Practical Program Repair *via* Bytecode Mutation

Ali Ghanbari
University of Texas at Dallas, USA
ali.ghanbari@utdallas.edu

Samuel Benton
University of Texas at Dallas, USA
samuel.benton1@utdallas.edu

Lingming Zhang
University of Texas at Dallas, USA
lingming.zhang@utdallas.edu

ABSTRACT

Automated Program Repair (APR) is one of the most recent advances in automated debugging, and can directly fix buggy programs with minimal human intervention. Although various advanced APR techniques (including search-based or semantic-based ones) have been proposed, they mainly work at the source-code level and it is not clear how bytecode-level APR performs in practice. Also, empirical studies of the existing techniques on bugs beyond what has been reported in the original papers are rather limited. In this paper, we implement the first practical bytecode-level APR technique, PraPR, and present the first extensive study on fixing real-world bugs (e.g., Defects4J bugs) using JVM bytecode mutation. The experimental results show that surprisingly even PraPR with only the basic traditional mutators can produce genuine fixes for 17 bugs; with simple additional commonly used APR mutators, PraPR is able to produce genuine fixes for 43 bugs, significantly outperforming state-of-the-art APR, while being over 10X faster. Furthermore, we performed an extensive study of PraPR and other recent APR tools on a large number of additional real-world bugs, and demonstrated the overfitting problem of recent advanced APR tools for the first time. Lastly, PraPR has also successfully fixed bugs for other JVM languages (e.g., for the popular Kotlin language), indicating PraPR can greatly complement existing source-code-level APR.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Program repair, Mutation testing, Fault localization, JVM bytecode

ACM Reference Format:
Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. Practical Program Repair *via* Bytecode Mutation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19), July 15–19, 2019, Beijing, China.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3293882.3330559

1 INTRODUCTION

Software systems are ubiquitous in today's world; most of our activities, and sometimes even our lives, depend on software. Unfortunately, software systems are not perfect and often come with bugs. Software debugging is a challenging activity that consumes over 50% of the development time/effort [78], and costs the global economy billions of dollars [17]. To date, a huge body of research has been dedicated to automatically localize [10, 13, 14, 44, 46, 48, 67, 79, 87, 94, 97] or fix [18, 21, 22, 28, 30, 35, 43, 49–51, 56, 58, 61, 66, 68, 76, 80, 82, 84, 92] software bugs. Automated Program Repair (APR) techniques aim to directly fix software bugs with minimal human intervention, and has been under intense research in spite of being a young research area [28].

Based on the actions taken for fixing a bug, state-of-the-art APR techniques can be divided into two broad categories: (1) techniques that monitor the dynamic execution of a program to find deviations from certain specifications, and then *heal* the program by modifying its runtime state in case of any abnormal behavior [51, 68]; (2) *generate-and-validate* (G&V) techniques that modify program code representations based on various rules/techniques, and then use either tests or formal specifications as the oracle to validate each generated candidate patch for finding *plausible* patches (i.e., the patches that can pass all the tests/checks), which are further checked to find *genuine* patches (i.e., the patches semantically equivalent to developer patches) [18, 21, 22, 30, 35, 43, 48–50, 56, 61, 66, 80, 85, 92]. Among these, G&V techniques, especially those based on tests, have gained popularity as testing is the prevalent way for detecting bugs, while very few systems are based on rigorous, formal specifications.

It is worth noting that, lately, multiple APR research papers get published in Software Engineering conferences and journals each year, introducing various delicately designed and/or implemented APR techniques. With such state-of-the-art APR techniques, more and more real bugs can be fixed fully automatically, e.g., the recent CapGen [85] technique, published in ICSE'18, has been reported to produce genuine patches for 22 bugs of Defects4J (a suite of real-world Java programs widely used for evaluating APR techniques [38]). Despite the success of recent APR techniques, as also highlighted in a recent survey [28], currently we have a *scattered* collection of findings and innovations with no thorough evaluation of them. In particular, it is not clear how a simplistic bytecode-mutation approach works for APR in practice.

In this paper, we present the first extensive study on APR techniques, with an emphasis on bytecode-level APR, on the widely used Defects4J dataset [31, 38]. To this end, we build a practical APR tool named PraPR (**Pra**ctical **P**rogram **R**epair) based on a set of simple JVM bytecode [47] mutation rules, including basic mutators from traditional mutation testing [36] (e.g., changing `a>=b` into `a>b`) and augmented mutators that occur frequently in real-world bug-fix commits (e.g., replacing field accesses or method invocations). We stress that although simplistic, PraPR offers various benefits and can complement state-of-the-art techniques. First, all the patches that PraPR generates can be directly validated without compilation, while existing techniques [18, 21, 22, 30, 35, 37, 43, 49, 50, 56, 61, 66, 80, 85, 92] have to compile and load each candidate patch. Even

ISSTA '19, July 15–19, 2019, Beijing, China

ACM ISBN 978-1-4503-6224-5/19/07…$15.00
https://doi.org/10.1145/3293882.3330559

though some techniques curtail compilation overhead by encoding a group of patches inside a single meta-program, it can still take up to 37 hours to fix a Defects4J program *due to numerous patch compilations and loading* [18]. Second, bytecode-level repair avoids messing up the source code in unexpected ways, and can even be applicable for fixing code without source code information, e.g., buggy 3rd-party libraries that do not have official patches yet. Third, manipulating programs at the level of JVM bytecode makes PraPR independent of the syntax of a specific target programming language, and applicable to different Java versions and even other popular JVM-based languages (notably Kotlin [34], Scala [62], and Groovy [25]). Lastly, PraPR does not require complex patching rules [43, 49, 92], complicated computations such as symbolic execution and constraint solving [18, 56, 61], or any training/mining [37, 71, 85, 91], making it directly applicable to real-world programs and easily adoptable as the baseline for future APR techniques.

We have applied PraPR to fix all the 395 bugs available in Defects4J V1.2.0. Surprisingly, even the basic traditional mutators can already produce genuine fixes for 17 bugs. With both the traditional and the augmented mutators, PraPR successfully produces genuine fixes for 43 bugs, thereby significantly outperforming state-of-the-art APR techniques (e.g., the recent CapGen [85] fixes only 22 bugs). Also, thanks to the bytecode-level manipulation, PraPR with only single thread can already be over an order of magnitude faster than state-of-the-art SimFix [37], CapGen, JAID [18] (that reduces compilation overhead by bundling patches in meta-programs), and SketchFix [33] (that reduces compilation overhead via sketching). We further study PraPR (and other recent APR tools) on 192 additional bugs from Defects4J V1.4.0 and bugs from another popular JVM language, Kotlin. The paper makes the following contributions:

- **Study.** We perform the first extensive study on the performance and efficiency of both source-code-level and bytecode-level APR techniques on 395 real-world Java bugs from Defects4J V1.2.0 [38]. We are also the first to evaluate recent advanced APR techniques on the 192 additional bugs from Defects4J V1.4.0 [31]. Furthermore, we report the first repair study on Kotlin bugs from Defexts [16] (a dataset with 225 real-world Kotlin bugs).
- **Implementation.** We implement a full-fledged practical program repair tool for JVM bytecode, PraPR (available on Maven Central Repo and GitHub [29]). To our knowledge, this is also the first general-purpose *polyglot* APR technique for JVM-based languages. Furthermore, we were unable to successfully apply the other studied APR tools on the bugs other than the ones in the original papers. We actively worked with the authors to address that: we reported several bugs to the CapGen authors, and also directly contributed to enable CapGen to run on more projects; we also managed to write our own code to produce all information needed by SimFix for fixing arbitrary Java programs.
- **Results.** Our results demonstrate that on Defects4J V1.2.0 PraPR can fix more bugs than the state-of-the-art APR techniques, while being over 10X faster. Also, PraPR showed a decent level of consistency both in the number of false positives and successfully fixed bugs when applied to additional bugs from Defects4J V1.4.0, while other techniques suffer from overfitting. Furthermore, PraPR successfully fixed various Kotlin bugs from Defexts.
- **Guidelines.** Our findings demonstrate for the first time that simple bytecode mutations can greatly complement state-of-the-art APR techniques in at least three aspects (effectiveness, efficiency, and applicability), and can inspire more work to advance APR in this direction.

2 RELATED WORK

2.1 Mutation Testing

Mutation testing [11] is a powerful method for assessing the quality of a given test suite in detecting potential software bugs. Mutation testing measures test suite quality via injecting "artificial bugs" into the subject programs. The basic intuition is that the more artificial bugs that a test suite can detect, the more likely is it to detect potential real bugs, hence the test suite is of higher quality [12, 39]. Central to mutation testing is the notion of *mutation operator*, aka *mutator*, which is used to generate artificial bugs to mimic real bugs. Applying a mutator on a program results in a *mutant* (or *mutation*) of the program—a variant of the program that differs from the original program only in the injected artificial bug, e.g., replacing `a+b` with `a-b`. This suggests that the resulting mutants should be syntactically valid and typeable, and the mutators are highly dependent on the target programming language.

Given a program $\mathcal{P}$, mutation testing will generate a set of mutants $\mathcal{M}$. Given a mutant $m \in \mathcal{M}$ of the program, a test suite $\mathcal{T}$ is said to *kill* mutant m if and only if there exists at least one test $t \in \mathcal{T}$ such that the observable final state of $\mathcal{P}$ on t differs from that of m on t, i.e., $\mathcal{P}[\![t]\!] \neq m[\![t]\!]$. Similarly, a mutant is said to *survive* if no test in $\mathcal{T}$ can kill it. Some of the survived mutants might be (semantically) *equivalent* to the original program, hence no test can ever kill such *equivalent mutants*. By having the number of killed and equivalent mutants for a given test suite $\mathcal{T}$, one may easily compute a *mutation score* to evaluate the quality of $\mathcal{T}$, i.e., the ratio of killed mutants to all non-equivalent mutants ($\mathcal{MS} = \frac{|\mathcal{M}_{killed}|}{|\mathcal{M}|-|\mathcal{M}_{equivalent}|}$). Besides its original application in test suite evaluation, recently mutation testing has also been widely applied in various other areas, such as simulating real bugs for software-testing experiments [12, 39], automated test generation [65, 95], fault localization [45, 46, 59, 63, 96], and even automated program repair [22, 55] and build repair [52]. When using mutation testing for program repair, the inputs are a buggy program $\mathcal{P}$ and its corresponding test suite $\mathcal{T}$ with failed tests due to the bug(s). The output will be a subset $M \subseteq \mathcal{M}$ of mutants that pass all the tests within $\mathcal{T}$. Such resulting mutants are plausible fixes for $\mathcal{P}$.

