PraPR: Practical Program Repair *via* Bytecode Mutation

Ali Ghanbari Lingming Zhang
University of Texas at Dallas, TX 75080, USA
{ali.ghanbari,lingming.zhang}@utdallas.edu

***Abstract*—Automated program repair (APR) is one of the recent advances in automated software engineering aiming for reducing the burden of debugging by suggesting high-quality patches that either directly fix the bugs, or help the programmers in the course of manual debugging. We believe scalability, applicability, and accurate patch validation are the main design objectives for a practical APR technique. In this paper, we present PraPR, our implementation of a practical APR technique that operates at the level of JVM bytecode. We discuss design decisions made in the development of PraPR, and argue that the technique is a viable baseline toward attaining aforementioned objectives. Our experimental results show that: (1) PraPR can fix more bugs than state-of-the-art APR techniques and can be over 10X faster, (2) state-of-the-art APR techniques suffer from dataset overfitting, while the simplistic template-based PraPR performs more consistently on different datasets, and (3) PraPR can fix bugs for other JVM languages, such as Kotlin.**
PraPR is publicly available at https://github.com/prapr/prapr.

***Index Terms*—Program Repair, JVM Bytecode, Mutation Testing**

I. Introduction

Debugging is a notoriously time consuming activity that is responsible for more than 50% of the development time/effort [1]. So far, a large body of research has been dedicated to automatically localize [2] or fix software bugs [3]. Automated Program Repair (APR) is intended to directly fix software bugs with minimal human intervention which has been under intense research despite being a young research area [3].

Based on the actions taken for fixing a bug, state-of-the-art APR techniques can be divided into two main classes: (1) techniques that monitor dynamic behavior of a program to find deviations from certain specifications, and *heal* the program by modifying its runtime state in case of any abnormal behavior [4]; (2) techniques that modify static representation of programs based on various rules/techniques, and use either tests or formal specifications as the oracle to validate each generated candidate patch for finding *plausible* patches (i.e., patches that can pass all the tests/checks). Plausible patches are further checked to identify *correct*, or *genuine*, patches (i.e., patches equivalent to developer patches) [5]–[18].

Scalability, applicability, and accurate patch validation are often cited as the main design objectives for building practical APR techniques [19]. Scalability refers to the ability of an APR technique in handling large, real-world programs. Applicability is the ability of the technique in handling different programming idioms, languages, or even different programming paradigms. Finally, patch validation refers to the process of classifying the patches generated by the APR tool into genuine and plausible patches. We introduce a practical, general-purpose APR technique, named PraPR (**Pra**ctical **P**rogram **R**epair) [20], that is operating at the level of JVM bytecode [21]. By discussing the design decisions made in the development of PraPR, we argue that the technique is a viable baseline toward achieving the aforementioned objectives.

II. Overall Approach

PraPR is based on three classes of mutators that can be seen as a spectrum of mutators ranging from traditional mutators (e.g., changing `a>=b` into `a>b`) to more complex program transformation operators that occur frequently in real-world bug-fix commits. In particular, we have adopted a set of 18 mutators from traditional mutation testing [22] which is augmented with a set of 12 replacement mutators (e.g., replacing field accesses or method invocations) and 14 mutators that are responsible for inserting checks in the vicinity of field dereferences and method calls. Table I illustrates two examples from each class of mutators wherein the white part lists examples from traditional mutators, light-gray block contains examples from augmented mutators that replace a field name or a method name with another, and the dark-gray part shows examples from augmented mutators that insert nullity checks before dereferences or virtual method calls and use default values [21] instead of triggering NullPointerException.

TABLE I: Mutators examples

ID	Mutator Illustration
AP	`y=o.m(x)`↪`y=x`
RV	`return x`↪`return x+1`
FR	`int x=o.f1`↪`int x=o.f2`
MR	`int y=o.m1(x)`↪`int y=o.m2(x)`
FG	`int x=o.f`↪`int x=(o==null?0:o.f)`
MG	`int y=o.m(x)`↪`int y=(o==null?0:o.m(x))`

