie in the approaches the two techniques use for finding related identifiers between the candidate and the target. ssFix finds identifiers only within the scope of the two chunks, and identify related identifiers by their usage contexts. sharpFix finds identifiers within a larger scope: the enclosing methods (and classses) of the two chunks. The used approach for identifying related identifiers is more complicated: it not only compares two identifiers' usage contexts but also their compatibility, locations, names, lengths, and tokens extracted from their names.

Code Matching: ssFix uses a tree matching algorithm with non-trivial matching rules and human-created thresholds. This makes its code matching inflexible. For example, it does not allow two method calls to match unless the method names are identical, and this can hinder it from repairing an incorrect method call. sharpFix's code matching is based on token matching. It uses siginificantly simplified matching rules with no thresholds.

Modification & Patch Validation: For modification, ssFix uses statement/expression replacement, statement insertion, and statement deletion. sharpFix adds two new modifications: adding if-guard and method replacement. sharpFix does not do statement deletion as it was shown in [10] to be likely to produce defective patches. sharpFix uses ssFix's statement/expression replacement but modifies ssFix's statement insertion. This is because the target and candidate sharpFix uses both contain single statements. For patch validation, compared to ssFix, sharpFix performs static analysis as an additional step to identify invalid patches. sharpFix works more efficient by using such a step to filter away invalid patches without actually compiling them.

For the bug example, in the code search stage, ssFix produced code chunks including the target and the candidate statements with more contexts. Doing code search using such chunks, ssFix failed to find any code fragment that is useful for repair like the one in Figure 3. Even if ssFix could find that code fragment, it would still fail to generate the good translation by renaming `distribution` as `tDistribution`: comparing their usage contexts would not work in this case. Due to such failures, ssFix finally produced no patch.

IV. Evaluation

We compared sharpFix and ssFix on the Defects4J dataset [11]. The results showed that, compared to ssFix, sharpFix has better code search, code reuse, and repair abilities. On the Defects4J dataset, sharpFix produced correct patches for 36 bugs, whereas ssFix only produced correct patches for 22 bugs. We also compared sharpFix against ssFix and four other repair techniques (jGenProg, jKali, Nopol, and HDRepair) on Bugs.jar-ELIXIR [21], a dataset of 127 real Java bugs. The results show that sharpFix outperformed all these techniques.

A. Fix Ingredient Experiment

To evaluate sharpFix's and ssFix's code search and code reuse, we conducted a fix ingredient experiment to see whether the fix code exists for a bug. We identified a total of 103 Defects4J bugs whose developer patches (available from the dataset) are *simple*, i.e., all the fixing changes are made within an expression or a primitive statement that has no children statements. For a simple patch, we defined six types of fix ingredients. And for each of the 103 bugs, we identified the fix ingredient and checked whether it exists in a code database that consists of the local faulty program and a code repository for which we used the DARPA MUSE repository [22] that contains 66,341 Java projects (about 81 GB). More details can be found in [17]. Our results show that (1) for 50 (48.5%) of the 103 bugs, we retrieved the exact fix ingredients from the code database and (2) for 80 (77.7%) bugs, we retrieved fix ingredients in the parameterized forms. For parameterization, we replaced program-specific (non-JDK) variables, types, and methods with special symbols. We used the results as truths for the code search and code reuse experiments.

B. Code Search Comparison

For evaluation, we ran sharpFix's and ssFix's code search to see for how many of the 103 bugs, they can effectively retrieve candidate chunks containing the fix ingredients that we identified. We call a candidate chunk (possibly after translation) that contains the fix ingredient in its exact form ***promising***. Our results show that sharpFix and ssFix retrieved promising candidate chunks within the top-200 results for 42 and 37 bugs respectively.

a) Experiment: For each of the 103 bugs, we provided sharpFix with the faulty statement, ran its code search to retrieve a list of candidate statements from the code database, performed its translation to translate a candidate's enclosing method, and produced a code chunk. The code chunk includes