2.2 Generate-and-Validate Program Repair

Modern G&V APR techniques usually first utilize existing fault localization [10, 13, 87] techniques to identify suspicious code elements, and then systematically change/insert/delete code at suspicious locations to search for a new program variant that can produce expected outputs. In practice, tests play a central role in both localizing the bugs and also checking if a program variant behaves as expected—i.e., tests are also used as *fix oracles*. Fault localization techniques use the information obtained from both failing and passing tests to compute degrees of suspiciousness for each element of the program. For example, *spectrum-based fault localization* techniques [87], which identify the program elements covered by more failed tests and less passed tests as more suspicious, have been widely adopted by various APR techniques [28, 55, 58]. Modifying a buggy program results in various *candidate patches* that could be verified using the available test suite. A candidate patch that can pass all the failing and passing tests within the original test suite is called a *plausible patch*, while a patch that not only passes all the tests but is also semantically equivalent to the corresponding developer patch denotes a *genuine patch*. Note that, due to the APR-*overfitting* problem [28, 32, 58, 70, 93], not all plausible patches might be considered genuine patches. Overfitting is a principal problem with test-driven G&V APR because of its dependence on the test suites to verify patches. In practice, test suites are usually not perfect, and a patch passing the test suite may not generalize to other potential tests of the program. Thus, various techniques [56, 61, 88, 90] have been proposed to mitigate overfitting.

Based on different hypotheses, state-of-the-art G&V APR tools use a variety of techniques to generate or synthesize patches. *Search-based* APR techniques are based on the hypothesis that most bugs could be solved by searching through all the potential candidate patches based on certain patching rules [22, 43]. Alternatively, *semantic-based* techniques use deeper semantical analyses (such as symbolic execution) to synthesize conditions, or even more complex code snippets, that can pass all the tests [56, 61, 83, 92]. There are also various other studies on APR techniques: while some studies show that generating patches just by deleting the original software functionality can be effective [69, 70], other studies [43, 85] demonstrate that fix ingredients could be adopted from somewhere in the buggy program itself or even other programs based on the *plastic surgery* hypothesis [15]. As discussed earlier, mutation testing has also been applied for APR. The hypothesis for mutation-based APR is that "*if the mutators mimic programmer errors, mutating a defective program can, therefore, fix it*" [22]. However, the existing studies either concern mutation-based APR on a set of small programs (e.g., the Siemens Suite [1]) with artificial bugs [22] or apply only a limited set of mutators [55]. For example, the most recent study [55] on mutation-based APR with 3 mutators shows that it can fix only 4 Defects4J bugs. Furthermore, all the existing studies [22, 55, 69] apply mutation at the source code level, which can incur substantial compilation/class-loading overhead and is language-dependent. Ma et al. leveraged domain knowledge to fix cryptography misuses for Android apps at the bytecode level [53]. Schulte et al. discussed the possibility to fix bugs through evolution of assembly code [74]. We present and study the first general-purpose mutation-based APR technique at the bytecode level.

3 PRAPR

This section first presents the overall approach of PraPR (§3.1), and then discusses mutator design (§3.2), which makes up the core of PraPR. Both our overall approach and mutator design are simplistic for easy result reproduction and future extension.

3.1 Overall Approach

The overall approach of PraPR is presented in Algorithm 1. The algorithm inputs are the original buggy program $\mathcal{P}$ and its test suite $\mathcal{T}$ that can detect the bug(s). For the ease of illustration, we represent the passing and failing tests in the test suite as $\mathcal{T}_p$ and $\mathcal{T}_f$, respectively. The algorithm output is $\mathbb{P}_{\checkmark}$, a set of plausible patches that

Algorithm 1: PraPR

Input: Original buggy program $\mathcal{P}$, failing tests $\mathcal{T}_f$, passing tests $\mathcal{T}_p$
Output: Plausible patch set $\mathbb{P}_{\checkmark}$

```
1  begin
2      ℒ ← FaultLocalization(𝒫)// Fault localization
3      ℙ ← MutGen(𝒫, ℒ) // Candidate patch generation
       /* Perform validation for each candidate patch */
4      for 𝒫′ ∈ ℙ do
5          falsified=False// Whether the patch is falsified
6          𝒯′ ← Cover(Diff(𝒫′, 𝒫))
7          if !𝒯′ ⊇ 𝒯f then continue;
           /* Check if originally failed tests still fail */
8          for t ∈ 𝒯f do
9              if 𝒫′⟦t⟧ = failing then
10                 falsified=True
11                 break // Abort current patch validation
12         if falsified=True then continue;
           /* Check if any originally passed test fails */
13         for t ∈ 𝒯p ∩ 𝒯′ do
14             if 𝒫′⟦t⟧ = failing then
15                 falsified=True
16                 break // Abort current patch validation
17         if falsified=False then
18             ℙ✓ ← ℙ✓ ∪ {𝒫′}// Store current plausible patch
19     return ℙ✓ // Return the resulting patch set
```

can pass all the tests in $\mathcal{T}$, and the developers can further inspect $\mathbb{P}_{\checkmark}$ to check if there is any genuine patch. Shown in the algorithm, Line 2 first computes and ranks the suspicious program locations $\mathcal{L}$ using off-the-shelf fault localization techniques (e.g., Ochiai [10] for this work). Line 3 then exhaustively generates candidate patches $\mathbb{P}$ for all suspicious locations (i.e., the locations executed by any failed test) using our mutators presented in §3.2. Following prior APR work [18, 55, 85], patches modifying more suspicious locations obtain a higher rank. Then, Lines 4 to 18 iterate through each candidate patch to find plausible patches.

To ensure efficient patch validation, following prior work [55, 85], each candidate patch is firstly executed against the failed tests (Lines 8-11), and will only be executed against the remaining tests once it passes all the originally failed tests. The reason is that the originally failed tests are more likely to fail again on candidate patches, whereas the patches failing any test are already falsified, and do not need to be executed against the remaining tests for sake of efficiency. Furthermore, we also apply two additional optimizations widely used in the mutation testing community (e.g., PIT [19] and Javalanche [73]). First, all the candidate patches are directly generated at the JVM bytecode level to avoid expensive recompilation of a huge number of candidate patches. Second, PraPR computes the tests covering the patched location (i.e., statements) of each candidate patch as $\mathcal{T}'$ (Line 6) to safely reduce test executions (recent APR techniques also adopted this optimization [33, 57]). For failing tests, if $\mathcal{T}'$ does not subsume $\mathcal{T}_f$, the candidate patch can be directly skipped since the patched location is not covered by all failed tests and thus cannot make all failed tests pass (Line 7); for passing tests, PraPR only needs to check the patch against the tests covering the patched location (Line 13) since the other passing tests do not touch the patched location and will still pass. If the patch passes all tests, it will be recorded in the resulting plausible patch set $\mathbb{P}_{\checkmark}$. Finally, PraPR returns $\mathbb{P}_{\checkmark}$ (Line 19).

Note that the bytecode-level patches include enough information for the developers to confirm/reject the patches and apply them to the source code. Shown in Figure 1, the two example bytecode-level

```
PraPR 2 (JDK 1.7) Fix Report - Mon Jan 14 21:01:01 CST 2019
Number of Plausible Fixes: 2
Total Number of Patches: 416
================================================
 1. Mutator: METHOD CALL (the call to java.lang.Character::isWhitespace(C)Z
        is replaced with the used of default value false)
 File Name: org/apache/commons/lang3/time/FastDateParser.java
 Line Number: 307
------------------------------------------------
 2. Mutator: CONDITIONAL (removed conditional - replaced equality check
        with false)
 File Name: org/apache/commons/lang3/time/FastDateParser.java
 Line Number: 307
```

Contents of the file org/apache/commons/lang3/time/FastDateParser.java for Lang 10:

```
305 for(int i= 0; i<value.length(); ++i) {
306 char c= value.charAt(i);
307 if(Character.isWhitespace(c)) {
308 ...
```

Figure 1: Two example patch reports automatically generated by PraPR (for bug Lang-10). Underlined parts convey sufficient information for locating and fixing the buggy if-statement shown in the bottom part of the figure.

```
appendQuoting(description);
~~description.appendText(wanted.toString());~~
+++description.appendText(wanted == null ? "null" : wanted.toString());
appendQuoting(description);
```

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

Figure 2: Developer fix for the bug Mockito-29, and decompiled patch generated by PraPR below it (with automatically generated line number information)

PraPR patches (in the first half of the figure) include sufficient debugging information, and it is trivial for the developers to understand and apply the patches. In addition, as shown in Figure 2, PraPR also supports automatically decompiling the mutated bytecode to present patched lines in the source-code format.

3.2 PraPR Mutators

PraPR mutators are intended to mutate the input programs via simple transformation rules that affect only one program statement at a time. All our mutators are implemented at the JVM level for sake of efficiency, and our implementation, for which we put a considerable engineering effort, supports the full set of JVM instructions and data types. For simplicity in presentation though, we chose to present all our mutators in a core Java language, named ClassicJava [24]. Our goal is to describe the mutators using a minimal subset of Java so that the functionality of the mutators could be described simply, yet unambiguously. Figure 3 presents the abstract syntax of an extended version of the ClassicJava. The full definition of the operational semantics and type-rules for the core part of ClassicJava, could be found in the original paper [24].

