LSRepair: Live Search of Fix Ingredients for Automated Program Repair

Kui Liu, Anil Koyuncu, Kisub Kim, Dongsun Kim, Tegawendé F. Bissyandé,
Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg, Luxembourg
{kui.liu, anil.koyuncu, kisub.kim, dongsun.kim, tegawende.bissyande}@uni.lu

***Abstract*—Automated program repair (APR) has extensively been developed by leveraging search-based techniques, in which fix ingredients are explored and identified in different granularities from a specific search space. State-of-the approaches often find fix ingredients by using mutation operators or leveraging manually-crafted templates. We argue that the fix ingredients can be searched in an online mode, leveraging code search techniques to find potentially-fixed versions of buggy code fragments from which repair actions can be extracted. In this study, we present an APR tool, LSRepair, that automatically explores code repositories to search for fix ingredients at the method-level granularity with three strategies of similar code search. Our preliminary evaluation shows that code search can drive a faster fix process (some bugs are fixed in a few seconds). LSRepair helps repair 19 bugs from the Defects4J benchmark successfully. We expect our approach to open new directions for fixing multiple-lines bugs.**

***Index Terms*—Program repair, code search, fix ingredients.**

I. Introduction

Automated program repair holds the promise of reducing the manual debugging effort by automatically suggesting patches for identified bugs. To date, most of the state-of-the-art approaches implement a generate-and-validate process where the fix is eventually selected or created after searching among several possibilities (which form the *search space*) [1], [2]. For example, GenProg [3] uses genetic programming to apply a sequence of mutation operations on a buggy source code until any generated patch passes the given tests. Since then, several directions to APR have been explored [4]–[29]. Many of the proposed approaches [3]–[5], [7]–[13] generate patch candidates by applying predefined mutations on the suspicious locations detected by fault localization techniques (e.g., Zoltar [30]). Experimental results have shown that such search-based APR techniques can fix a wide range of bugs and be scalable to large programs without extra specifications [31].

Unfortunately, although existing search-based APR techniques have achieved promising results, there still exist two important issues: (1) the correct patches are not always in the search space defined by each APR approach, which makes it impossible to fix the corresponding bugs successfully even after exploring the whole space; (2) the search space can be exploded. Widening the search space can indeed increase the probability of including the correct patches, but it will reduce the probability to find them earlier. A larger search space will also increase the probability to generate plausible patches [32], which increases probability of accidentally blocking the search for correct patches [13], [23].

Reflecting on the aforementioned threats, recent state-of-the-art APR approaches build on the assumption that the adequate fix ingredients can be found in existing code bases (e.g., open source repositories) [5], [7]–[10], [19], [33]. Nevertheless, most approaches still rely on simple patterns (e.g., Modify If-statement expression in Nopol [28]) or on templates (e.g., manually written as in PAR [7] or systematically learned as in Prophet [12] and FixMiner [34]), instead of leveraging a fix ingredients as it is. There is currently a research effort on extracting fix ingredients by leveraging abstract syntax tree differencing [35]–[37]. Unfortunately, AST diff patterns offer fine-grained, high-level repair actions, which will rapidly increase the search space and thus further aggravate the issue of search space explosion [13].

We note that, in practice, many programs consist of developing routines, data structures, and designs that are also implemented by other programs [16], [38], [39]. Developers are indeed recurrently writing code to address similar tasks, or cloning (e.g., via copy/paste) other code. Our intuition is that **while some code may be buggy, the similar code may have been fixed**. Recent APR approaches start with this intuition as well: symbolic execution-based approaches [40] would use reference implementations to drive the search for fixes; code-search-based approaches such as SearchRepair [20] perform by encoding a large database of human-written code fragments as SMT constraints on input-output behavior.

In this paper, we investigate the potential of using code search techniques on-the-fly to implement a APR tool, LSRepair. This approach leverages a live search of fix ingredients in real-world code bases to fix bugs automatically. Concretely, we limit the threat to space explosion by focusing on fix ingredients at the method-level instead of the statement-level, which are often used by other APR tools. The fixing process, however, may iterate over entities at the statement level.

The main contributions of this paper are as follows:

- We investigate the potential of code-to-code search techniques, clone detection, and semantic code search techniques, to rapidly produce relevant fix ingredients for APR.
- We particularly assess the proportion of bugs in the Defects4J benchmark that can be automatically fixed by a simple APR prototype implementation based on live code search. Concretely, we show how we managed to repair 19 bugs from the Defects4J bugs, including correctly fixing 10 bugs that no state-of-the-art tool was reported to have fixed.

