Potential Solutions to Challenges in C Program Repair: A Practical Perspective

Jifeng Xuan
School of Computer Science
Wuhan University
Wuhan, China
jxuan@whu.edu.cn

Qi Xin
School of Computer Science
Wuhan University
Wuhan, China
qxin@whu.edu.cn

Liqian Chen
College of Computer
National University of Defense Technology
Changsha, China
lqchen@nudt.edu.cn

Xiaoguang Mao
College of Computer
National University of Defense Technology
Changsha, China
xgmao@nudt.edu.cn

2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 979-8-3503-2996-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/ASE56229.2023.00211

***Abstract*—Automated program repair is to reduce the manual work for bug fixing by human developers. In recent 15 years, the research community of program repair has created many novel techniques. However, these techniques share several assumptions that cannot always be satisfied in daily software development. This badly hurts the application of program repair in practice. For example, many repair techniques assume that test cases are well written before patch generation; many techniques assume that specific language features can be ignored (or already-processed). In this paper, we propose a framework of C program repair, which mainly addresses two challenges: test-independent repair and preprocessor directive processing. Our solution to test-independent repair is to automatically construct patch conditions for C programs via parsing the syntax structures; our solution to preprocessor directive processing is to generate code symbols to replace preprocessor directives. We plan to implement these potential solutions with program analysis techniques. The goal of this paper is to present practical solutions for developers to automate C program repair.**

***Index Terms*—C program repair, test-independent repair, preprocessor directives, industrial solutions, program analysis**

I. Introduction

Human developers fix program bugs with programming skills and experience; automated program repair aims at fixing bugs via automatic techniques. The original goal of automated program repair is to replace the repair efforts made by human developers [1], [2]. In industry, it is not easy to make automated program repair practical. One of the major reasons is that automated program repair lacks program specifications that can be mapping to requirements for developers.

In recent 15 years, the research community of program repair is evolving. Since the pioneering method GenProg by Weimer et al. [3], test cases have severed as the practical specifications in program repair. The benefit of applying test cases in program repair is direct. First, a test case consists of two major parts: a test input and a test oracle. The test oracle is naturally designed to ensure the correctness of a test case. Second, the emerging tendency of test-driven development provides rich test cases for program repair. Applying test cases in program repair is also called test-based program repair [4]. However, the ideal scenario of test-based program repair may fail in practice. For human developers, designing a large number of test cases is time-consuming; for automated program repair, the current number of test cases cannot provide and guard the correctness of automated generated patches [5], [6]. Many bugs are never covered by manually-written test cases by developers [7].

The research community has created many novel approaches to program repair. Most of these approaches are designed for fixing general bugs and evaluated on a specific programming language: detailed language features of the syntax and semantics may be ignored in automatic approaches [8]. Recent progress of deep learning and large language models even relaxes the requirements of processing programming languages: source code in programming languages is transformed into text in natural languages [9]. However, for accurate program repair in industry, the ignored language features cannot be bypassed, such as preprocessor directives in C programs or bytecode structures in Java programs.

In this paper, we propose a framework of C program repair, which mainly addresses two challenges: test-independent repair and preprocessor directive processing. The goal of this repair framework is to provide practical solutions to fix bugs in C programs. The application scenario of the proposed framework is the same as daily software development by developers: the framework takes a buggy C program as input and returns a patched program as output. This framework consists of five iterative steps: code preprocessing, condition construction, patch generation, patch update, and repair assessment. Fig. 1 shows the overview of our framework for C program repair. Among these five steps, the three steps of patch generation, patch update, and repair assessment share the same ideas

This work is supported by the National Natural Science Foundation of China under Grant Nos. 62141221 and 62202344.