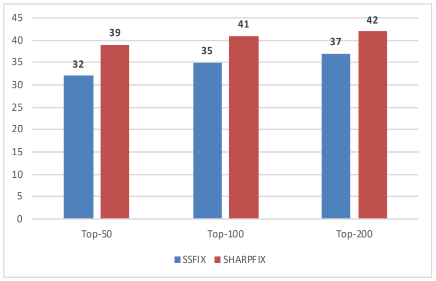

Fig. 4. The Retrieval of Promising Candidate Chunks that Contain the Fix Ingredients (Columns show the number of bugs for which promising candidate chunks were retrieved)

the candidate statement, its two neighbouring statements (used for insertion), and the enclosing if-condition (used for adding if-guard) if the enclosing statement is an if-statement. We then checked whether the code chunk is promising. To evaluate ssFix's code search, for each bug, we provided ssFix with the faulty statement, ran its code search to retrieve a list of candidate chunks each containing at most three statements, performed its translation to translate each chunk, and checked for promising chunks. For ssFix, we used the same five projects used in [10] as the local programs. For sharpFix, the local programs are the faulty programs for the 103 bugs. The code repository we used is the DARPA MUSE repository. For both experiments, we filtered away candidate chunks that are syntactically duplicated (they would be given the same rank) and those that are simply from the bug-fixed versions. We looked at the top-200 chunks as the retrieval results.

b) Result: Figure 4 shows the numbers of promising candidate chunks sharpFix and ssFix retrieved within the top-k results (with k being 50, 100, or 200). Within the top-200 results, sharpFix retrieved in total 59 chunks that contain the fix ingredients in the parameterized forms, among which, 42 are promising, i.e., contain the exact fix ingredients after translation. Our fix ingredient experiment shows that for as many as 80 bugs, the fix ingredients in the parameterized forms exist. So sharpFix retrieved promising fix ingredients for 42/80=52.5% bugs. We found that ssFix retrieved promising candidate chunks for 37 bugs, and it retrieved promising fix ingredients for 37/80=46.3% bugs. Compared to ssFix's code search, sharpFix's code search retrieved five more promising candidate chunks within the top-200 results, and it retrieved 39 promising chunks within the top-50 results which are more than all the promising chunks ssFix retrieved within the top-200 results.

Our results show that sharpFix's code search is better: It retrieved promising candidate chunks that contain the exact fix ingredients for 52.5% bugs while ssFix retrieved promising candidate chunks for 46.3% bugs.

C. Code Reuse Comparison

To evaluate sharpFix's and ssFix's code reuse, we wanted to see how many of the retrieved candidate chunks that contain fix ingredients can be successfully reused by the two techniques for producing correct patches. To do this, for sharpFix, we looked at the 59 bugs for which sharpFix retrieved candidate chunks that contain the parameterized fix ingredients. For each such bug, we provided sharpFix with the target and the retrieved candidate (the statements), and ran its reuse automatically. If sharpFix produced a plausible patch, we manually checked whether the patch is correct. Our results show that sharpFix produced 30 plausible patches which are all correct. It successfully reused 30/59=50.8% candidate chunks.

The exact fix ingredients (without any translation) are contained in 39 candidate chunks, and we expect sharpFix to be able to reuse those fix ingredients in producing the correct patches. For the other 20 (59-39) chunks which only contain the fix ingredients in the parameterized forms, we identified only three chunks that can be reasonably reused for repair: it may not be reasonable for a repair technique to translate an arbitrary, parameterized fix ingredient into the exact one to be reused for repair. We analyzed sharpFix's failures in reusing the chunks for repairing the 12 (39+3-30) bugs and found that the candidate chunks, though containing the fix ingredients, are not ideal for repairing 9 bugs. As an example, for the bug Cl92, the target statement `int indexOfDot = namespace.indexOf('.')` to be repaired uses the incorrect method call `indexOf`. sharpFix found the candidate statement as a while-loop containing the fix ingredient `namespace.lastIndexOf('.')` (after translation) in the loop body, but it also uses `namespace.indexOf('.')>0` as the loop condition. In reusing the candidate for repair, by code matching, sharpFix matched the incorrect method call in the target statement with the loop condition in the candidate statement and therefore missed the opportunity of reusing `namespace.lastIndexOf('.')` to repair the bug. Though it is possible to make sharpFix's code matching more sophisticated, we think a better solution for this case is to find a better candidate statement that contains the correct method call like the statement in the loop body. So we consider the candidate chunk as not ideal for this case. To successfully reuse the candidate chunks to repair the other 3 bugs, sharpFix's modification needs to be more sophisticated.