Table 1 presents the details of PraPR mutators in rewrite rules. Each rule is represented in the form of $p \vdash e \hookrightarrow e'$, which denotes that when the premise p holds, a candidate patch can be generated via mutating a single instance of expression e to e' (note that all the other portions of the input program is intended to remain unchanged). In the case of no premises, p is omitted, e.g. as in $\vdash e \hookrightarrow e'$. In addition, the overloaded operator $\tau(\cdot)$ computes typing information if the input is an expression and returns a type descriptor (i.e., the parameter types and return types according to JVM

$$
\begin{array}{rcl}
P & = & \mathit{defn}^*\ e \\
\mathit{defn} & = & \textbf{class}\ c\ \textbf{extends}\ c\ \textbf{implements}\ i^*\{\mathit{field}^*\ \mathit{meth}^*\} \\
 & & \mid \textbf{interface}\ i\ \textbf{extends}\ i^*\{\mathit{meth}^*\} \\
\mathit{field} & = & t\ \mathit{fd} \\
\mathit{meth} & = & t\ \mathit{md}(\mathit{arg}^*)\{\mathit{body}\} \mid \textbf{void}\ \mathit{md}(\mathit{arg}^*)\{\mathit{body}\} \\
\mathit{arg} & = & t\ \mathit{var} \\
\mathit{body} & = & e \mid \textbf{abstract} \\
e & = & ct \mid ae \mid be \mid \textbf{new}\ c \mid \mathit{var} \mid e.\mathit{fd} \mid e.\mathit{fd}{=}e \\
 & & \mid e.\mathit{md}(e^*) \mid \textbf{super}.\mathit{md}(e^*) \mid \textbf{let}\ \mathit{var} = e\ \textbf{in}\ e \\
 & & \mid be\ ?\ e : e \mid \textbf{switch}(e)\ (\textbf{case}\ ct\text{: } e)^*\ \textbf{default:}\ e \\
 & & \mid \textbf{fail} \mid \textbf{return}\ e \mid \mathit{var}{+}{+} \mid \mathit{var}{-}{-} \mid e\,;\,e \\
 & & \mid \textbf{try}\ \{\ e\ \}\ \textbf{catch}\ (c\ \mathit{var})\ \{\ e\ \} \mid \textbf{throw}\ e \\
ae & = & n \mid e + e \mid {-e} \mid e - e \mid \ldots \\
be & = & !e \mid e\ \&\&\ e \mid e == e \mid e < e \mid \ldots \\
\mathit{var} & = & \text{a variable name or } \textbf{this} \\
c & = & \text{a class name or } \textbf{Object} \\
i & = & \text{an interface name or } \textbf{Empty} \\
\mathit{fd} & = & \text{a field name} \\
\mathit{md} & = & \text{a method name} \\
t & = & c \mid i \mid \textbf{int} \mid \textbf{boolean} \\
ct & = & \text{n} \mid \textbf{true} \mid \textbf{false} \mid \textbf{null} \\
n & = & \text{an integer}
\end{array}
$$

Figure 3: Abstract syntax of extended ClassicJava

Table 2: Mutator illustration

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

specification [47]) when the input is the fully qualified name of a method. Function defVal$(\cdot)$, given a type-name, returns the default value corresponding to a given type as described in JVM specification [47]. $\tau_1 \leq \tau_2$ denotes that type τ_1 is a subtype of τ_2. Table 2 presents some example mutators.

In the table, the white block presents all the mutators directly supported by PIT. Note that although a slightly different categorization is used here, the table includes all the official PIT mutators. The light-gray block presents our augmented mutators used for expression replacement. Finally, the dark-gray block presents all our augmented mutators responsible for inserting conditionals in the vicinity of method calls and field dereferences as guards, and at the entry/exit of methods as pre-/post-condition checkers. It is worth noting that we omit the presentation of PraPR mutators involving datatypes absent in the syntax of ClassicJava (e.g., **float** and **double**). We stress that our mutators are either well-known mutators extensively studied in mutation testing literature [11, 19, 64, 73] or developed to handle common, simple bugs *without* any bias towards the bugs in Defects4J (both expression relacement and conditional insertion are simple rules widely explored in prior repair work [33, 42, 71, 85, 91]). To further confirm the generality of PraPR mutators, we built a fix-pattern extraction program (with 4K LoC Java code) based on the GumTree AST diffing framework [23], to automatically extract fix patterns in another HD-Repair dataset [42] that comprises 3,000+ real patches from 700+ GitHub projects (overlapping projects with Defects4J were removed). Table 3 summarizes the set of mutators,

Table 3: PraPR mutator frequency in HD-Repair dataset

Mutator	Freq.	Mutator	Freq.
MR	8.76%	**IS**	0.15%
CO	2.26%	**RV**	0.09%
FR	2.17%	**TR**	0.09%
VR	1.80%	**FG**	0.09%
MC	0.95%	**CC**	0.06%
IC	0.76%	**MV**	0.06%
AP	0.37%	**PC**	0.06%
MG	0.31%	**SW**	0.00%
AO	0.15%	**IN**	0.00%