2643-1572/23/$31.00 ©2023 IEEE
DOI 10.1109/ASE56229.2023.00211

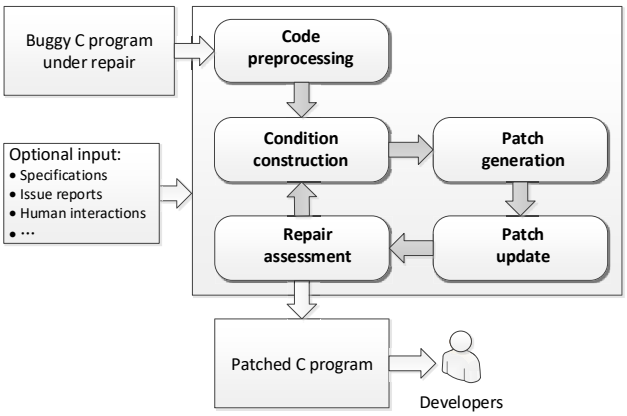

Fig. 1. Overview of the proposed framework for C program repair.

of test-based program repair; the other two steps of code preprocessing and condition construction are designed to meet the requirements and challenges in C program repair.

In our framework, we focus on solutions to two challenges in C program repair. One main challenge is how to generate patches without pre-defined test cases, called *test-independent* repair. Patch generation in test-independent repair is a challenging problem since there is no test case for assessing correctness of patches. Our major solution to test-independent repair is to automatically construct patch conditions for C programs via parsing the syntax structures. We use program analysis to implement the construction of patch conditions. A patch condition is identified via static or dynamic program analysis via parsing its related syntax in the program language.[1] For example, a pointer in C programs should be initialized before the pointer is used. Such initialization of a pointer can be viewed as a case of patch conditions. In our proposed framework, the solution to test-independent repair is designed in the steps of condition construction, patch generation, and repair assessment.

The other main challenge is how to process preprocessor directives in program repair. A preprocessor directive is a language feature in C programs, which is defined outside the C syntax. In a compiler of C programs, preprocessor directives are replaced with specific grammars before compilation. For example, a preprocessor directive `#ifdef LIB_H` is to check whether a symbol `LIB_H` is defined when the line that contains this preprocessor directive is processed. Our major solution to preprocessor directive processing is to generate code symbols (also called *tokens* in compilers) to replace preprocessor directives. We use program analysis to implement the generation code symbols. The dependency among preprocessor directives are extracted via static analysis. In our proposed framework, the solution to preprocessor directive processing is designed in the steps of code preprocessing and patch update.

This paper makes the following major contributions:

- The goal of this paper is to address the two listed challenges in C program repair. This paper provides practical solutions for developers to automate patch generation for C programs.
- We present a framework of C program repair with five main steps. This framework supports potential solutions to the challenges of test-independent repair and preprocessor directive processing.

The rest of this paper is organized as follows. Section II presents the potential solutions to challenges in C program repair and our proposed framework. Section III discusses the techniques in our solutions. Section IV concludes.

II. Potential Solutions to Challenges in C Program Repair

This section presents our potential solutions to the challenges in C program repair. We show a framework and two solutions to the challenges.

A. Overview

In the community of program repair, most of existing repair methods follow an idea that first localizes bugs and then fixes bugs [10], [11]. The framework in this paper shares the same idea and adapts to fix C program bugs. As mentioned in Section I, Fig. 1 shows an overview of our framework for C program repair. To make program repair practical, our framework assumes that the input is a buggy C program and the output is its patched version. This assumption shares the same input and output as manually bug fixing. Optional inputs include pre-defined specifications by developers, issue reports from bug tracking systems, and human interactions with developers. The optional inputs are unnecessary, but can improve the correctness of patch generation.

As shown in Fig. 1, the proposed framework consists of five main steps: code preprocessing, condition construction, patch generation, patch update, and repair assessment. Among these steps, *code processing* severs as a pre-step of the other four steps and generates code symbols that replace the preprocessor directives. The other four steps construct an iterative process that keeps updating patches until the buggy program is assessed as a fixed version.

In the four iterative steps, *condition construction* is designed to extract patch conditions that can guide the repair behaviors that lead to a correct patch. A *patch condition* in our work is defined as a constraint that describes potential behaviors. For example, if a pointer `ptr` is invoked in the source code, a patch condition can be used to restrict the initialization of the pointer, i.e., `ptr != null`. The patch conditions are designed to meet the lack of test cases in test-independent repair. The step of *patch generation* is to generate patches via automatic techniques, like evolutionary computation in GenProg [3] or symbolic execution in SemFix [8]. The step of *patch update* is to statically rewrite the source code via

[1] As a solution paper, we choose program analysis as the priority in the implementation. Other techniques like program synthesis or abstract interpretation can also be workable.

merging the patch to the buggy version. The general idea of patch generation and patch update in our work is similar to that in existing methods [3], [4]. The step of *repair assessment* provides a metric that assesses the quality of generated patches. Such metrics can be defined as a fitness function that qualifies the correctness of patches. The process of these four steps iterates until a patch is acceptable by the step of repair assessment.