We stress that our mutators are designed without any bias towards our dataset of buggy programs: the mutators are either from traditional mutation testing or have already been widely used in existing APR techniques [5]–[7], [23]. To confirm the generality of our mutators, we built a fix-pattern extraction program (with 4K LoC Java code) based on the GumTree AST diffing framework [24] to mine real-world bug-fix commits from HD-Repair dataset [7], and have confirmed that our mutators also appear in real bug fixes. Please note that although these mutators are intended to make small changes to the programs, implementing most of the mutators is non-trivial. For example, shown in the side figure is the skeleton of the code that is inserted by PraPR before `INVOKEVIRTUAL` instructions when applying the mutator **MG**. In this code snippet, we assume that the callee method returns a reference-typed object, m is the number of arguments of the callee, and x, depending

```
xSTORE temp_m
...
xSTORE temp_1
DUP
IFNONNULL restore
POP
ACONST_NULL
goto escape
restore:
xLOAD temp_1
...
xLOAD temp_m
INVOKEVIRTUAL ...
escape:
```

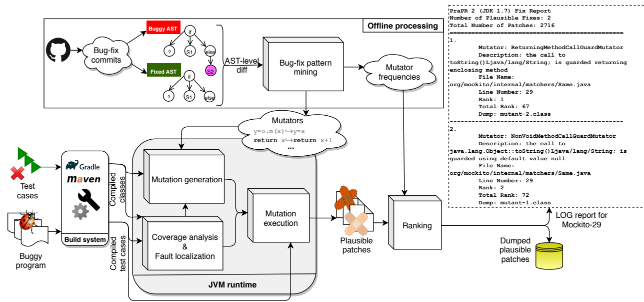

Fig. 1: PraPR workflow overview

on the type of the parameters of the callee, could be `I` (`int`), `L` (`long`), etc. The mutation is done as follows. First, we create m temporary local variables for each parameter of the callee and store the argument values in the variables (using the leading group of x`STORE`s). Then, we check if the receiver is `null` (note that we duplicate the reference to the receiver object since the instruction `IFNONNULL` consumes an object reference from the stack): if it is `null`, we pop the other copy of the receiver object from the stack, push the intended default value, and continue normal execution by jumping to label `escape`; otherwise, we push the arguments back to stack and invoke the target method.

An architectural overview of PraPR is depicted in Fig. 1. PraPR is available in the form of Maven/Gradle plugin and it is invoked by the build system. The workflow is as follows. The class files for a buggy program together with a set of test cases, containing at least one failing test case that exposes the bug, are fed to the system. In *Coverage analysis & Fault localization* phase, by collecting test coverage information, we compute suspiciousness values (Ochiai [25], by default) for each covered JVM instruction. Coverage and suspiciousness information are used by *Mutation generation* phase to generate a pool of mutants by applying the mutators described above. We exhaustively mutate instructions that are covered by all failing tests, since only such mutations are likely to be fixes. In *Mutation execution* phase, we apply all of test cases on each of the mutants to identify plausible patches. Inspired by [26], we improve the speed of patch validation by sorting the test cases based on their previous running time in ascending order hoping the nonviable mutants get killed as soon as possible. We also run the originally failing test cases first for they are most likely to fail again. Plausible patches contain information about the mutated location and its suspiciousness value. Thus, in the *Ranking* phase, we can sort the plausible patches based on their suspiciousness values in descending order. We break the ties by using the information about mutation frequencies obtained by mining HD-Repair dataset of real-world bug fix commits [7]. Finally, PraPR generates a human-readable fix report that contains enough information for applying each patch on the source code. The plausible patches also get dumped onto the disk, and the users can use various Java decompilers (e.g. [27]) to decompile the mutants and see the mutations in the context of decompiled source code which increases understandability of fix reports.

Please note that the upper half of Fig. 1, shows the offline process of mining mutators, and their frequencies, from HD-Repair dataset. We use GumTree framework to infer the operations transforming a buggy program to a fixed one.

III. Design Decisions

In this section, we elaborate on the objectives mentioned in §I and discuss the impacts of design decisions made in implementation of PraPR on achieving each of the objectives. A brief comparison to the most recent state-of-the-art APR techniques is also made.

A. Scalability

An important goal in constructing a practical, industrial-strength APR technique is to make it scalable to large, real-world programs. Such a technique should be able to produce genuine patches, otherwise it might mislead the developers rather than helping them [19], [28], [29].

PraPR does not need any kind of complicated computation (e.g., symbolic execution or constraint solving) that limits scalability. Thanks to this fact and the bytecode-level manipulation, PraPR (with only single thread) is already *over an order of magnitude* faster than state-of-the-art SimFix [5], CapGen [6], JAID [8] (that reduces compilation overhead by bundling patches in meta-programs), and SketchFix [9] (that curtails compilation overhead via sketching).