- We present a discussion on the research challenges to be addressed towards building a scalable and effective APR technique based on state-of-the-art code search tools.

II. Background and Related work

A. Code Clone & Code Search

The applications of state-of-the-art code clone techniques [41]–[44] have shown that code clones are pervasive. In addition, several other empirical studies have confirmed that code changes are repeatedly performed in software code bases [45]–[48]. Indeed, same changes are prevalent because multiple occurrences of the same bug (e.g., due to copy/paste clones) require the same change. Similarly, when an API evolves, or when migrating to a new library/framework, all client code fragments must be adapted by the same collateral changes [49]. More recently Kim et al. have proposed FaCoY [50], a code-to-code search engine for real-world code fragments based on query alternation, and discussed the possibility to find out similar patch code for the bugs in Defects4J. Thus, we leverage FaCoY in one of our search strategies to find fix ingredients for automated repair.

B. Software Transplantation and APR with Code Search

Our work is largely related to the field of software transplantation that aims at transforming a functionality from a donor program to a recipient program. Software transplantation is often used in repair. For example, CodePhage [51] can fix common program errors (e.g., out of bounds, divide by zero, etc.), which involve missing checks, by finding donor code with error checking and transplanting it to a recipient buggy code. Trying to copy code is however challenging as data structures and name space may significantly vary between donor and recipient code, thus it may require an intermediate code-independent representation.

To work around this issue, SearchRepair [20] proposes to leverage semantic code search to guide repair. This technique utilizes static analysis to build a searchable database of open-source code fragments that describe behaviors as a set of SMT constraints. The technique then leverages dynamic analysis to identify candidate faulty regions in a program, and construct input-output behavior profiles. We argue that such an approach remains expensive and can only be exercised for small and trivial code fragments. A fully static and live code search approach, as we proposed in this work, may allow to address more real-world bugs as the ones in Defects4J.

III. Live Search of fix ingredients

Figure 1 shows the workflow of the proposed approach, LSRepair. In our work, we use information of fault localization at different levels. This section presents our methodology of searching and leveraging fix ingredients to repair bugs.

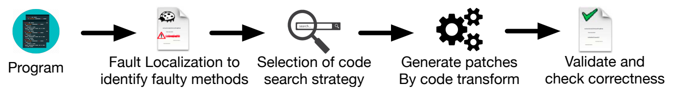
Program
Fault Localization to identify faulty methods
Selection of code search strategy
Generate patches By code transform
Validate and check correctness

Fig. 1: Approach workflow.

A. Fault Localization

The first step of our approach is to identify suspicious code locations that need to be repaired. The approach leverages Zoltar [30], a spectrum-based fault localization toolset. This tool produces a ranked list of code lines that are likely to be a bug. In our approach, depending on the search strategy, we may need to extract the encompassing statements whose code blocks are located in these lines, and the methods where those statements belong to.

B. Search Strategies

Once the faulty method is identified, we unfold several search strategies for discovering other methods in the wild, which may hold fix ingredients for repairing a given buggy program. We consider three levels of similarity allowing for an iterative exploration of the search space.

1) **Signature-similar** methods: We consider that code reuse (e.g., copy/paste) and implementation of basic routines (e.g., string equality) are pervasive in software projects. Thus, it is possible to discover two implementations of the same functionality with slight differences representing corner-cases addressing defects. In this study, we implement a fast (although loose) search by looking up similar method signatures. (S1 in Table II of Section IV.)
2) **Syntactically similar** code: When the first strategy fails, it may become necessary to look at actual code fragments to search for the syntactically similar methods by using a code clone detection technique. (S2 in Table II.)
3) **Semantically similar** code: If the first and second strategy fail, this may imply that the actual fix is largely different from the buggy version with respect to syntactic similarity. Thus, we use a semantic code search approach to discovering other code fragments implementing semantically similar functionalities. (S3 in Table II.)

We now detail how these strategies are implemented. Note that the first strategy relies only on the method signatures, while the other two strategies use each suspicious statement code block as input fragment for the similar code search.

1) Methods with similar signatures: This search strategy is straightforward. Consider the definition below:

Definition 1: **Method Signature (MS)** is defined as a 3-value tuple as below:

$$MS = (rt, mn, Args) \tag{1}$$

where *rt* is the return type of the method named *mn*, and *Args* is a set of parameters (i.e., their types) of *mn*.