For comparison, we also evaluated ssFix's code reuse. By code search, ssFix retrieved 57 candidate chunks that contain parameterized fix ingredients. For each of the 57 bugs, we provided ssFix with the retrieved candidate chunk and ran its code reuse. Our results show that ssFix produced 25 plausible patches among which 23 are correct. It successfully reused 23/57=40.4% candidate chunks. We found the exact fix ingredients are contained in 37 chunks. For the other 20 (57-37) chunks which only contain the fix ingredients in the parameterized forms, we manually determined whether they can be reasonably reused. We identified only 4 of such chunks. We analyzed the failures of ssFix in reusing the 18 (37+4-23) reasonable chunks for producing the correct patches. We found that 7 candidate chunks are not ideal for repair. ssFix yielded bad candidate translations for 3 cases, it created bad code matching results for 2 cases, and its modifications are not sophisticated enough for producing the correct patches for 6 cases.

TABLE I
REPAIRING THE DEFECTS4J BUGS

Project (#Bugs)	sharpFix						ssFix					
	Time (min.)				#P	#C	Time (min.)				#P	#C
	Min	Max	Med	Avg			Min	Max	Med	Avg		
C (26)	0.8	115.7	7.2	19.2	9	4	1	80.7	12.4	20.7	7	2
Cl (133)	1.8	96.1	21.3	26	17	4	2.5	54.9	10.1	16.3	14	2
M (106)	0.7	118.5	11.3	33.2	33	13	1	119.3	14.7	30.2	26	8
T (27)	1.6	30	12.2	15.1	5	0	1.4	37.3	7.5	13.5	4	0
L (65)	0.8	116.1	4.8	18	25	15	0.8	117.8	4.3	13.1	18	10
Sum (357)	0.7	118.5	11.3	25.1	89	36	0.8	119.3	10.1	21	69	22

We show the projects in their abbreviations: C is JFreeChart; Cl is Closure Compiler; M is Commons Math; T is Joda-Time; and L is Commons Lang. #P and #C are the respective numbers of the plausible and correct patches generated.

TABLE II
REPAIRING BUGS.JAR-ELIXIR BUGS (SHARPFIX & SSFIX)

Project (#Bugs)	sharpFix						ssFix					
	Time (min.)				#P	#C	Time (min.)				#P	#C
	Min	Max	Med	Avg			Min	Max	Med	Avg		
ACC (10)	1.2	1.2	1.2	1.2	1	1	1.3	4.1	2.7	2.7	2	1
CML (16)	39.2	46.5	42.9	42.9	2	1	35.6	118.1	69.6	73.3	4	2
FLK (7)	0.8	0.8	0.8	0.8	1	1	6.9	6.9	6.9	6.9	1	1
OAK (31)	2.6	98.5	11.2	28.3	10	0	0.6	111.2	5.4	21.4	14	1
MAT (21)	0.8	103.4	26.2	32.2	10	6	0.8	64.9	11.5	17.2	9	5
MNG (5)	32.4	32.4	32.4	32.4	1	0	0.6	0.6	0.6	0.6	1	0
WCT (37)	0.9	91.4	8.4	21.6	14	6	3	82.8	8.7	22.1	12	1
Sum (127)	0.8	103.4	12.8	26.3	39	15	0.6	118.1	8.5	23.9	43	11

We show the projects in their abbreviations: ACC is Accumulo; CML is Camel; FLK is Flink; OAK is Jackrabbit Oak; MAT is Commons Math; MNG is Maven; and WCT is Wicket. #P and #C are the respective numbers of the plausible and correct patches generated.

Our results show sharpFix's code reuse is better than ssFix's: sharpFix reused 50.8% of the candidate chunks it retrieved for successful repair while ssFix only reused 40.4% candidate chunks.

D. Repair

We ran sharpFix and ssFix to repair all the 357 Defects4J bugs automatically. We also ran sharpFix, ssFix, and four other repair techniques jGenProg [12], jKali [12], Nopol (version 2015) [13], and HDRepair [14] automatically to repair bugs in another dataset Bugs.jar-ELIXIR created by Saha et al. [21] that contains 127 real bugs. We set the time and memory budgets for repairing each bug as two hours and 8 GB for all experiments. We ran all the experiments on a machine with 32 Intel-Xeon-2.6GHz CPUs and 128 GB memory. Given that jGenProg and HDRepair use randomness for patch generation, we ran each technique in three trials to repair a bug. Despite the number of trials, we believe our results are sufficient to show that sharpFix outperforms the two tools: it generated more than 10 correct patches in one trial than the tools did in three trials. We did not compare sharpFix to many other repair techniques that are written for C (e.g., SearchRepair [8], Code Phage [9], Prophet [23], and Angelix [24]) or are not publicly available (e.g., PAR [3]) including ELIXIR [21].