The speed in patch generation and validation makes it possible for PraPR to use a larger set of mutators to exhaustively mutate every suspicious location in the buggy program, which in turn enables the tool explore a larger search space in a reasonable amount of time. Our experiments show that PraPR successfully fixes 43/395 bugs from Defects4J V1.2.0 [30], outperforming state-of-the-art APR techniques (e.g., the most recent SimFix and CapGen fix 34 and 22 bugs, resp.). We further applied PraPR on 192 additional bugs from Defects4J V1.4.0 [31] from which the tool successfully fixed 12 bugs. Meanwhile, CapGen produces genuine patches for only 2 bugs, while SimFix was unable to generate any plausible patch, in spite of exhausting its search space for most cases, and timed out (a 5-hour limit) in 52 bugs. Also, CapGen generates thousands of plausible but incorrect patches for the additional bugs, while PraPR shows a decent level of consistency both in the number of fixed bugs and the false positives.

The notable performance drop of CapGen on the new dataset is because, for performance reasons, the tool applies only a subset of its mutators that happen to be ineffective on the new bugs. Lastly, as also confirmed by SimFix authors, SimFix was unable to locate reusable code snippets in the new dataset. This indicates that simplistic bytecode-level mutation is a viable approach for constructing a scalable APR tool.

B. Applicability

Program source code contains a wealth of information that researchers might exploit to develop more effective APR techniques. However, the process of mutation and/or extraction of fixing ingredients can be significantly different from one programming language to another. This makes APR techniques

```
28 appendQuoting(description);
29 -description.appendText(wanted.toString());
29 +description.appendText(wanted == null ? "null" : wanted.toString());
30 appendQuoting(description);
```

```
/*28*/ this.appendQuoting(description);
/*29*/ description.appendText(this.wanted == null?null:this.wanted.toString());
/*30*/ this.appendQuoting(description);
```

Fig. 2: Developer fix for the bug Mockito-29, and decompiled patch generated by Eclipse Decompiler [27] to its right

to be *hardwired* to work with a specific programming language. With the advent of more expressive, and less verbose, JVM-based programming languages such as Java 8 (which adds many syntactic sugars to the older versions of the language), Kotlin, Scala, and Groovy the need for applicability is especially pronounced for nowadays many real-world projects are written in a combination of these languages [32], so the APR techniques should be applicable in a uniform fashion.

PraPR works at the level of JVM bytecode that makes the tool JVM language agnostic and readily applicable to more than 6 popular programming languages [33]. We have applied PraPR to fix 118 Kotlin bugs from Defexts [32] database, and the tool successfully fixed 14 bugs. We stress that *this is first study on applying the same general-purpose APR technique on different programming languages*. A similar ratio of fixed bugs for the Kotlin systems also reduces threats to external validity for our claims, and shows that simplistic bytecode-level mutation alleviates the applicability challenge in the development of practical APR techniques.

C. Accurate Patch Validation

In a practical situation, we usually lack any kind of formal specifications. Thus, virtually all recent APR techniques depend on test cases so as to verify the generated patches. But since test cases usually underspecify the desired behavior of the system, we end up with a large number of plausible but incorrect patches (a.k.a. *test case overfitted patches* [34]). In the absence of an effective automatic classifier, the developer has to verify each and every one of the plausible patches.

Lately, several techniques for identification of test case overfitted patches, ranging from manual [28], [34] to fully automatic [29], [35], has been proposed. Unfortunately, none of the automatic techniques were applicable in our case; this is mainly due to two reasons: (1) PraPR makes tiny changes to the program which are difficult to be distinguished by the syntactic and semantic heurisitcs studied in [35]; (2) PraPR targets JVM-based languages, so the idea of fuzzing [29] is not effective [35]. Furthermore, we realized that anti-patterns [28] are also not applicable in our research since it is highly dependent on the C programming language.

We have mined HD-Repair dataset [7] to find the frequency in which our mutators appear in real-world bug fix commits. We prioritize our mutators based on the frequency of their appearance in the dataset. After ranking the patches according to the Ochiai suspiciousness [25] value of the mutated locations, we break the ties with regard to the priority of the mutators. This results in ranking 30/43 patches in Top-1 position.

Backed by our experimental results, we emphasize that ranking based on the frequencies of the bytecode-level mutators is generalizable to Java and Kotlin. Applying this technique in experiments with the Kotlin systems also shows an improvement in the number of patches in Top-1 position.