When the method is a constructor method, $rt \leftarrow Null$. When the method has no parameter, $Args \leftarrow \emptyset$. When two methods have matching signatures ($rt_1 = rt_2 \wedge mn_1 = mn_2 \wedge Args_1 = Args_2$) then we consider them to have similar method signatures. Given a buggy method, we consider all methods with similar method signatures as candidates for fix ingredients. In practice, we consider that most fixes are simple, and thus we rank the candidate methods by the distance with the buggy method in terms of source lines of code count. We also use the hashing function to dismiss methods whose code is also identical to the buggy method.

2) Syntactically similar code fragments: The second strategy starts from the assumption that although methods may have different signatures, they may still present inner code fragments syntactically similar to others, and whose differences may reflect a fix ingredient to be extracted. In this stragegy, we build on recent studies, which suggest that deep learning embedding can offer good code representations in numeric vectors that are useful for fast similarity computation. To that end, we leverage Convolutional Neural Networks (CNNs), which we used in previous studies [52] to embed method bodies into feature vectors. We then leverage these vectors to identify syntactically similar methods by computing the cosine similarity.

3) Semantically similar code fragments: The third strategy is the most relaxed search scenario where we explore more possibilities beyond syntactic similarity. To find semantically similar code fragments, we build on a recent state-of-the-art code-to-code search engine [50], namely FaCoY. Unlike other semantic code detection approaches, FaCoY is static and can scale to a large-scale search space dataset such as GitHub. In this work, we leverage the package Virtual Machine provided by the authors of FaCoY and input the buggy method statements code blocks for search. The engine yields a ranked list of code fragments, which are semantically similar to the body of buggy methods that have been used as input query.

C. Generate and Validate

Our fixing process unfolds two different scenarios depending on the strategy used to find the similar code. For the first strategy (where method signatures are the same), we perform a naïve transplantation of the body of the searched similar method into that of a given buggy method as a replacement.

For the second and third strategies of code search, the methods, however, may not have the same signatures. We must identify a way to transform the necessary code into the buggy code so that the resulting program may still be compilable. To that end, we expect to find differences relevant to the statements actually highlighted as being involved in the fault: these are statements whose code lines are pointed out by the fault localization tools. We refer to them as *pivot statements*, which must be analyzed and modified with contextual information differences between buggy code and discovered similar methods. In this study, to speed up the search of fix ingredients and reduce the effect of fault localization false positives, we focus on four statements types (namely ExpressionStatement, IfStatement, VariableDeclarationStatement, and ReturnStatement); these statements are reported in the literature as the most recurring buggy statements [53].

a) Pivot IfStatement: when a pivot statement is of such type, we identify the differences between its constituting parts (conditional expression and operator) and attempt to transform a fix ingredient based on heuristics. For example, if the conditional expression is an infix expression (e.g., $a > b$) with the same type as another infix expression in a searched similar method (e.g., $a < b$), we borrow the operator as a fix ingredient and mutate the buggy code to match the other expression.

b) Other Pivot Statements: if the pivot statement is one of the three other types (ExpressionStatement, VariableDeclarationStatement, and ReturnStatement), we first check whether there is an associated IfStatement preceding or encompassing the statement. When an IfStatement is identified as relevant to one of those pivot statements (i.e., a variable in the pivot statement is used in the conditional statement), we treat this statement as a pivot statement. Indeed, the fault localization may not have highlighted such a statement, but given that condition-check bugs are pervasive, the fix may actually be relevant to that part. At this point, we apply the fix ingredient to searching the pivot Ifstatement. Nevertheless, in this case, when the buggy pivot statement has an associated IfStatement but the candidate similar method does not have an IfStatement, we simply delete the IfStatement from our buggy method as a repair action. Similarly, if the pivot statement is matching a statement in the candidate method body and that statement has a related IfStatement while the buggy pivot statement does not have any, we simply insert an IfStatement after mutating its conditional expression with variables used in the buggy pivot statement. Finally, we further check the differences between the buggy pivot statement and the candidate one, and thus mutate the buggy statement with the differences to generate patches. The code of our prototype patch generator (reflecting the heuristics that we implement) as well as the patches that we have generated (cf. Section IV) can be found in our replication package: `https://github.com/AutoProRepair/LSRepair`.