The basic idea of our proposed framework is to address two challenges in C program repair, including test-independent repair and preprocessor directive processing. Among these five steps in Fig. 1, the two steps of code preprocessing and condition construction are specialized to meet the challenges in C program repair. Condition construction is to restrict the patch behaviors in the scenario of test-independent repair while code preprocessing is to replace preprocessor directives. Solutions to the challenge of test-independent repair are mainly designed in the steps of condition construction, patch generation, and repair assessment; solutions to the challenge of preprocessor directive processing are mainly designed in the steps of code preprocessing and patch update.

Existing methods in patch generation can be directly used in the framework in Fig. 1. For example, the typical method GenProg [3] assesses patches via defining a fitness function based on the evaluation of test execution, i.e., how many test cases are passed. Such a fitness function can be transformed into a new function of evaluating how many patch conditions are passed.

B. Solutions to Test-Independent Repair

The scenario of test-independent repair does not assume that pre-defined test cases can cover all buggy behaviors in the program under repair. This is different from the scenario of existing work in test-based program repair. In test-based program repair, test cases play an important role of isolating incorrect patches although existing studies show that test cases can result in plausible patches [12] and patch overfitting [5] in program repair. If a patched program fails in executing a test case, the patch can be labeled as an incorrect one; if a patched program passes all test cases, the patch may be correct or incorrect. That is, the failing of test execution can be directly used to detect incorrect patches, but the passing of test execution may not.

Test cases cannot cover all positions of patches [13]. This is common in large C programs. In existing research of test-based program repair, the lack of test cases relaxes the boundary of patch generation. Then the output of program repair tends to be the output of program synthesis [14]. That is, a large number of patches can be generated and it is difficult to isolate incorrect patches from all these generated patches. The emerging application of large language models aggravates this difficulty [9].

Our solution to the challenge of test-independent repair is to extract constraints from source code as patch conditions. Such constraints can cover the program safety, but cannot cover the functional behaviors. For example in C programs,