IV. PraPR Usage

We have implemented PraPR as a 1-click APR tool publicly available on Maven Central Repo [36]. Being compatible with a variety of testing frameworks (e.g., JUnit, TestNG, and Spek), PraPR is readily applicable to arbitrary Java projects under Maven/Gradle build systems (not just Defects4J) and even projects in other JVM languages in a hassle-free manner, thereby allowing researchers replicate our experiments.

In a Maven project, in order to use PraPR plugin, all the developer needs is to add the following snippet under `<plugins>` tag in the POM file for the project.

```
<plugin>
  <artifactId>prapr-plugin</artifactId>
  <groupId>org.mudebug</groupId>
  <version>2.0.0</version>
</plugin>
```

By default, PraPR shall apply all of its mutators and use single thread of execution to validated generated mutants. Also, default output format is `LOG`; meaning that the tool will generate human-readable fix reports (the report for Mockito-29 from Defects4J is shown in Fig. 1). Besides `LOG` reports, PraPR can generate two other types of reports: (1) pretty-printed HTML file for the source files and description of the patches applied at different locations; (2) compressed XML file with detailed information about each mutation, suitable for machine processing. By adding a `<configuration>` section under `<plugin>`, one can customize the behavior of the tool. Our companion website [36] contains more information about PraPR configuration and the details about using PraPR in Gradle projects. The website also contains instructions for running the tool from a self-contained Docker image.

Before running PraPR plugin, the user needs to compile source and test files to generate the `.class` files. Depending on the build system, different commands can be used to invoke the tool. In case of a Maven project, the command `mvn org.mudebug:prapr-plugin:prapr`, while for a Gradle project `gradle prapr-repair`, is used to invoke PraPR to fix a bug. Once PraPR finishes running, it will generate report(s) and dump (mutated) `.class` files for plausible fixes. Based on our experience in using PraPR in 900 real-world Java/Kotlin projects, we emphasize that the generated fix reports contain enough information for the patches to be understood and applied at the level of source code. Meanwhile, the users can always decompile [27] the dumped mutants to see the patches in the context of the source code that matches the actual source code for the program. Fig. 2 shows an example of developer patch and decompiled patch which shows the patch in the context of original source code (with automatically generated line number information).

V. Related Work

In §III, we briefly compared the effectiveness of PraPR with state-of-the-art SimFix [5] and CapGen [6].

In this section, we give a more complete account of comparison with related APR techniques. Specifically, we compare PraPR with the APR techniques that have been evaluated on Defects4J V1.2.0 before, including SimFix, CapGen, JAID [8], SketchFix [9], ELIXIR [10], ssFix [11], ACS [12], HD-Repair [7], xPAR [7] (a reimplementation of PAR [23]), NOPOL [13], jGenProg [37] (a reimplementation of GenProg [14] for Java), jMutRepair [37] (a reimplementation of source-level mutation-based repair [15] for Java), and jKali [37] (a reimplementation of Kali [16] for Java). Fig. 3 illustrates the distribution of the bugs that can be successfully fixed by PraPR and the aforementioned APR techniques. We observe that PraPR can fix 10 bugs that have not been fixed by any of other techniques. Also, the tools are complementary, i.e. putting all the tools together, we can fix 90+ Defects4J bugs.

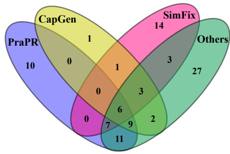

Fig. 3: Fixed bugs dist.

We conclude this section by discussing other techniques that also operate at the level of JVM bytecode. Ma et al. leveraged domain knowledge to fix cryptography misuses for Android apps at the bytecode level [38]. Schulte et al. discussed the possibility to fix bugs through evolution of assembly code [39]. In their paper [40], Staples et al. introduce a semi-automatic bytecode repair framework for mitigating security vulnerabilities. PraPR is the first general-purpose APR technique at the bytecode level.

VI. Conclusion and Future Work

We have implemented PraPR, the first practical, general-purpose APR tool at the JVM bytecode level. The experimental results on the widely used Defects4J V1.2.0 benchmark show that PraPR can generate genuine patches for 43 bugs, significantly outperforming state-of-the-art Java repair techniques, while being 10+X faster; with no learning/search information, PraPR also avoids the overfitting problem of advanced techniques on additional bugs from Defects4J V1.4.0. Lastly, PraPR successfully fixed 14 of the 118 studied bugs for Kotlin programs. We are currently working on integrating PraPR with state-of-the-art fault localization [41]–[44].

References

[1] https://tinyurl.com/y3qea8go, accessed: Jun-12-2019.

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

[3] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *TSE*, pp. 34–67, 2017.