IV. Assessment

We now present experimental results obtained with LSRepair. Given the preliminary nature of this study, we also provide discussions about the limitations and the short-term challenges to be addressed in this research line.

A. Research Question

Our assessment is built for answering a single question: *to what extent real-world bugs can be repaired by* LSRepair?

We expect that a reasonable performance would lead to:

- More investigations, by the APR community, of program repair based on code search, specifically on issues with transplanting fix ingredients.
- Extensive assessment of state-of-the-art approaches to repair, in order to focus on more challenging bugs (i.e., those for which it is difficult to find repair ingredients, or the necessary changes are entangled).

Overall, our experiments set a clear baseline given the straightforward nature of the search and patch generation strategies.

B. Experimental Setup

1) Repair benchmark: In this study, we perform experiments based on the Defects4J [57] benchmark, which includes a manually reviewed set of real-world Java bugs. It has been proposed to enable reproducible studies in the software testing community, but it is recently becoming a de-facto benchmark for repair tools targeting Java programs. Defects4J lists 395 real bugs from six open-source Java projects, and comes with

TABLE I: Comparison of the number of fixed bugs by different APR tools.

	LSRepair	jGenProg [9]	HDRepair [10]	Nopol [28]	ACS [29]	ssFix [54]	ELIXIR [55]	SketchFix [56]	CapGen [13]
# bugs	**19/38**	5/29	13/16	5/35	18/21	20/60	26/41	19/26	21/25

† In each column, the left and right numbers denote the number of correct and plausible patches generated by each APR tool.

test cases for verifying fix *plausibility* as well as the *correct* patches supplied by developers for each bug.

2) Code search space: We leverage the GitHub repository to build the search space, namely a *Method Index* where all methods from java projects are indexed. Although we could list almost 3 million projects tagged with Java as the main language in GitHub, we reduce the noise of toy projects [58] by selecting projects that have been forked at least once by other developers and further dropping out projects where the source code includes non-ascii characters. We also constrain the search space by filtering out *test-related* code files (i.e., having 'test' in their name), setter and getter methods (which are trivial methods), as well as constructor and main methods (which are too specific and may pollute the search space). Note that, for fairness, we also do not consider in our dataset project files (including cloned programs) that are associated to the Defects4J bugs. Overall, we collected 10,449 Java projects in GitHub, of which we indexed 11,043,044 methods.

C. Evaluation of Generated Patches

Given a Defects4J bug, we execute the test suite to localize the buggy method and generate patches by using our code search-based strategy. Then, for each of generated patches, we execute the test suites again. The program is said to be **fully repaired** if the patched program passes all test cases. Table II lists all Defects4J bugs that LSRepair can successfully repair. Most of the bugs are fixed by the first two code search strategies. Nevertheless, we have cases where syntactic similarity and semantic similarity have provided relevant fix ingredients that were possible to exploit.

Nevertheless, in light with recent studies [59], we note that passing all test cases only indicates that the generated patch is a **plausible** fix. To check for correctness, we manually compare the suggested plausible patch against the actual developer patch in the Defects4J benchmark.

D. Quantitative Comparison with State-of-the-art APR Tools

Table I reports on the current achievements of APR tools on the Defects4J dataset and comparison with the results of LSRepair. For each column, a pair of numbers are reported, where the left and right are the number of correct and plausible patches, respectively. We note that our approach provides comparable performance with the state-of-the-art tools. Finally, we have identified 10 bugs, which are fixed by our approach and are not yet fixed by state-of-the-art APR tools. All details about the results are available in our replication package.

E. Runtime Performance

We note that the repair of certain bugs can be fast when the search space (i.e., # similar methods) is small. Table II reports the time elapsed to fix each bug, which are from a few seconds to dozens of minutes.

TABLE II: The bugs correctly repaired by LSRepair.

Bug ID	S1	S2	S3	Time	Search Space
Chart-1			✓	-	10
Chart-4	✓			3m12s	30
Chart-11	✓			37s	2
Lang-21	✓			12s	3
Lang-24	✓			19m17s	142
Lang-29		✓		-	10
Lang-46	✓			1m59s	16
Lang-48	✓			30s	9
Lang-51	✓			58s	30
Lang-52	✓			15s	5
Lang-54	✓			19s	25
Math-63	✓			8m3s	27
Math-70	✓			37s	40
Math-75	✓			15s	20
Math-79	✓			17s	549
Math-89			✓	-	10
Math-91	✓			13s	30
Math-94	✓			1m43s	182
Mockito-13	✓			43m32s	6

h: hour, m: minute, s: second. Time for S2 and S3 is not provided (to avoid bias) since the two strategies need to preprocess suspicious methods before code search.