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void func(int bound, int index)
{
    char *ptr = (char *) malloc (bound * sizeof(char));
    strcpy(ptr, "demo");  // No potential patch condition for ptr
    printf("%c", ptr[index]);  // Two potential patch conditions for
      ptr and index
    [...]
}
```

Fig. 2. Snippet of a C program with a pointer `ptr` and an array index `index`.

a pointer must be initialized before its invocation; an index of arrays must be not out of the array boundary; a dynamic memory allocation must be released after its usage. The above examples of C code can be converted into constraints for patch generation, such as a pointer `ptr != null` for pointer invocation, an array index `index >= 0 && index < arrayBoundary` for array usage, and a pair of `malloc(); free();` for memory allocation. Developers can define such constraints for different scenarios, i.e., patch conditions in our work.

Patch conditions can be automatically constructed. For each constraint, human developers can define a rule for constraint extraction. We plan to use program analysis to automate the process of constraint extraction. Program analysis can parse source code into Abstract Syntax Trees (ASTs) and then traverse the ASTs [15], [16]. Code symbols like pointers or APIs of memory usage can be extracted via program analysis. Program analysis consists of two general categories: static analysis and dynamic analysis [17]. Static analysis parses the whole program structures without running programs while dynamic analysis monitors and parses the executed program paths with running programs. With the support of dynamic analysis, concrete values of variables during program execution can be monitored and collected. For a buggy program, there are more than one patch conditions in the source code. Then a general solution is to define a metric to rank potential positions for patch generation and then generate patches for each potential position.

Fig. 2 uses a code snippet to demonstrate the patch conditions in a C program. For the scenario of test-independent repair, no test case is used to supervise the patch generation. Among the three lines in the code snippet of `func()`, both Line 2 and Line 3 contains the pointer `ptr`: at Line 2, `ptr` receives an assignment of a string; at Line 3, `ptr` is invoked to access its value. Then at Line 2, no potential patch condition exists for `ptr`; at Line 3, a patch condition `ptr != null` for pointer invocation should be satisfied. Line 3 also accesses the value of `ptr[index]`. Then, another patch condition `index >= 0 && index < bound` should be satisfied. Thus, as shown in Fig. 2, there are two potential patch conditions at Line 3 of `func()` and no patch condition

at Line 2. In the step of patch generation, any patch that violates the two patch conditions at Line 3 can be discarded.

In the proposed framework in Fig. 1, the step of condition construction is to designed to extract patch conditions. We provide a method of implementation for condition construction. First, for a given buggy program, we apply static analysis to collect all positions that potential patch conditions exist. Second, for each patch condition, we use program instrumentation techniques to update the source code via inserting monitors of variable values. A simple way of implementing monitors is to add logging code for a patch condition. Third, for the buggy code with instrumentation, executing the code can collect the runtime values for constructing patch conditions.

In the step of patch generation, all positions (e.g., all statements) in the buggy programs can be ranked according a pre-defined metric. For example, a pre-defined metric could be the number of patch conditions in each position or the depth in a dependency graph in the ASTs. Then automatic patch generation can be used to generate patches for each position. Existing techniques of patch generation can be directly used, such as evolutionary computation in GenProg [3], constraint solving in Nopol [11], code reuse in sharpFix [18], and code transplantation in TransplantFix [19]. The output of the step of patch generation is one or more patches that can be used to update the buggy program.

In the step of patch assessment, runtime values of patch conditions are collected and evaluated. An ideal patch should be a patch that violates none of patch conditions. A simple way of designing a quantitative metric is to convert the number of violated patch conditions into a numeric value. This quantitative metric can be used to rank patches: a patch with a low value of violated patch condition is prioritized.

In the above paragraphs of in this section, we present potential solutions to the challenge of test-independent repair. The basic assumption is no test case that covers the buggy position. However, it is possible to add other inputs to improve the quality of patch generation. Such optional inputs may contain pre-defined specifications, issue reports, warnings from static parsers [20], etc. These optional inputs can be transformed into constraints, which sever as the supplementation to patch conditions for test-independent repair.

C. Solutions to Preprocessor Directive Processing

Preprocessor directives are a type of special language feature in the C language. The preprocessor directives are not contained in C grammars, but widely exist in most of C programs. If a C program with preprocessor directives is sent to a compiler, preprocessor directives are replaced with code symbols (also called tokens) before compilation. Thus, the preprocessor directives are not visible to C compilers. A preprocessor directive can be used to control the C program behaviors, including the setup of values (e.g., `#define`) or the control of compilation (e.g., `#ifdef-#else-#endif`). Since preprocessor directives are not defined in C grammars, existing methods of program repair directly ignore the processing of preprocessor directives. Fig. 3 presents a code snippet

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
#ifdef PI
    printf("PI is %f\n", PI);
#else
    #define PI 3.14159
    printf("PI is %f\n", PI);
#endif
    [...]
    return 0;
}
```

Fig. 3. Snippet of a C program with preprocessor directives.

with preprocessor directives that can control the numerical precision of the circular constant inside the C code. The preprocessor directives can be replaced with C code before the source code is compiled.

Our solution to preprocessor directive processing is to generate code symbols by replacing preprocessor directives. This shares the same idea that a C compiler processes the preprocessor directives: preprocessor directives can be handled before the program is compiled. This indicates that all the dependencies of preprocessor directives can be extracted. We use static analysis to extract the constraints among preprocessor directives and then parse the ASTs to detect all values preprocessor directives (a value in a preprocessor directive can be a constant or a string). Based on the static analysis, code symbols can be identified from the ASTs. There is no existing tool for preprocessor directives in C programs. The implementation of such a tool relies on the context-free grammar in C programs.[2]

In our proposed framework, the step of code preprocessing is specialized for the preprocessor directives reprocessing. The basic idea is to implement static analysis for preprocessor directives and to replace these preprocessor directives with code symbols. The step of code reprocessing can be viewed as a first step in a tool of C program repair. As shown in Fig. 1, the step of code reprocessing does not need to be involved in the loop of patch generation and patch update.