The results for the Defects4J bugs are shown in Table I. sharpFix produced in total 89 plausible patches with me-

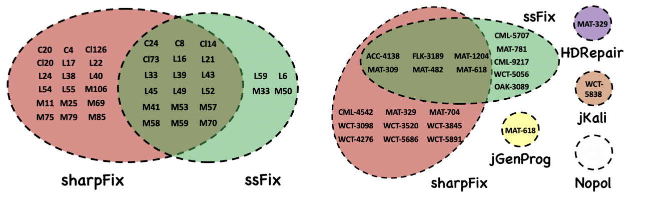

Fig. 5. The Overlap of Correctly Patched Bugs (Left: Defects4J; Right: Bugs.jar-ELIXIR)

TABLE III
REPAIRING BUGS.JAR-ELIXIR BUGS (ALL TECHNIQUES)

Tool	Time (min.)				#Plausible	#Correct
	Min	Max	Med	Avg		
sharpFix	0.8	103.4	12.8	26.3	39	15
ssFix	0.6	118.1	8.5	23.9	43	11
jGenProg	1.8	61.9	14.6	20.7	5	1
jKali	1.2	32.7	21.6	18.8	6	1
Nopol	4.3	29	9.5	12.6	8	0
HDRepair	93.8	108.1	101	101	2	1

dian and average times of producing a patch being about 11 and 25 minutes respectively. Among the 89 patches, 36 are correct. We manually determined the correctness of a plausible patch by comparing it to the developer patch and checking whether the two patches are semantics-equivalent. Compared to sharpFix, ssFix produced 69 plausible patches among which 22 are correct. The running times of the two techniques are comparable. With better code search and code reuse abilities, sharpFix works significantly better than ssFix in repairing 14 more bugs with correct patches generated. As shown in Figure 5 (the left one), it produced 18 correct patches that ssFix failed to produce. Since the experiment of repairing all 357 Defects4J bugs is expensive, we did not run jGenProg, jKali, and HDRepair for comparison. Given that ssFix outperformed these techniques on this dataset [10], we believe sharpFix would also outperform them.

Table II shows the repairing results of sharpFix and ssFix for each of the 7 projects contained in the Bugs.jar-ELIXIR dataset and for all of them. According to the result table, sharpFix and ssFix have comparable results for six of the projects. For WCT, however, sharpFix does significantly better with five more correct patches generated. For those WCT bugs for which sharpFix successfully repaired while ssFix did not, we found sharpFix effectively retrieved the key candidates: For five of the bugs that ssFix failed to repair, sharpFix looked at no more than 8 candidates to yield the correct patches (for the other bug WCT-5686, it found a candidate ranked 31th).

```
1  //WCT-5891 bug: substring(0,5) should be changed to substring(0,6)
2  int firstDigits=Integer.parseInt(creditCardNumber.substring(0,5));
3  if (firstDigits>=622126 && firstDigits<=622925) {
4    return CreditCard.CHINA_UNIONPAY; }
5
6  //sharpFix's candidate
7  int firstSixDigits=Integer.parseInt(creditCardNumber.substring(0,6));
8
9  //ssFix's candidate
10 int firstDigits=Integer.parseInt(creditCardNumber.substring(0,3));
11 if (firstDigits>=300 && firstDigits<=305) {
12   return CreditCard.DINERS_CLUB_CARTE_BLANCHE; }
```

As an example, for WCT-5891, using the faulty statement at line #2, sharpFix retrieved the fix statement at line #7 from the local program that contains the correct argument (integer `6`) for the method call `substring`. Using the local context of the faulty statement, ssFix found a candidate that is similar to the context but does not contain the correct integer argument. Using such a candidate, ssFix produced an overfitting patch by modifying not the faulty statement but the neighbouring if-statement: the condition at line #3.

```
1 //M33 bug: maxUlps should be changed to epsilon
2 if (Precision.compareTo(entry, 0d, maxUlps)>0) {
3 + if (Precision.compareTo(entry, 0d, epsilon)>0) { //sharpFix's patch
4   columnsToDrop.add(i); }}
```

Although sharpFix's approaches are more effective overall, there are cases where sharpFix failed to produce correct patches that ssFix produced. As an example, for the bug M33, ssFix and sharpFix both targeted the statement at line #4 for repair. ssFix produced a target chunk including the if-statement at line #2, and successfully found another if-statement that contains the correct usage of `comparedTo` and produced the correct patch. Since sharpFix failed to include the if-condition in the target chunk, it missed the opportunity of repairing the condition. It finally produced an overfitting patch by using an if-condition to guard the statement.