Table 1: Supported Mutators

ID	Mutator Name	Rules
AP	ARGUMENT PROPAGATION	$i \in \{0, \ldots, n\}, \tau(e_i) \preceq \tau(e_0.m(e_1, \ldots, e_n)), i > 0, \forall j > i.\tau(e_j) \npreceq \tau(e_0.m(e_1, \ldots, e_n)) \vdash e_0.m(e_1, \ldots, e_n) \hookrightarrow e_i$
RV	RETURN VALUE	$\tau(e) = \mathbf{boolean} \vdash \mathbf{return}\ e \hookrightarrow \mathbf{return}\ !e$
		$\tau(e) = \mathbf{int}, e' \in \{\mathbf{0}, (e == \mathbf{0}\ ?\ \mathbf{1} : \mathbf{0})\} \vdash \mathbf{return}\ e \hookrightarrow \mathbf{return}\ e'$
		$\tau(e) = \mathbf{Object}, e' \in \{\mathbf{null}, (e == \mathbf{null}\ ?\ \mathbf{fail} : e)\} \vdash \mathbf{return}\ e \hookrightarrow \mathbf{return}\ e'$
CC	CONSTRUCTOR CALL	$\vdash \mathbf{new}\ c\mathbf{()} \hookrightarrow \mathbf{null}$
IS	INCREMENTS	$\star, \star' \in \{{+}{+}, {-}{-}\}, \star \neq \star', e \in \{var, var\star'\} \vdash var\star \hookrightarrow e$
		$\star, \star' \in \{{+}{+}, {-}{-}\}, \star \neq \star', e \in \{var, \star' var\} \vdash \star var \hookrightarrow e$
IC	INLINE CONSTANTS	$n' \in \{\mathbf{0}, (n + \mathbf{1})\} \vdash n \hookrightarrow n'$
MV	MEMBER VARIABLE	$\tau(e_1.fd) = t, \mathrm{defVal}(t) = v \vdash e_1.fd = e_2 \hookrightarrow e_1.fd = v$
SW	SWITCH	$\vdash \mathbf{switch(}e\mathbf{)}\ \mathbf{case}\ ct_1\mathbf{:}\ e_1\ \ldots\ \mathbf{case}\ ct_n\mathbf{:}\ e_n\ \mathbf{default:}\ e_d \hookrightarrow \mathbf{switch(}e\mathbf{)}\ \mathbf{case}\ ct_1\mathbf{:}\ e_d\ \ldots\ \mathbf{case}\ ct_n\mathbf{:}\ e_d\ \mathbf{default:}\ e_1$
		$1 \leq i \leq n \vdash \mathbf{switch(}e\mathbf{)}\ \mathbf{case}\ ct_1\mathbf{:}\ e_1\ \ldots\ \mathbf{case}\ ct_n\mathbf{:}\ e_n\ \mathbf{default:}\ e_d \hookrightarrow \mathbf{switch(}e\mathbf{)}\ \ldots \mathbf{case}\ ct_i\mathbf{:}\ e_d \ldots\ \mathbf{default:}\ e_d$
MC	METHOD CALL	$\tau(e.md(e_1, \ldots, e_n)) = t, \mathrm{defVal}(t) = v \vdash e.md(e_1, \ldots, e_n) \hookrightarrow v$
		$\tau(md) = \mathbf{void}\ md(t_1, \ldots, t_n), \tau(e_1) = t_1, \ldots, \tau(e_n) = t_n \vdash e.md(e_1, \ldots, e_n) \hookrightarrow \square$
IN	INVERT NEGATIVES	$\tau(e) = \mathbf{int} \vdash \text{-}e \hookrightarrow e$
AO	ARITHMETIC OPERATOR	$\star, \star' \in \{+, -, *, /, \%, >>, >>>, <<, \&, \vert, \hat{}\}, \star \neq \star' \vdash e_1 \star e_2 \hookrightarrow e_1 \star' e_2$
CO	CONDITIONAL	$\star, \star' \in \{\leq, \geq, <, >, ==, !=\}, \star \neq \star' \vdash e_1 \star e_2 \hookrightarrow e_1 \star' e_2$
		$\star, \star' \in \{\leq, \geq, <, >, ==, !=\}, \star \neq \star' \vdash e_1 \star e_2 \hookrightarrow \mathbf{true}$
		$\star, \star' \in \{\leq, \geq, <, >, ==, !=\}, \star \neq \star' \vdash e_1 \star e_2 \hookrightarrow \mathbf{false}$
VR	VARIABLE REPLACEMENT	$var_1 \neq var_2, \tau(var_1) = \tau(var_2) \vdash var_1 \hookrightarrow var_2$
		$\tau(var) = \tau(e.fd) \vdash var \hookrightarrow e.fd$
		$\tau(var) = \tau(e.md\mathbf{()}) \vdash var \hookrightarrow e.md\mathbf{()}$
FR	FIELD REPLACEMENT	$fd_1 \neq fd_2, \tau(e.fd_1) = \tau(e.fd_2) \vdash e.fd_1 \hookrightarrow e.fd_2$
		$\tau(e.fd) = \tau(var) \vdash e.fd \hookrightarrow var$
		$\tau(e.fd) = \tau(e.md\mathbf{()}) \vdash e.fd \hookrightarrow e.md\mathbf{()}$
		$\tau(e_2) = t, \tau(md) = t_r\ md(t) \vdash e_1.fd{=}e_2 \hookrightarrow e_1.md(e_2)$
MR	METHOD REPLACEMENT	$md \neq md', \tau(md) = \tau(md') \vdash e.md(e_1, \ldots, e_n) \hookrightarrow e.md'(e_1, \ldots, e_n)$
		$e_i' \in \{e_1, \ldots, e_n\} \cup \{var \mid \exists e_i.\tau(var) = \tau(e_i)\} \cup \{\mathbf{this}.fd \mid \exists e_i.\tau(\mathbf{this}.fd) = \tau(e_i)\} \cup \{\mathbf{0}, \mathbf{false}, \mathbf{null}\}$
		$\vdash e.md(e_1, \ldots, e_n) \hookrightarrow e.md(e_1', \ldots, e_m')$
		$\tau(e.md(e_1, \ldots, e_n)) = \tau(var) \vdash e.md(e_1, \ldots, e_n) \hookrightarrow var$
		$\tau(e.md(e_1, \ldots, e_n)) = \tau(e.fd) \vdash e.md(e_1, \ldots, e_n) \hookrightarrow e.fd$
TR	TYPE REPLACEMENT	$t_1 \preceq t_2 \vdash t_1\ e \hookrightarrow t_2\ e$
FG	FIELD GUARD	$t\ md(\ldots)\{\ldots e.fd \ldots\}, \mathrm{defVal}(t) = v \vdash e.fd \hookrightarrow (e == \mathbf{null}\ ?\ \mathbf{return}\ v : e.fd)$
		$t\ md(\ldots)\{\ldots e.fd \ldots\}, \tau(var) = t \vdash e.fd \hookrightarrow (e == \mathbf{null}\ ?\ \mathbf{return}\ var : e.fd)$
		$t\ md(\ldots)\{\ldots e.fd_1 \ldots\}, \tau(\mathbf{this}.fd_2) = t \vdash e.fd_1 \hookrightarrow (e == \mathbf{null}\ ?\ \mathbf{return\ this}.fd_2 : e.fd_1)$
		$\tau(e.fd) = t, \mathrm{defVal}(t) = v \vdash e.fd \hookrightarrow (e == \mathbf{null}\ ?\ v : e.fd)$
		$\tau(e.fd) = \tau(var) \vdash e.fd \hookrightarrow (e == \mathbf{null}\ ?\ var : e.fd)$
		$\tau(e.fd_1) = \tau(\mathbf{this}.fd_2) \vdash e.fd_1 \hookrightarrow (e == \mathbf{null}\ ?\ \mathbf{this}.fd_2 : e.fd_1)$
MG	METHOD GUARD	$\tau(e.md(e_1, \ldots, e_n)) = t, \mathrm{defVal}(t) = v \vdash e.md(e_1, \ldots, e_n) \hookrightarrow (e == \mathbf{null}\ ?\ \mathbf{return}\ v : e.md(e_1, \ldots, e_n))$
		$\tau(e.md(e_1, \ldots, e_n)) = \tau(var) \vdash e.md(e_1, \ldots, e_n) \hookrightarrow (e == \mathbf{null}\ ?\ \mathbf{return}\ var : e.md(e_1, \ldots, e_n))$
		$\tau(e.md(e_1, \ldots, e_n)) = \tau(\mathbf{this}.fd) \vdash e.md(e_1, \ldots, e_n) \hookrightarrow (e == \mathbf{null}\ ?\ \mathbf{return\ this}.fd : e.md(e_1, \ldots, e_n))$
		$\tau(e.md(e_1, \ldots, e_n)) = t, \mathrm{defVal}(t) = v \vdash e.md(e_1, \ldots, e_n) \hookrightarrow (e == \mathbf{null}\ ?\ v : e.md(e_1, \ldots, e_n))$
		$\tau(e.md(e_1, \ldots, e_n)) = \tau(var) \vdash e.md(e_1, \ldots, e_n) \hookrightarrow (e == \mathbf{null}\ ?\ var : e.md(e_1, \ldots, e_n))$
		$\tau(e.md(e_1, \ldots, e_n)) = \tau(\mathbf{this}.fd) \vdash e.md(e_1, \ldots, e_n) \hookrightarrow (e == \mathbf{null}\ ?\ \mathbf{this}.fd : e.md(e_1, \ldots, e_n))$
PC	PRE/POST- CONDITION	$e_1', \ldots, e_m' \in \{e_i \mid t_i \preceq \mathbf{Object} \wedge 0 \leq i \leq n\}, \mathrm{defVal}(t) = v$
		$\vdash t\ md(t_1\ e_1, \ldots, t_n\ e_n)\{e\} \hookrightarrow t\ md(t_1\ e_1, \ldots, t_n\ e_n)\{(e_1' == \mathbf{null} \vert\vert \ldots \vert\vert e_m' == \mathbf{null})\ ?\ \mathbf{return}\ v : e\}$
		$t\ md(\ldots)\{\ldots e.md(e_1, \ldots, e_n) \ldots\}, \tau(e.md(e_1, \ldots, e_n)) \preceq \mathbf{Object}, \mathrm{defVal}(t) = v, \tau(var) = t, \tau(\mathbf{this}.fd) = t, e' = \{v, var, \mathbf{this}.fd\}$
		$\vdash e.md(e_1, \ldots, e_n) \hookrightarrow (e.md(e_1, \ldots, e_n) == \mathbf{null}\ ?\ \mathbf{return}\ e' : e.md(e_1, \ldots, e_n))$

together with their frequency (i.e., the ratio of patches that each mutator occur), that we afforded to implement at the level of bytecode. Interestingly, the data in the table is consistent with what we observed when we actually fixed Defects4J bugs. In particular, the two least frequent mutators (as per Table 3) were unable to produce any plausible patch. We next discuss design challenges for each augmented mutator in PraPR:

3.2.1 Expression Replacement. This set of mutators mutate the commonly used variables, fields, methods, and types into other type-compatible ones. Mutator **VR** replaces the definition or use of a variable with the definition or use of another *visible* variable, field, or method with the same (return) type. Obtaining the set of visible variables at the mutation point is the most challenging part of implementing this mutator. We compute the set of visible variables at each point of the method under mutation using a simple dataflow analysis [60], before doing the actual mutation. Mutator **FR** mutates all field access instructions, namely `GETFIELD`, `PUTFIELD`, `GETSTATIC`, and `PUTSTATIC`. Upon visiting a field access instruction, the mutator loads the owner class of the field to extract all the information about its fields. The mutator then selects a different visible field (e.g., `public` fields), local variable, or method invocation, whose (return) type is compatible with that of the current field. It is worth noting that the newly selected field/method should be `static` if and only if the current field is `static`. Finally, the field access instruction is mutated to access the new element. Mutator **MR** aims to mutate all kinds of method invocation instructions (`static` and `virtual`). The operational details of this mutator is similar to that of **FR**, i.e., replacing a method invocation with another method invocation or variable/field access. Note that when mutating it to another method invocation, the mutator selects another method with a different name but with the same method descriptor (i.e., the same parameter and return types), or another method with the same name and compatible return type but with different parameter types (i.e., another overload of the callee). Note that replacing a method invocation with another overload can be non-trivial – we take advantage of the utility library shipped with ASM bytecode manipulation framework [2] to create temporary local variables so as to store the old argument values, and then use a variant of Levenshtein's edit distance algorithm [86] to find the minimal set of operations needed for reordering these local variables or using some other values (such as the default value corresponding to the type of a given parameter, or a visible local variable/field of the appropriate type) in order to prepare the stack before calling newly selected method overload. Finally, mutator **TR** aims to replace one type with another compatible one. Note that, for performance reasons, we only consider type widening in our implementation (via replacing a type with its immediate supertype) and apply the mutation only to `catch(T e)` blocks, because it usually does not make much sense in other contexts.

3.2.2 Conditional Insertion. The mutator **FG** mutates field dereference sites so as to inject code checking if the base expression is **null** at a given site. If it is non-**null** the injected code does nothing, otherwise it does either of the following: (1) returns the default value corresponding to the return type of the mutated method; (2) returns a local variable visible at the mutation point whose type is compatible with the return type of the mutated method; (3) returns a field whose type is compatible with the return type of the mutated method; (4) uses the default value corresponding to the type of the field being dereferenced instead of the field dereference expression; (5) uses a local variable visible at the mutation point whose type is compatible with that of the field being dereferenced; (6) uses a field whose type is compatible with that of the field being dereferenced.

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

The mutator **MG** targets virtual method invocation instructions. As the name suggests, the mutator **PC** is intended to add nullness checks for (1) the object-typed parameters and (2) what the method returns, provided that it is a subtype of **Object**, to avoid NullPointerExceptions. Note that although the mutators look trivial, they can be challenging to implement to support the full set of JVM insturctions/data types. For example, the set of JVM instructions shown in the side-figure illustrate the general form of the checking code injected by **MG** before an `INVOKEVIRTUAL` instruction, where m is the number of arguments of the callee, n is the index of a visible local variable to be used instead of the method call, while x, depending on the type of the parameters of the callee, could be `I` (`int`), `L` (`long`), and so on. 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 temporaries (using the leading group of x`STORE`s). Then, we check if the receiver object is `null` (please note that we duplicate the reference to the receiver object since instruction `IFNONNULL` consumes an object reference from the top of stack): if it is `null`, we pop the remaining copy of the receiver object off top of the stack, load the intended local n, and continue normal execution by jumping to label `escape`; otherwise, we push the arguments back to stack and invoke the target method.