V. Discussions

Code Transformation: The main limitation of our approach comes from the difficulty to explore and exploit automatically the fix ingredients available in the searched similar methods. This is known as the code transformation problem [20]. Our prototype uses straightforward heuristics to transform the ingredients. In previous work, SearchRepair [20] achieved good performance on the IntroClass datasets of simple C programs by using a simple textual replacement for renaming variables. We explored a similar strategy. Nevertheless, Defects4J's real-world bugs are known [60] to require more advanced code transform [61]–[63] techniques, which we plan to explore in future work.

On the Similar Code Search Problem: To improve the efficiency in patch generation, it is necessary to implement a re-ranking scheme of similar code. In this study, we used naïve heuristics to prioritize code, based on the similarity (e.g., syntactic and semantic) with a given buggy code. However, in the future work, we need a smarter approach to prioritizing simialr code based on the likelihood of fixing a given bug, instead of similarity. Such an approach can build on recurrent change patterns for fixing bugs [53].

Threats to Validity: The main threat to the validity of our study is the method-level granularity. Bugs located in a class or field declaration [53], it cannot be addressed by our tool.

VI. Conclusion

In this paper, we investigated the potential of using code search strategies on-the-fly to find fix ingredients for automated program repair. Our prototype implementation showed that LSRepair can correctly fix 19 bugs from Defects4J, and 10 of them are not yet fixed by other APR tools. Although such number might appear small, it should be reflected against the current status in the field of Java APR, where state-of-the-art APR tools can only fixed a few more (1 to 7) bugs than our tool. We thus propose this approach and the associated results as a baseline for the future research direction on APR techniques based on code search.

Acknowledgements. This work is supported by the Fonds National de la Recherche (FNR), Luxembourg, through RECOMMEND 15/IS/10449467 and FIXPATTERN C15/IS/9964569.

References

[1] X.-B. D. Le, "Towards efficient and effective automatic program repair," in *ASE*, 2016.

[2] M. Monperrus, "Automatic software repair: a bibliography," *ACM Computing Surveys (CSUR)*, 2018.

[3] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *ICSE*, 2009.

[4] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A generic method for automatic software repair," *TSE*, 2012.

[5] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *ICSE*, 2012.

[6] T. F. Bissyandé, "Harvesting fix hints in the history of bugs," *arXiv preprint arXiv:1507.05742*, 2015.

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

[8] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The Strength of Random Search on Automated Program Repair," in *ICSE*, 2014.

[9] M. Martinez and M. Monperrus, "Astor: A program repair library for java," in *ISSTA*, 2016.

[10] X. B. D. Le, D. Lo, and C. L. Goues, "History Driven Program Repair," in *SANER*, 2016.

[11] F. Long and M. Rinard, "Staged program repair with condition synthesis," in *ICSE*, 2015.

[12] ——, "Automatic patch generation by learning correct code," *ACM SIGPLAN Notices*, 2016.

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

[14] A. Koyuncu, T. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. Le Traon, "Impact of Tool Support in Patch Construction," in *ISSTA*, 2017.

[15] W. Weimer, Z. P. Fry, and S. Forrest, "Leveraging program equivalence for adaptive program repair: Models and first results," in *ASE*, 2013.

[16] A. Carzaniga, A. Gorla, A. Mattavelli, N. Perino, and M. Pezze, "Automatic recovery from runtime failures," in *ICSE*, 2013.

[17] Z. Coker and M. Hafiz, "Program transformations to fix c integers," in *ICSE*, 2013.

[18] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: program repair via semantic analysis," in *ICSE*, 2013.

[19] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro, "The plastic surgery hypothesis," in *FSE*, 2014.

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

[21] S. Mechtaev, J. Yi, and A. Roychoudhury, "Directfix: Looking for simple program repairs," in *ICSE*, 2015.

[22] M. Martinez and M. Monperrus, "Mining repair models for reasoning on the search space of automated program fixing," *EMSE*, 2013.

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

[24] X.-B. D. Le, Q. L. Le, D. Lo, and C. Le Goues, "Enhancing automated program repair with deductive verification," in *ICSME*, 2016.