Table III shows the repairing results of all the six techniques. We found that compared to sharpFix and ssFix, the other four techniques have limited repair abilities. They each produced correct patches for no more than one bug. jGenProg only looks at finding the fix ingredients as statements from the local faulty program. This type of repair constrains itself from finding useful fix ingredients that are expressions and are from non-local programs. jKali can only do deletions and is unable to produce many types of non-deletion patches. Nopol looks at producing if-condition-related patches and is prone to synthesizing if-conditions that are either too constrained or too loose. HDRepair leverages mined bug-fixing changes to guide the search of a correct patch. However, according to our results, this type of guidance is not effective.

Our results show sharpFix is better than ssFix in successfully repairing 14 more Defects4J bugs and 5 more Bugs.jar-ELIXIR bugs, and it outperforms the other four techniques in repairing many more bugs.

All the experimental results can be found at https://github.com/sharpFix18/sharpFix/tree/master/expt0.

V. THREATS TO VALIDITY

To determine patch correctness, one of the authors manually analyzed each generated plausible patch and determined it to be correct if (1) the patch made changes at the right locations where changes in the developer patch were made and (2) there was a relatively obvious semantics-preserving transformation between the patch and the developer patch. We released all the generated plausible patches and explained each patch identified as correct as to why. Identifying patches that are semantically equivalent is in general challenging, and it is possible that there are patches that are indeed semantics-equivalent to the developer patches but made changes at locations not targeted by the developer patches and thus were not identified as correct. Understanding the failure cases of sharpFix's and ssFix's reuse is also based on manual analysis and could also be biased. For example, it might not be clear whether a failure was due to a weak code matching or a weak modification. We also released the reuse results. We compared sharpFix and ssFix on two bug datasets and found sharpFix to be better than ssFix. It is possible to have results different from ours using other datasets.

VI. RELATED WORK

sharpFix finds and reuses existing code from a code database for repair. It follows ssFix's basic idea [10] but uses different approaches for code search and reuse. sharpFix is closely related to SearchRepair [8] and Code Phage [9] which also do code search to find existing code for bug repair. Different from sharpFix which performs syntactic code search, SearchRepair's code search is based on symbolic execution and constraint-solving, and Code Phage's code search is based on program execution. CSAR [25] is similar to SearchRepair but performs string matching on constraints rather than doing constraint-solving to identify semantics-related code. sharpFix is also related to SimFix [26] which leverages similar code to produce patches. Different from sharpFix, SimFix also leverages existing patches to build the search space, and it only looks at the local program for finding similar code. The syntactic features used by the two techniques for finding similar code are also different. GenProg [1], [27] is an early APR technique that is related to sharpFix. It uses a genetic algorithm to reuse code from the faulty program itself to produce patches.

sharpFix is related to many repair techniques that use equivalence analysis and cost model [2], human-written templates [3], bug-fixing instances [28], [29], program comparison [30], program synthesis [24], [31]–[34], condition synthesis [35], [36], modifications with patch ranking models [21], [23], [37], learned transformations [38], [39], reference implementation [40], and non-test-suite specifications [41].

VII. Conclusion and Future Work

The success of a search-based APR technique like ssFix hinges on its abilities in accurately finding the right code for bug-fixing and effectively reusing it to produce the correct patch. We identified ssFix's weakness in doing code search and code reuse, developed sharpFix which uses improved code search and reuse approaches, and demonstrated that it can do better repair. Our future work will look at evaluating sharpFix on more bug datasets and comparing it with more APR techniques.

References

[1] C. L. Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A generic method for automatic software repair," *TSE*, pp. 54–72, 2012.

[2] W. Weimer, Z. P. Fry, and S. Forrest, "Leveraging program equivalence for adaptive program repair: models and first results," in *ASE*, 2013, pp. 356–366.

[3] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *ICSE*, 2013, pp. 802–811.

[4] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *ICSE*, 2014, pp. 254–265.