4 EXPERIMENTAL SETUP

Our study investigates the following five research questions:

- **RQ1** How does PraPR perform in terms of effectiveness on automatically fixing real bugs?
- **RQ2** How does PraPR perform in terms of efficiency?
- **RQ3** How does PraPR compare with the state-of-art?

Table 4: Defects4J V1.4.0 programs

Sub.	Name	#Bugs	#Tests	LoC
Chart	JFreeChart	26	2,205	96K
Time	Joda-Time	27	4,130	28K
Mockito	Mockito framework	38	1,366	23K
Lang	Apache commons-lang	65	2,245	22K
Math	Apache commons-math	106	3,602	85K
Closure	Google Closure compiler	133	7,927	90K
Cli	Apache commons-cli	24	409	4K
Codec	Apache commons-codec	22	883	10K
Csv	Apache commons-csv	12	319	2K
JXPath	Apache commons-jxpath	14	411	21K
Gson	Google GSON	16	N/A	12K
Guava	Google Guava	9	1,701,947	420K
Core	Jackson JSON processor	13	867	31K
Databind	Jackson data bindings	39	1,742	71K
Xml	Jackson XML extensions	5	177	6K
Jsoup	Jsoup HTML parser	63	681	14K
Total		587	26,964	503K

- **RQ4** How do PraPR and recent APR techniques perform on additional bugs?
- **RQ5** How does PraPR perform on fixing real bugs from other JVM languages besides Java?

Subjects We conduct our experiments on Defects4J V1.4.0 [27, 31, 72], a collection of 16 real-world Java programs from GitHub with known, reproducible real bugs that subsumes all the bugs in Defects4J V1.2.0 [38]. These programs are real-world projects developed over an extended period of time, so they contain a variety of programming idioms and are a good representative of those programs found randomly in the wild. Thus, Defects4J programs are suitable for evaluating the effectiveness of candidate program repair techniques. Shown in Table 4, Column "#Bugs" presents the number of bugs for each program, while Columns "#Tests" and "LoC" present the number of tests (i.e., JUnit test methods) and the lines of code for the `HEAD` buggy version of each program. The first half of the table lists the projects (on or before Defects4J V1.2.0) that are already widely studied in prior APR research [18, 37, 42, 55, 71, 83, 85, 91] and also used in our **RQ1**-**RQ3**, while the second half of the table lists the projects that have not been used before and are used to answer **RQ4**. The two highlighted rows belong to the projects excluded due to build/testing framework incompatibility issues with PraPR.

Due to its minimalist syntax, and having a more sophisticated type system, Kotlin has gained popularity in recent years [34]. Kotlin has become the official "first-class" language for Android at Google I/O 2018 (in addition to Java) [20]; since then, 95% of developers show interest in using Kotlin for Android development and the number of Play Store apps using Kotlin grew 6X [77], including Uber, Square, Coursera, and Twitter apps. In addition, according to a recent Stack Overflow survey, Kotlin is the 2nd loved/wanted language (above Python) [5]. Therefore, in **RQ5**, we investigate bug fixing for Kotlin-based systems. More specifically, we applied PraPR on all the Kotlin bugs from a recent bug dataset Defexts [16]. Note that we were only able to run PraPR on 118 out of 225 Defexts Kotlin bugs, e.g., due to testing framework incompatibility.

Implementation PraPR has been implemented as a full-fledged program repair tool for JVM bytecode (publicly available on Maven Central Repo and our project website [29]). Currently it supports Java and Kotlin projects under different popular build systems (i.e., Maven [26] and Gradle [7]), and testing frameworks (i.e., JUnit [8], TestNG [3], and Spek [9] with JUnit runner). Given any such program with at least one failed test, PraPR can be applied using a

Table 5: Overall PraPR repair results

	Original Mutators		All Mutators			Original Mutators		All Mutators	
BugID	1-T(s)	#P	1-T(s)	#P	BugID	1-T(s)	#P	1-T(s)	#P
Chart-1	74	703	199	2624	Closure-130	987	9772	3782	34380
Chart-3	44	307	65	801	Closure-133	409	3240	1338	12732
Chart-4	76	835	158	2772	Lang-6	51	92	84	207
Chart-5	35	103	38	244	Lang-7	40	368	65	725
Chart-7	38	267	55	1039	Lang-10	60	416	127	919
Chart-8	38	122	52	403	Lang-22	83	78	170	177
Chart-11	34	52	36	106	Lang-25	20	3	21	18
Chart-12	50	440	76	1517	Lang-26	27	403	52	1066
Chart-13	43	571	66	2308	Lang-27	27	338	47	657
Chart-15	122	1774	237	6481	Lang-31	21	43	25	91
Chart-20	33	48	35	205	Lang-33	20	17	20	20
Chart-24	31	23	33	96	Lang-39	51	164	198	687
Chart-25	247	5497	745	19275	Lang-43	3046	66	11952	173
Chart-26	191	2658	449	9481	Lang-44	29	106	35	201
Closure-1	1147	6662	4117	22352	Lang-51	30	123	31	205
Closure-2	857	8893	3037	31634	Lang-57	24	4	24	10
Closure-3	1221	11358	4610	39365	Lang-58	28	177	40	372
Closure-5	884	8731	3300	31264	Lang-59	25	35	27	113
Closure-7	409	3036	1271	12538	Lang-60	31	125	45	436
Closure-8	731	6832	2845	24838	Lang-61	34	89	43	342
Closure-10	692	7481	2624	25929	Lang-63	67	322	126	1039
Closure-11	1421	11825	4774	42402	Math-2	562	332	581	1325
Closure-12	1090	11027	4203	38084	Math-5	1473	48	1493	201
Closure-13	1787	19832	6644	66760	Math-6	1443	116	1449	317
Closure-14	306	1962	799	6844	Math-7	1750	2454	2767	11117
Closure-15	981	9662	3759	33480	Math-8	1504	266	1545	1086
Closure-17	1187	12358	4529	44261	Math-18	894	3288	1410	12466
Closure-18	1071	10926	3820	36773	Math-20	1095	3189	1671	11645
Closure-21	754	7757	2956	27366	Math-28	784	1101	976	3364
Closure-22	748	7715	2949	27247	Math-29	849	419	1166	1601
Closure-29	969	8184	3805	28404	Math-32	943	3510	1508	17591
Closure-30	971	8684	3528	30053	Math-33	788	1179	861	3712
Closure-31	824	7487	2545	23931	Math-34	700	63	705	145
Closure-33	1303	13849	5065	49455	Math-39	177	1038	365	4171
Closure-35	1221	13349	4789	47397	Math-40	258	432	290	1661
Closure-36	2073	24838	7864	82595	Math-42	298	1069	403	3283
Closure-38	315	2636	768	8139	Math-49	252	351	270	1222
Closure-40	838	7954	3069	27621	Math-50	252	238	260	970
Closure-42	330	2923	1135	11251	Math-57	216	135	238	373
Closure-45	806	8615	3383	30263	Math-58	551	1486	1693	6276
Closure-46	284	2191	1048	8916	Math-59	175	642	231	1739
Closure-48	1095	11832	4310	42152	Math-60	74	540	99	1919
Closure-50	662	6026	2545	21198	Math-62	61	427	84	2310
Closure-59	1876	21648	6531	68137	Math-63	45	44	46	76
Closure-62	138	123	140	346	Math-64	134	929	322	4690
Closure-63	137	123	145	346	Math-65	89	979	150	4346
Closure-64	1208	14017	4014	44167	Math-70	33	61	35	189
Closure-66	586	6194	1881	21424	Math-71	287	649	766	2852
Closure-68	372	2606	1078	10527	Math-73	30	239	44	1187
Closure-70	921	8060	3217	27873	Math-74	489	1925	1535	8135
Closure-72	707	8408	2608	28075	Math-75	31	145	44	381
Closure-73	274	2392	676	7181	Math-78	55	421	129	2279
Closure-76	674	6992	2691	23868	Math-80	248	1922	919	10001
Closure-81	258	2462	962	9425	Math-81	157	1498	679	7647
Closure-84	258	2514	959	9637	Math-82	44	665	69	2051
Closure-86	345	1615	899	5255	Math-84	50	190	82	574
Closure-92	496	5704	1922	19029	Math-85	95	372	250	1195
Closure-93	493	5704	1965	19028	Math-88	47	775	82	2356
Closure-101	1020	12569	4059	39882	Math-95	3571	287	13320	928
Closure-107	1166	11714	4665	39195	Math-101	20	120	30	360
Closure-108	1036	8775	5299	33521	Math-104	56	212	141	823
Closure-109	568	3158	1458	12451	Mockito-5	38	97	57	184
Closure-111	651	4670	1792	17601	Mockito-8	35	119	41	246
Closure-115	1453	9496	5081	32442	Mockito-15	65	808	88	1885
Closure-119	787	7424	2901	27729	Mockito-28	91	1069	134	2525
Closure-120	963	9589	3624	33008	Mockito-29	78	1210	112	2716
Closure-121	985	9589	3669	33008	Mockito-38	29	115	34	258
Closure-122	439	2907	1437	11076	Time-4	59	768	84	1812
Closure-123	434	4097	1350	13491	Time-11	81	1327	102	2908
Closure-124	742	6586	2713	23706	Time-14	60	504	70	1019
Closure-125	1463	14754	5652	52866	Time-17	114	2100	184	6324
Closure-126	780	6567	2814	23569	Time-19	121	1422	152	3302
Closure-127	1067	7363	3725	25626	Time-20	144	2582	225	6996
Closure-129	1551	16465	5413	54648	Time-24	109	1560	166	3395
Σ #Plau. Original Mutators			106		**Σ #Gen. Original Mutators**			17	
Σ #Plau. All Mutators			148		**Σ #Gen. All Mutators**			43	

single command, "mvn org.mudebug:prapr-plugin:prapr". During the repair process, PraPR uses the ASM bytecode manipulation framework [2] and Java Agent [4] to collect coverage information (used for Ochiai-based fault localization [10]) and perform patch generation. We have built PraPR via extending the mutators employed by the state-of-the-art bytecode-level mutation engine PIT [19], since PIT is the most robust and widely used mutation testing tool both in academia and industry [19, 41]. All our experimentation is done on a Dell workstation with Intel Xeon CPU E5-2697 v4@2.30GHz and 98GB RAM, running Ubuntu 16.04.4 LTS and Oracle Java 64-Bit Server version 1.7.0_80. PraPR supports multi-thread patch validation, and we run PraPR using both 1 and 4 threads *exhaustively on all candidate patches* to precisely measure its cost.