[4] F. Long, S. Sidiroglou-Douskos, and M. C. Rinard, "Automatic runtime error repair and containment via recovery shepherding," in *PLDI*, 2014.

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

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

[7] X. B. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *SANER*, vol. 1. IEEE, 2016, pp. 213–224.

[8] L. Chen, Y. Pei, and C. A. Furia, "Contract-based program repair without the contracts," in *ASE*, 2017, pp. 637–647.

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

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

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

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

[13] 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," *TSE*, pp. 34–55, 2017.

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

[15] V. Debroy and W. E. Wong, "Using mutation to automatically suggest fixes for faulty programs," in *ICST*, April 2010, pp. 65–74.

[16] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *ISSTA*. ACM, 2015, pp. 24–36.

[17] Z. Y. Ding, Y. Lyu, C. S. Timperley, and C. L. Goues, "Leveraging program invariants to promote population diversity in search-based automatic program repair," in *GI*. IEEE, 2019, pp. 2–9.

[18] C. S. Timperley, "Advanced techniques for search-based program repair," Ph.D. dissertation, University of York, 2017.

[19] X. B. D Le, "Overfitting in automated program repair: Challenges and solutions," Ph.D. dissertation, Singapore Management University, 2018.

[20] A. Ghanbari, S. Benton, and L. Zhang, "Practical program repair via bytecode mutation," in *ISSTA*, 2019, pp. 19–30.

[21] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java Virtual Machine Specification, Java SE 8 Edition*, 2014.

[22] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *TSE*, vol. 37, no. 5, pp. 649–678, 2011.

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

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

[25] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," in *TAICPART-MUTATION*, 2007.

[26] L. Zhang, D. Marinov, and S. Khurshid, "Faster mutation testing inspired by test prioritization and reduction," in *ISSTA*, 2013, pp. 235–245.

[27] "Eclipse Class Decompiler ," https://ecd-plugin.github.io/ecd/.

[28] S. H. Tan, H. Yoshida, M. R. Prasad, and A. Roychoudhury, "Anti-patterns in search-based program repair," in *FSE*, 2016, pp. 727–738.

[29] J. Yang, A. Zhikhartsev, Y. Liu, and L. Tan, "Better test cases for better automated program repair," in *FSE*. ACM, 2017, pp. 831–841.

[30] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *ISSTA*. ACM, 2014, pp. 437–440.

[31] https://github.com/Greg4cr/defects4j/tree/additional-faults-1.4.

[32] S. Benton, A. Ghanbari, and L. Zhang, "Defexts: A curated dataset of reproducible real-world bugs for modern jvm languages," in *ICSE*, 2019.

[33] "Wikipedia", "List of JVM Languages," https://tinyurl.com/cgy8pqv, 2019, accessed May-19-2019.

[34] Q. Xin and S. P. Reiss, "Identifying test-suite-overfitted patches through test case generation," in *ISSTA*. ACM, 2017, pp. 226–236.

[35] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang, "Identifying patch correctness in test-based program repair," in *ICSE*, 2018, pp. 789–799.

[36] "PraPR," https://github.com/prapr/prapr, accessed May-27-2019.

[37] M. Martinez and M. Monperrus, "Astor: A program repair library for java (demo)," in *ISSTA*, 2016, pp. 441–444.

[38] S. Ma, D. Lo, T. Li, and R. H. Deng, "Cdrep: Automatic repair of cryptographic misuses in android applications," in *ASIACCS*, 2016.

[39] E. Schulte, S. Forrest, and W. Weimer, "Automated program repair through the evolution of assembly code," in *ASE*, 2010, pp. 313–316.

[40] J. Staples, C. Endicott, L. Krause, P. Pal, P. Samouelian, R. Schantz, and A. Wellman, "A semi-autonomic bytecode repair framework," *IEEE Software*, vol. 36, no. 2, pp. 97–102, 2019.

[41] X. Li, W. Li, Y. Zhang, and L. Zhang, "Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization," in *ISSTA*, 2019.

[42] J. Sohn and S. Yoo, "Fluccs: using code and change metrics to improve fault localization," in *ISSTA*, 2017, pp. 273–283.

[43] M. Zhang, X. Li, L. Zhang, and S. Khurshid, "Boosting spectrum-based fault localization using pagerank," in *ISSTA*, 2017, pp. 261–272.

[44] X. Li and L. Zhang, "Transforming programs and tests in tandem for fault localization," *OOPSLA*, pp. 92:1–92:30, 2017.