To make a user-friendly patch, the step of patch update has to reversely converted code symbols in a patched program into preprocessor directives. For the reversed conversion from code symbols to preprocessor directives, the trace of replacement in the step of code reprocessing should be recorded. This is different from the general design of patch update in test-based program repair.

III. Discussions

Existing research work has rarely covered test-independent repair and preprocessor directives reprocessing in C program

[2]The context-free grammar of preprocessor directives is used in the implementation of C compilers. This grammar can be used in the implementation of the static analysis of replacing preprocessor directives.

repair. We discuss empirical evaluation and extension of the proposed framework in this section.

A. Empirical Evaluation

Automated program repair is expected to replace the manually bug fixing by developers. To date, fully replacing manual work by automated program repair is far from being carried out. A straightforward way of evaluating program repair is the empirical evaluation on real-world buggy programs.

Existing benchmarking datasets such as CoreBench [21], C-Pack of IPAs [22], BugsCpp [23] pay great efforts in data collection and reproduction. However, these benchmarking datasets cannot be directly used to evaluate test-independent repair and preprocessor directive processing due to the assumption of test availability. To conduct a dataset that can be used to evaluate the challenges of C program repair, one way is to revise existing benchmarking datasets via removing functional bugs and test cases; another way is to extract new buggy C programs that relate to safety or security bugs, like instances of Common Vulnerabilities & Exposures (CVEs) [24].

B. Extension of the Framework

Our solutions in Section II are automatic techniques of program analysis. This is a bit different from the majority of the research community of program repair, like repair methods based on machine learning or deep learning.

The scenario of test-independent repair does not rely on test cases while the scenario of preprocessor directives allows the replacement of code symbols. This indicates that we lose several toolkits for the correctness evaluation of patch generation or the isolation of incorrect patches. In our work, we choose program analysis to implement the proposed solutions since the result of program analysis can be inferred and is explainable. It is possible to extend the propose framework with intelligent methods based on machine learning, deep learning, or large language models. For example, the code symbols that replacing the preprocessor directives can be predicted with learning-based repair methods.

IV. Conclusions

This paper presents potential solutions to two challenges in C program repair, including test-independent repair and preprocessor directive processing. The solution to test-independent repair is to automatically construct patch conditions for C programs via parsing the syntax structures; the solution to preprocessor directive processing is to generate code symbols to replace preprocessor directives. Both above solutions are considered to be implemented with program analysis techniques.

To implement the idea of C program repair, we propose a framework that embeds our solutions. This framework consists of five iterative steps, called code preprocessing, condition construction, patch generation, patch update, and repair assessment. The two steps of condition construction and code preprocessing are designed for solutions to test-independent repair and preprocessor directive processing, respectively. The other three steps, i.e., patch generation, patch update, and repair assessment, share similar ideas of existing methods of program repair. For the step of patch generation, existing techniques like evolutionary computation and symbolic execution can be directly applied to generate C patches. This reduces the difficulties of creating new methods of patch generation in tackling the challenges in C program repair.

References

[1] B. Demsky, M. D. Ernst, P. J. Guo, S. McCamant, J. H. Perkins, and M. C. Rinard, "Inference and enforcement of data structure consistency specifications," in *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006*, L. L. Pollock and M. Pezzè, Eds. ACM, 2006, pp. 233–244. [Online]. Available: https://doi.org/10.1145/1146238.1146266

[2] M. C. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee, "Enhancing server availability and security through failure-oblivious computing," in *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, E. A. Brewer and P. Chen, Eds. USENIX Association, 2004, pp. 303–316. [Online]. Available: http://www.usenix.org/events/osdi04/tech/rinard.html

[3] W. Weimer, T. Nguyen, C. L. Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, 2009, pp. 364–374.

[4] 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," *Empir. Softw. Eng.*, vol. 22, no. 4, pp. 1936–1964, 2017. [Online]. Available: https://doi.org/10.1007/s10664-016-9470-4

[5] E. K. Smith, E. Barr, C. Le Goues, and Y. Brun, "Is the cure worse than the disease? overfitting in automated program repair," in *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Bergamo, Italy, September 2015.