5 RESULT ANALYSIS

5.1 RQ1: PraPR Effectiveness

Table 5 presents the main repair results for all the bugs from Defects4J V1.2.0 for which PraPR can generate plausible fixes. In the table, Column "Original Mutators" presents the repair results using only the original PIT mutators for each bug, including the total repair time (using single thread) for validating all patches (Column "1-T(s)") and the number of all validated patches (Column "#P"). The cells highlighted with light gray denote plausible fixes, while those highlighted with dark gray correspond to genuine fixes. Note that we only present the number of validated patches (i.e., the patches passing the check at Line-7 in Algorithm 1), since the other patches cannot pass all the failed tests and do not need to be validated. Similarly, Column "All Mutators" presents the corresponding repair results using all the mutators (i.e., further including our augmented mutators). Finally, the last two rows show the number of plausible/genuine fixes produced by the two classes of mutators.

According to the table, surprisingly, even the original PIT mutators can generate plausible fixes for 106 bugs and genuine fixes for 17 bugs from Defects4J V1.2.0, comparable to the most recent work CapGen [85] that produces genuine fixes for 22 bugs. On the contrary, prior jMutRepair work [55] showed that mutation testing can find only 17 plausible and 4 genuine fixes for the same version of Defects4J. One potential reason is that the prior work was based on source-code mutation which incurs expensive recompilation and loading for each mutant, and thus does not scale to large programs like Closure. Another reason is that the prior work used only 3 mutators (we found that had jMutRepair been able to scale to all the Defects4J programs, it would generate up to 7 genuine fixes). *To our knowledge, this is the first study demonstrating that plain mutation testing can be comparable to state-of-the-art APR for fixing real bugs.*

Furthermore, all PraPR mutators (including the original PIT mutators and our augmented mutators) can produce plausible and genuine fixes for 148 and 43 bugs, respectively. To our knowledge, this is the largest number of bugs reported as fixed for Defects4J to date. The key reason for this result is PraPR's capability in exploring such a large number of potential patches within a short time due to the bytecode-level patch generation/validation and our execution optimizations. For example, even for the largest Closure, PraPR with 1 thread is still able to validate approximately 10 patches per second. *This demonstrates the effectiveness of PraPR and shows the importance of fast (and exhaustive) patch generation and validation for automatic program repair: faster mutation allows us to apply more mutators and hence exploring a larger portion of the search space.*

Next we show some example genuine fixes produced by PraPR to qualitatively illustrate the effectiveness of PraPR. As shown in Figure 4, PraPR using the mutator **CO** is able to produce a genuine fix identical to the developer patches. Note that those patches are as expected for they directly fall into the capability of the employed

```
// Developer and PraPR patches
        } else if (offsetLocal > 0) {
     +++} else if (offsetLocal >= 0) {
```

Figure 4: Time-19 patches

```
// Developer patch
@Override
public JSType getLeastSupertype(JSType that) {
        if (!that.isRecordType()) {
            return super.getLeastSupertype(that);}...}
// PraPR patch
@Override
public JSType getLeastSupertype(JSType that) {
        if (!that.isRecordType()) {
    +++if (!false) {
            return super.getLeastSupertype(that); }...}
```

Figure 5: Closure-46 patches

Table 6: Average PraPR time cost with single thread

Sub.	Original Mutators				All Mutators			
	#P	Avg(s)	Min(s)	Max(s)	#P	Avg(s)	Min(s)	Max(s)
Chart	619.9	59.4	31	247	2158.3	112.1	33	745
Closure	6876	739	128	128	23877.7	2659.2	140	11080
Lang	147.5	80.3	16	3046	356	236.8	16	11952
Math	550.4	554.4	15	6997	2258.9	1143.1	18	13320
Mockito	728.7	74.1	14	204	1702.9	104.8	14	331
Time	781	74	32	155	1835.1	99.1	33	225

mutators. Interestingly, we also observe that in a couple of cases PraPR is able to suggest more complex genuine fixes that require simple semantic reasoning. Figure 5 presents both the developer and PraPR patches for Closure-46. According to the figure, the developer patch removes an overriding method from a subclass, which is not directly handled using PraPR mutators, but the PraPR patch, generated via the mutator **CO**, forces the overriding method to always directly invoke the corresponding overridden method, which is semantically equivalent to removing the overriding method.

5.2 RQ2: PraPR Efficiency

We present the efficiency information of PraPR on all the Defects4J bugs using the default, single thread, settings in Table 6. In the table, Column "Original Mutators" presents the average number of all validated patches (Column "#P"), as well as the average/minimum/-maximum time cost with 1 thread (Column "Avg"/"Min"/"Max") for all the bugs of each subject system using the original PIT mutators. Similarly, Column "All Mutators" presents the information when using all PraPR mutators. We observe that PraPR is remarkably efficient even using only a single thread, e.g., it costs at most 3.7 hours among all studied bugs (i.e., Math-95 because the majority of the mutations modify the program control-flow in such a way that resulting in a huge number of infinite/costly loops). Furthermore, we have also run PraPR on all the studied bugs with 4 threads and observed up to 2.1X performance gain.

Note that besides the *machine* time, the repair efficiency also involves the *manual* efforts in inspecting plausible patches. Thus, we further present the ranking of genuine patches within all validated/plausible patches to truly understand PraPR efficiency. Table 7 presents the ranking of the genuine fixes among all validated patches and all plausible fixes. Columns "Rank Orig." and "Rank All' present the rank of the first genuine fix among the validated patches when using the original PIT mutators and all PraPR mutators, respectively. The rank of the first genuine fix among all plausible fixes is shown in parentheses. Note that for the patches with tied

Table 7: Rank of PraPR genuine fixes

BugID	Rank Orig.		Rank All		BugID	Rank Orig.		Rank All	
Chart-1	54	(1)	205	(1)	Lang-10	247	(1)	300	(2)
Chart-8	N/A	(N/A)	95	(2)	Lang-26	N/A	(N/A)	967	(1)
Chart-11	N/A	(N/A)	106	(1)	Lang-33	N/A	(N/A)	20	(1)
Chart-12	N/A	(N/A)	118	(2)	Lang-57	N/A	(N/A)	10	(3)
Chart-20	N/A	(N/A)	45	(1)	Lang-59	N/A	(N/A)	93	(2)
Chart-24	N/A	(N/A)	77	(1)	Math-5	N/A	(N/A)	53	(1)
Chart-26	N/A	(N/A)	1111	(17)	Math-33	N/A	(N/A)	602	(1)
Closure-10	N/A	(N/A)	1677	(1)	Math-34	N/A	(N/A)	22	(1)
Closure-11	2006	(1)	7230	(1)	Math-50	21	(5)	113	(40)
Closure-14	N/A	(N/A)	1	(1)	Math-58	N/A	(N/A)	401	(2)
Closure-18	6773	(1)	22034	(1)	Math-59	N/A	(N/A)	29	(1)
Closure-31	3851	(2)	17383	(6)	Math-70	N/A	(N/A)	17	(1)
Closure-46	21	(1)	61	(1)	Math-75	N/A	(N/A)	24	(1)
Closure-62	21	(1)	55	(1)	Math-82	270	(5)	754	(9)
Closure-63	21	(1)	55	(1)	Math-85	204	(4)	582	(4)
Closure-70	229	(1)	827	(1)	Mockito-5	N/A	(N/A)	74	(31)
Closure-73	34	(1)	71	(1)	Mockito-29	N/A	(N/A)	72	(2)
Closure-86	1	(1)	1	(1)	Mockito-38	N/A	(N/A)	11	(2)
Closure-92	N/A	(N/A)	174	(1)	Time-4	N/A	(N/A)	315	(5)
Closure-93	N/A	(N/A)	174	(1)	Time-11	24	(1)	70	(1)
Closure-126	12	(2)	55	(5)	Time-19	870	(1)	1939	(2)
Lang-6	N/A	(N/A)	160	(1)					
Avg. Total Rank Original			862.3		**Avg. Plau. Rank Original**			(1.8)	
Avg. Total Rank All			1353.1		**Avg. Plau. Rank All**			(3.8)	

suspiciousness, PraPR favors the patches generated by mutators with smaller ratios of plausible to validated patches since the mutators with larger ratios tend to be resilient to the corresponding test suite. If the tie remains, PraPR uses the *worst* ranking for all the tied patches. From the table, we can observe that the genuine fixes are ranked high among validated and plausible patches when using both original and all mutators. For example, surprisingly, among the plausible fixes, the genuine fixes are ranked only 1.8th using original mutators and ranked only 3.8th using all mutators, demonstrating that few manual efforts will be involved when inspecting the repair results of PraPR. We found that one reason is the small number of plausible fixes even when using all the mutators since the test suites of the Defects4J subjects are strong enough to falsify the vast majority of non-genuine patches. To illustrate, the number of plausible patches is usually smaller for Closure (which has the most candidate patches) due to the stronger test suite of Closure, e.g., Closure has 300+ contributors and the largest test suite among the subjects studied in this section.