[5] X. B. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *SANER*, 2016, pp. 213–224.

[6] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *TSE*, pp. 707–740, 2016.

[7] F. Long and M. Rinard, "An analysis of the search spaces for generate and validate patch generation systems," in *ICSE*, 2016, pp. 702–713.

[8] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun, "Repairing programs with semantic code search (t)," in *ASE*, 2015, pp. 295–306.

[9] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard, "Automatic error elimination by horizontal code transfer across multiple applications," in *PLDI*, 2015, pp. 43–54.

[10] Q. Xin and S. P. Reiss, "Leveraging syntax-related code for automated program repair," in *ASE*, 2017, pp. 660–670.

[11] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in *ESEC/FSE*, 2014, pp. 437–440.

[12] "SpoonLabs Astor," https://github.com/SpoonLabs/astor.

[13] "SpoonLabs Nopol," https://github.com/SpoonLabs/nopol.

[14] HDRepair, "HDRepair repository," https://github.com/xuanbachle/bugfixes, 2016.

[15] J. Campos, A. Riboira, A. Perez, and R. Abreu, "GZoltar: an eclipse plug-in for testing and debugging," in *ASE*, 2012, pp. 378–381.

[16] M. F. Porter, "An algorithm for suffix stripping," *Program*, pp. 130–137, 1980.

[17] "Revisiting ssFix for better program repair," 2018. [Online]. Available: https://bit.ly/2M9ec7E

[18] "Eclipse JDT," https://www.eclipse.org/jdt.

[19] E. K. Smith, E. T. Barr, C. L. Goues, and Y. Brun, "Is the cure worse than the disease? overfitting in automated program repair," in *ESEC/FSE*, 2015, pp. 532–543.

[20] S. Mechtaev, J. Yi, and A. Roychoudhury, "DirectFix: Looking for simple program repairs," in *ICSE*, 2015, pp. 448–458.

[21] S. P. Reiss, "Semantics-based code search," in *ICSE*, 2009, pp. 243–253.

[22] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *POPL*, 2016, pp. 298–312.

[23] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "Elixir: effective object oriented program repair," in *ASE*, 2017, pp. 648–659.

[24] DARPA MUSE, "DARPA MUSE repository," https://www.darpa.mil/program/mining-and-understanding-software-enclaves, 2016.

[25] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *ICSE*, 2016, pp. 691–701.

[26] A. Hill, C. S. Păsăreanu, and K. T. Stolee, "Automated program repair with canonical constraints," in *ICSE-Companion*, 2018, pp. 339–341.

[27] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *ISSTA*, 2018, pp. 298–309.

[28] C. L. Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: fixing 55 out of 105 bugs for $8 each," in *ICSE*, 2012, pp. 3–13.

[29] Q. Gao, H. Zhang, J. Wang, Y. Xiong, L. Zhang, and H. Mei, "Fixing recurring crash bugs via analyzing q&a sites (t)," in *ASE*, 2015, pp. 307–318.

[30] X. Liu and H. Zhong, "Mining stackoverflow for program repair," in *SANER*, 2018, pp. 118–129.

[31] S. H. Tan and A. Roychoudhury, "relifix: Automated repair of software regressions," in *ICSE*, 2015, pp. 471–482.

[32] L. D'Antoni, R. Samanta, and R. Singh, "Qlose: Program repair with quantitative objectives," in *CAV*, 2016, pp. 383–401.

[33] 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 *ESEC/FSE*, 2017, pp. 593–604.

[34] R. Singh, S. Gulwani, and A. Solar-Lezama, "Automated feedback generation for introductory programming assignments," in *PLDI*, 2013, pp. 15–26.

[35] J. Xuan, M. Martinez, F. DeMarco, M. Clément, S. Lamelas, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs," *TSE*, pp. 34–55, 2016.

[36] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise condition synthesis for program repair," in *ICSE*, 2017, pp. 416–426.

[37] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *ICSE*, 2018, pp. 1–11.

[38] R. Rolim, G. Soares, D. Loris, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann, "Learning syntactic program transformations from examples," in *ICSE*, 2017, pp. 404–415.

[39] F. Long, P. Amidon, and M. Rinard, "Automatic inference of code transforms for patch generation," in *ESEC/FSE*, 2017, pp. 727–739.

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

[41] R. van Tonder and C. Le Goues, "Static automated program repair for heap properties," in *ICSE*, 2018, pp. 151–162.