[6] H. Tian, K. Liu, Y. Li, A. K. Kaboré, A. Koyuncu, A. Habib, L. Li, J. Wen, J. Klein, and T. F. Bissyandé, "The best of both worlds: Combining learned embeddings with engineered features for accurate prediction of correct patches," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 4, pp. 92:1–92:34, 2023. [Online]. Available: https://doi.org/10.1145/3576039

[7] G. Fraser and A. Arcuri, "1600 faults in 100 projects: automatically finding faults while achieving high coverage with evosuite," *Empirical Software Engineering*, vol. 20, no. 3, pp. 611–639, 2015. [Online]. Available: https://doi.org/10.1007/s10664-013-9288-2

[8] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *Proceedings of International Conference on Software Engineering*, 2013, pp. 772–781.

[9] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," *CoRR*, vol. abs/2107.03374, 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[10] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *Software Engineering, IEEE Transactions on*, vol. 38, no. 1, pp. 54–72, 2012.

[11] J. Xuan, M. Martinez, F. Demarco, M. Clement, S. R. L. Marcote, T. Durieux, D. L. Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," *IEEE Trans. Software Eng.*, vol. 43, no. 1, pp. 34–55, 2017. [Online]. Available: https://doi.org/10.1109/TSE.2016.2560811

[12] Z. Qi, F. Long, S. Achour, and M. C. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, M. Young and T. Xie, Eds. ACM, 2015, pp. 24–36. [Online]. Available: https://doi.org/10.1145/2771783.2771791

[13] P. Ma, H. Cheng, J. Zhang, and J. Xuan, "Can this fault be detected: A study on fault detection via automated test generation," *J. Syst. Softw.*, vol. 170, p. 110769, 2020. [Online]. Available: https://doi.org/10.1016/j.jss.2020.110769

[14] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, J. Kramer, J. Bishop, P. T. Devanbu, and S. Uchitel, Eds. ACM, 2010, pp. 215–224. [Online]. Available: https://doi.org/10.1145/1806799.1806833

[15] C. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, V. Sarkar and M. W. Hall, Eds. ACM, 2005, pp. 190–200. [Online]. Available: https://doi.org/10.1145/1065010.1065034

[16] C. Lattner and V. S. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, 2004, pp. 75–88. [Online]. Available: https://doi.org/10.1109/CGO.2004.1281665

[17] D. A. Wagner and D. Dean, "Intrusion detection via static analysis," in *2001 IEEE Symposium on Security and Privacy, Oakland, California, USA May 14-16, 2001*. IEEE Computer Society, 2001, pp. 156–168. [Online]. Available: https://doi.org/10.1109/SECPRI.2001.924296

[18] Q. Xin and S. P. Reiss, "Better code search and reuse for better program repair," in *Proceedings of the 6th International Workshop on Genetic Improvement, GI@ICSE 2019, Montreal, Quebec, Canada, May 28, 2019*, J. Petke, S. H. Tan, W. B. Langdon, and W. Weimer, Eds. ACM, 2019, pp. 10–17. [Online]. Available: https://doi.org/10.1109/GI.2019.00012

[19] D. Yang, X. Mao, L. Chen, X. Xu, Y. Lei, D. Lo, and J. He, "Transplantfix: Graph differencing-based code transplantation for automated program repair," in *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 2022, pp. 107:1–107:13. [Online]. Available: https://doi.org/10.1145/3551349.3556893

[20] K. Liu, D. Kim, T. F. Bissyandé, S. Yoo, and Y. L. Traon, "Mining fix patterns for findbugs violations," *IEEE Trans. Software Eng.*, vol. 47, no. 1, pp. 165–188, 2021. [Online]. Available: https://doi.org/10.1109/TSE.2018.2884955

[21] M. Böhme and A. Roychoudhury, "Corebench: Studying complexity of regression errors," in *Proceedings of the 23rd ACM/SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2014, 2014, pp. 105–115.

[22] P. Orvalho, M. Janota, and V. M. Manquinho, "C-pack of ipas: A C90 program benchmark of introductory programming assignments," *CoRR*, vol. abs/2206.08768, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2206.08768

[23] "The bugscpp dataset," https://github.com/Suresoft-GLaDOS/bugscpp/, accessed: 2023-07-20.

[24] "The cve list," https://cve.mitre.org/, accessed: 2023-07-20.