5.3 RQ3: Comparison with the State-of-Art

Effectiveness To investigate this question, we compare PraPR with the state-of-the-art APR techniques that have been evaluated on Defects4J (V1.2.0) before, including SimFix [37], CapGen [85], JAID [18], SketchFix [33], ELIXIR [71], ssFix [89], ACS [91], HD-Repair [42], xPAR [42] (a reimplementation of PAR [40]), NOPOL [92], jGenProg [54] (a reimplementation of GenProg [43] for Java), jMutRepair [55] (a reimplementation of source-level mutation-based repair [22] for Java), and jKali [55] (a reimplementation of Kali [70] for Java). Following [18, 85, 91], except for SimFix, CapGen, SketchFix, and JAID, we obtained the repair results for prior APR techniques from their original papers. In Table 8, Column "Tech." lists all the compared techniques. Column "All Positions" presents the number of genuine and non-genuine plausible fixes found when inspecting all the generated plausible fixes for each bug. Similarly, the columns "Top-10 Positions" and "Top-1 Position" present the number of genuine and non-genuine plausible fixes found when inspecting Top-10 and Top-1 plausible fixes, resp. Except for the case of Top-1, we can observe that PraPR can fix the most number of bugs compared to all the studied techniques. Figure 6 further presents the distribution of the bugs that can be successfully fixed by PraPR and other recent APR techniques. We can observe that PraPR can fix 10 bugs that have not been fixed by any of the aforementioned techniques. Also, *the studied tools are complementary, i.e., putting all the tools together, we can fix 90+ bugs from Defects4J.*

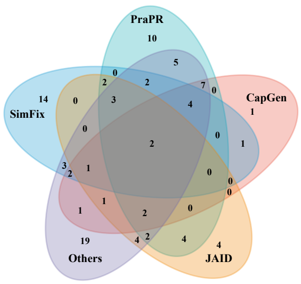

Figure 6: Fixed bug dist.

Another interesting observation worth discussion is that PraPR produces only non-genuine plausible fixes for more bugs than the other techniques. We found a couple of reasons. First, our main goal in this work is to propose a baseline repair technique that does not require any mining/learning information [85, 91] for both practical application and experimental evaluation; also, recently various patch correctness checking techniques [32, 81, 90] have been proposed, and can be directly applied to further improve the PraPR patch validation process. We have already explored one of these possibilities. Specifically, we used the mined mutator frequency presented in Table 3 to break the ties after sorting the plausible fixes according to their suspiciousness (more frequent mutators get higher priorities). The results in Row "PraPR*" shows that such simple mining information can already rank 30 genuine fixes in Top-1, comparable to the state-of-art. Second, PraPR is able to explore a large search space during a short time due to the lightweight bytecode-level patch generation, while existing techniques usually have to terminate early due to time constraints. Third, prior work using intensive mining/learning information can suffer from the *overfitting* problem: the original CapGen was not evaluated on Closure and Mockito, while SimFix was not evaluated on Mockito; working together with their authors, we were able to run such experiments, but observed a much lower precision than their original subjects (shown in the last two rows of Table 8) – CapGen produces 1092 plausible fixes in total for 10/14 bugs from Mockito/Closure, and SimFix fails to locate any suitable code snippets for Mockito.

Lastly, in this work, we also manually inspected all the 105 bugs for which PraPR is only able to produce non-genuine plausible fixes. Surprisingly, we observe that even the non-genuine plausible fixes for such bugs can still provide useful debugging hints. For example, the plausible fixes ranked at the 1st position for 50 bugs share the same methods with the actual developer patches, i.e., for 48% cases the non-genuine plausible fixes can directly point out the patch locations for manual debugging while even state-of-the-art spectrum-based (e.g., Ochiai) and mutation-based (e.g., MUSE [59] and Metallaxis [64]) fault localization can localize at most 21% of the same bugs within Top-1, indicating a promising future for *using APR patches to boost fault localization (in contrast to the current paradigm of using fault localization to boost APR).*

Efficiency We further executed the publicly available recent APR tools (i.e., SimFix, CapGen, JAID, and SketchFix) on the same platform with single-thread PraPR for a fair efficiency comparison. Table 9 shows the average time data on the bugs that the compared tools can correctly fix. Columns 2 to 4 present the following information for SimFix: the number of patches validated, the average number of patches validated per time unit (s), and the speedup gained by PraPR in terms of the average number of patches per second. The other columns show the corresponding information for CapGen, JAID, and SketchFix. Note that the gray row marks that we were unable to reproduce any patch for Lang when using SketchFix. According to Table 9, JAID and SketchFix are usually faster than CapGen and SimFix on the same subject, due to their compilation optimization strategies, e.g., meta-program encoding and sketching; PraPR is almost at least an order of magnitude faster compared with all tools on all subjects except some minor cases, e.g., when compared with JAID on the smallest subject Lang. The reason is that there is only one bug that both PraPR and JAID can fix (i.e., Lang-33), and PraPR fixes it using 20 patches within 20 seconds (a similar speed with JAID) since the startup cost for such a small number of patches makes PraPR's per-patch time non-trivial. Actually, if we average over all fixable bugs across all subjects, *PraPR is over 10X faster than all the compared techniques* (including JAID). We attribute this substantial speedup to the fact that PraPR operates completely at the bytecode level; it does not need any recompilation and loading from disk for any patch. For manual-effort efficiency, we also found prior tools require various configurations to get started, and are usually not designed to be used for arbitrary Java projects. On the contrary, PraPR offers a 1-click APR tool publicly available on Maven Central Repo and applicable to arbitrary Java project under Maven/Gradle build systems (not just Defects4J) and even projects in other JVM languages.

Table 8: Comparison with state-of-the-art techniques

Tech.	All Positions		Top-10 Positions		Top-1 Position	
	Gen.	Non-gen.	Gen.	Non-gen.	Gen.	Non-gen.
PraPR	43	105	40	108	26	122
PraPR*	43	105	39	109	30	118
SimFix	N/A	N/A	N/A	N/A	34	22
CapGen	22	3	22	3	21	4
JAID	25	6	15	16	9	22
SketchFix	19	7	N/A	N/A	9	17
ELIXIR	N/A	N/A	N/A	N/A	26	15
ssFix	N/A	N/A	N/A	N/A	15	45
ACS	N/A	N/A	N/A	N/A	18	5
HD-Repair	16	N/A	N/A	N/A	10	N/A
xPAR	4	N/A	4	N/A	N/A	N/A
NOPOL	5	30	5	30	5	30
jGenProg	5	22	5	22	5	22
jMutRepair	4	13	4	13	4	13
jKali	1	21	1	21	1	21
SimFix $_{\text{Mockito}}$	0	0	0	0	0	0
CapGen $_{\text{Mockito,Closure}}$	0	24	0	24	0	24

Table 9: Time costs of recent APR tools

Sub.	SimFix			CapGen			JAID			SketchFix		
	#P	P/s	Gain	#P	P/s	Gain	#P	P/s	Gain	#P	P/s	Gain
Chart	1141.5	0.3	(27.5X)	254.8	0.4	(16.9X)	3561.8	1.3	(4X)	3186	0.7	(9.5X)
Closure	311.2	0.1	(51.5X)	N/A	N/A	(N/A)	7110.1	0.3	(23.8X)	903.3	0.07	(93.5X)
Lang	412.3	0.3	(3X)	807.4	0.3	(27.2X)	3602.4	1	(1.04X)	N/A	N/A	(N/A)
Math	360.2	0.3	(16.4X)	604.6	0.3	(19.5X)	8348	0.5	(19X)	1561.8	0.4	(20.8X)
Time	431	0.3	(N/A)	N/A	N/A	(N/A)	N/A	N/A	(N/A)	N/A	N/A	(N/A)

5.4 RQ4: APR Tools on Additional Bugs

To further reduce the threats to external validity, we have applied PraPR and the publicly available recent APR tools (i.e., JAID, SketchFix, SimFix and CapGen) on an additional 192 bugs from Defects4J V1.4.0 (§4). Unfortunately, we were unable to successfully apply the other studied APR tools in our first try. Thus, we actively worked with all the authors to address those issues. For the time being, we choose to report the results of experimenting with only SimFix and CapGen because (1) they are the most recent and effective tools,

Table 10: Recent APR tools on additional bugs

Sub.	PraPR			SimFix			CapGen		
	#Gen.(Top-1)	#Plau.(μ)	F/TO	#Gen.(Top-1)	#Plau.(μ)	F/TO	#Gen.(Top-1)	#Plau.(μ)	F/TO
Cli	3(1)	7(4.6)	0/0	0(0)	0(0)	0/1	0(0)	7(4.9)	0/0
Codec	1(1)	6(8.3)	0/0	0(0)	0(0)	0/0	1(1)	8(91.1)	0/0
Csv	1(1)	2(8)	0/0	0(0)	0(0)	0/0	0(0)	2(8)	0/0
JXPath	1(0)	4(10.5)	0/0	0(0)	0(0)	0/0	0(0)	5(304.8)	0/0
Core	0(0)	10(28.5)	0/0	0(0)	0(0)	0/13	0(0)	6(80.3)	0/0
Databind	4(2)	16(6.4)	0/0	0(0)	0(0)	0/32	0(0)	15(55.1)	1/1
Xml	0(0)	0(0)	0/0	0(0)	0(0)	0/2	0(0)	0(0)	0/0
Jsoup	2(2)	12(4.3)	0/0	0(0)	0(0)	0/4	1(0)	14(19.9)	0/0