[25] L. Chen, Y. Pei, and C. A. Furia, "Contract-based program repair without the contracts," in *ASE*, 2017.

[26] 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 *FSE*, 2017.

[27] F. Long, P. Amidon, and M. Rinard, "Automatic inference of code transforms for patch generation," in *FSE*, 2017.

[28] J. Xuan, M. Martinez, F. Demarco, M. Clement, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," *TSE*, 2017.

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

[30] T. Janssen, R. Abreu, and A. J. van Gemund, "Zoltar: A toolset for automatic fault localization," in *ASE 2009*.

[31] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus, "Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset," 2016.

[32] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang, "Identifying patch correctness in test-based program repair," in *ICSE*, 2018.

[33] 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 *ICSE-NIER*, 2014.

[34] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. L. Traon, "Fixminer: Mining relevant fix patterns for automated program repair," *arXiv preprint*, 2018.

[35] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *ASE*, 2014.

[36] B. Fluri, M. Wuersch, M. PInzger, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *TSE*, 2007.

[37] M. Martinez, L. Duchien, and M. Monperrus, "Automatically extracting instances of code change patterns with ast analysis," in *ICSME*, 2013.

[38] A. Carzaniga, A. Gorla, N. Perino, and M. Pezzè, "Automatic workarounds for web applications," in *FSE*, 2010.

[39] M. Gabel and Z. Su, "A study of the uniqueness of source code," in *FSE*, 2010.

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

[41] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: a tool for finding copy-paste and related bugs in operating system code," in *OSDI*, 2004.

[42] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *ICSE*, 2007.

[43] J. Krinke, "Identifying similar code with program dependence graphs," in *RE*, 2001.

[44] C. Liu, C. Chen, J. Han, and P. S. Yu, "Gplag: detection of software plagiarism by program dependence graph analysis," in *KDD*, 2006.

[45] M. Kim and D. Notkin, "Discovering and representing systematic code changes," in *ICSE*, 2009.

[46] S. Kim, K. Pan, and E. Whitehead Jr, "Memories of bug fixes," in *FSE*, 2006.

[47] T. Molderez, R. Stevens, and C. De Roover, "Mining change histories for unknown systematic edits," in *MSR*, 2017.

[48] R. Yue, N. Meng, and Q. Wang, "A characterization study of repeated bug fixes," in *ICSME*, 2017.

[49] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller, "Documenting and automating collateral evolutions in linux device drivers," in *OSR*, 2008.

[50] K. Kim, D. Kim, T. F. Bissyandé, E. Choi, L. Li, J. Klein, and Y. Le Traon, "FaCoY–a code-to-code search engine," in *ICSE*, 2018.

[51] S. Sidiroglou, E. Lahtinen, N. Rittenhouse, P. Piselli, F. Long, D. Kim, and M. Rinard, "Targeted automatic integer overflow discovery using goal-directed conditional branch enforcement," in *ACM Notices*, 2015.

[52] K. Liu, D. Kim, T. F. Bissyandé, S. Yoo, and Y. L. Traon, "Mining fix patterns for findbugs violations," *IEEE TSE*, 2019 (to appear).

[53] K. Liu, D. Kim, L. Li, K. Anil, T. F. Bissyandé, and Y. L. Traon, "A closer look at real-world patches," in *ICSME*, 2018.

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

[55] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "Elixir: Effective object-oriented program repair," in *ASE*, 2017.

[56] J. Hua, M. Zhang, K. Wang, and S. Khurshid, "Towards practical program repair with on-demand candidate generation," in *ICSE*, 2018.

[57] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java," in *ISSTA*, 2014.

[58] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The Promises and Perils of Mining GitHub," in *MSR*, 2014.

[59] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, "Is the cure worse than the disease? overfitting in automated program repair," in *FSE*, 2015.

[60] V. Sobreira, T. Durieux, F. Madeiral, M. Monperrus, and M. de Almeida Maia, "Dissection of a Bug Dataset: Anatomy of 395 Patches from Defects4J," in *SANER*, 2018.

[61] E. T. Barr, M. Harman, Y. Jia, A. Marginean, and J. Petke, "Automated software transplantation," in *ISSTA*, 2015.

[62] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard, "Automatic error elimination by horizontal code transfer across multiple applications," in *ACM SIGPLAN Notices*, 2015.

[63] S. Sidiroglou-Douskos, E. Davis, and M. Rinard, "Horizontal code transfer via program fracture and recombination," 2015.