Table 11: PraPR results on Kotlin projects

BugId	LoC	#P	Fixes	1-T(s)	Mutator
kog-1	3804	307	2(1)	18	**MR**
Simple-MsgPack-1	1565	1445	1(1)	104	**AO**
rapier-2	414	501	2(1)	82	**IC**,**CO**
jenjin-1	22261	1057	1(1)	44	**MR**
seven-wonders-1	10318	11	1(1)	5	**CO**
thrifty-3	7256	4148	14(14)	231	**TR**
thrifty-4	7956	2588	4(4)	226	**AP**
rimu-kt-1	2291	3076	1(1)	296	**CO**
patchtools-2	1171	2692	1(1)	616	**MG**
icfpc2016-2	6173	315	2(2)	18	**MC**
Kartvelang-1	1252	1130	5(1)	36	**MC**
lambda-1	1066	220	6(6)	74	**CO**
parallel-feature-selection-1	7371	560	10(10)	16	**CO**
UltimateTTT-1	2296	603	4(1)	153	**MR**

and (2) we received eager cooperation from the authors. Together with the authors, we were able to run SimFix and CapGen. It is worth noting that, we reported several bugs to the CapGen authors and also directly contributed to enable CapGen to work on more projects; we also managed to write our own code to produce all the information that SimFix needs for fixing arbitrary Java programs, which was confirmed by the authors of SimFix. Table 10 summarizes the results of our experiments. For each technique, Column "#Gen. (Top-1)" presents the number of bugs with genuine patches (with the number of bugs with genuine patches ranked at Top-1 inside parentheses), Column "#Plau.(μ)" represents the total number of bugs with plausible patches (with the average number of plausible patches for each bug inside parentheses), Column "F/TO" reports the number of times each tool crashed, and the number of times each tool has timed out within the allotted 5-hour limit.

According to the table, PraPR is able to generate genuine patches for 12 bugs that 7 appear in Top-1 positions. Meanwhile, CapGen produces genuine patches for only 2 bugs (1 within Top-1), while SimFix was unable to generate any plausible patch, despite the fact that it exhausted its search space for most cases and timed out in 52 bugs. We attribute the slight performance drop of PraPR (c.f. §5.1) to the fact that these bugs mostly need multiple edits to fix. The huge 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. We also observed that the studied tools are rather robust except for one case, where CapGen crashed due to a failure of the Understand tool [75] that CapGen uses for slicing. Another interesting finding is CapGen generates much more false positives than PraPR on this new dataset. *To our knowledge, this is the first study demonstrating recent advanced APR techniques may suffer from the overfitting problem in case of unexpected bugs, while a simplistic approach shows a decent level of consistency.*

5.5 RQ5: PraPR Repair for Real Kotlin Bugs

We applied PraPR to fix all the 225 Defexts Kotlin bugs, out of which 118 bugs are PraPR-compatible, i.e., exclusively using JUnit/TestNG tests or using Spek [9] tests with JUnit runners. These buggy projects range from 248 LoC to 170,789 LoC. Of the 118 bugs, 14 were correctly repaired by PraPR. Table 11 summarizes the data for the bugs with genuine patches. In this table, Column "BugId" presents the identifiers of the bugs as recorded inside Defexts database, Column "LoC" presents the project size, Column "#P" presents the total number of patches PraPR performed on the project, Column "Fixes" presents the number of plausible fixes PraPR generated alongside the rank of genuine patches among plausible fixes (in parentheses), Column "1-T (s)" presents PraPR's execution time with 1 thread, and Column "Mutator" presents the mutators which produced the genuine fix. *To our knowledge, this is the first repair study for Kotlin systems; the similar ratio of fixed bugs for Kotlin systems also reduces our threats to external validity.*

6 DISCUSSION

Limitation. Bytecode mutation clearly cannot fix all types of bugs. At the level of bytecode, we do not have access to lots of information (such as detailed typing and contextual information) useful for fixing bugs beyond simple mutations. Also, fixing complex bugs at the bytecode level can be challenging and tedious. Despite this fact, our experimental results demonstrate that the sheer speed of patch generation/validation and language agnosticism of bytecode-level APR can complement existing source-code level APR techniques.
Threats to internal validity. Understanding patch reports for some JVM-based languages might be challenging. We emphasize that based on our experience with PraPR, the PraPR patch reports for Java and all the Kotlin programs that we have experimented with, can easily be reconstructed with simple manual inspection. Note that PraPR also supports automatically decompiling bytecode patches via Eclipse Class Decompiler [6]. Furthermore, during the manual inspection for patch correctness, there might be mistakes in judging whether a particular patch is indeed a genuine fix. To minimize such mistakes, we have confined ourselves to syntactic equality and simple semantic equivalence. Furthermore, we also released all our patches in PraPR website.
Threats to external validity. Our claims about any of the studied APR techniques might be biased because of the limited number of benchmark programs that we have considered. To this end, we have tried our best to apply the studied techniques to a newer version of Defects4J that has not been studied for APR before, and have also applied PraPR on Defexts, a new Kotlin bug dataset.

7 CONCLUSION

We have implemented PraPR, the first practical 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 Defects4J bugs, significantly outperforming state-of-the-art Java repair techniques, while being over 10X faster; with no learning/search information, PraPR also avoids the overfitting problem of existing techniques on additional bugs from a newer version of Defects4J; finally, PraPR successfully fixed 14 of the 118 studied bugs for Kotlin systems.

ACKNOWLEDGMENTS

We thank Jiajun Jiang, Ming Wen, and Xia Li for their help, and ISSTA reviewers for the insightful comments. This work is partially supported by NSF grants CCF-1566589, CCF-1763906, and Amazon.

REFERENCES

[1] 2001. Siemens Suite (SIR). http://sir.unl.edu/. Accessed Jan-9-2019.
[2] 2011. ASM bytecode manipulation framework. http://asm.ow2.org/. Accessed Jan-9-2019.
[3] 2017. TestNG Documentation. https://testng.org/doc/index.html. Accessed Jan-8-2019.
[4] 2018. Java Agent. https://docs.oracle.com/javase/7/docs/api/java/lang/instrument/package-summary.html. Accessed Jan-9-2019.
[5] 2018. Stack Overflow Survey. https://insights.stackoverflow.com/survey/2018/. Accessed Jan-9-2019.
[6] 2019. Eclipse Class Decompiler . https://www.eclipse.org/community/eclipse_newsletter/2017/february/article8.php. Accessed Jan-18-2019.
[7] 2019. Gradle Build Tool. http://gradle.org/. Accessed Jan-20-2019.
[8] 2019. JUnit. http://junit.org/. Accessed Jan-20-2019.
[9] 2019. Spek Framework. https://spekframework.org/. Accessed Jan-24-2019.
[10] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2007. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007.* IEEE, 89–98.
[11] Paul Ammann and Jeff Offutt. 2008. *Introduction to Software Testing* (1 ed.). Cambridge University Press, New York, NY, USA.
[12] J. H. Andrews, L. C. Briand, and Y. Labiche. 2005. Is Mutation an Appropriate Tool for Testing Experiments?. In *ICSE (ICSE'05)*. 402–411.
[13] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. 2008. Using Static Analysis to Find Bugs. *IEEE Software* 25, 5 (Sept 2008), 22–29.
[14] Tien-Duy B Le, David Lo, Claire Le Goues, and Lars Grunske. 2016. A learning-to-rank based fault localization approach using likely invariants. In *ISSTA (ISSTA'16)*. ACM, 177–188.
[15] Earl T. Barr, Yuriy Brun, Premkumar T. Devanbu, Mark Harman, and Federica Sarro. 2014. The plastic surgery hypothesis. In *FSE (FSE'14)*. 306–317.
[16] Samuel Benton, Ali Ghanbari, and Lingming Zhang. 2019. Defexts: A Curated Dataset of Reproducible Real-World Bugs for Modern JVM Languages (demo). In *ICSE Demo*. to appear.
[17] CO Boulder. 2013. University of Cambridge Study: Failure to Adopt Reverse Debugging Costs Global Economy $41 Billion Annually. https://www.roguewave.com/company/news/2013/university-of-cambridge-reverse-debugging-study. Accessed: Jan. 8, 2019.
[18] Liushan Chen, Yu Pei, and Carlo A. Furia. 2017. Contract-based Program Repair Without the Contracts. In *ASE (ASE 2017)*. IEEE Press, Piscataway, NJ, USA, 637–647.
[19] Henry Coles. 2019. State of the art mutation testing system for the JVM. https://github.com/hcoles/pitest. Accessed Jan-19-2019.
[20] Stephanie Cuthbertson. 2018. Google I/O 2018. https://android-developers.googleblog.com/2018/05/google-io-2018-whats-new-in-android.html. Accessed Jan-19-2019.
[21] Valentin Dallmeier, Andreas Zeller, and Bertrand Meyer. 2009. Generating Fixes from Object Behavior Anomalies. In *ASE (ASE'09)*. IEEE Computer Society, Washington, DC, USA, 550–554.
[22] V. Debroy and W. E. Wong. 2010. Using Mutation to Automatically Suggest Fixes for Faulty Programs. In *ICST*. 65–74.
[23] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and Accurate Source Code Differencing. In *ASE (ASE'14)*. ACM, New York, NY, USA, 313–324.
[24] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. 1998. Classes and Mixins. In *POPL (POPL'98)*. 171–183.
[25] Apache Software Foundation. 2019. Groovy Programming Language. http://groovy-lang.org/. Accessed Jan-21-2019.
[26] Apache Software Foundation. 2019. Maven. https://maven.apache.org. Accessed Jan-9-2019.
[27] Gregory Gay. 2018. Detecting real faults in the Gson library through search-based unit test generation. In *International Symposium on Search Based Software Engineering*. Springer, 385–391.
[28] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2017. Automatic software repair: A survey. *IEEE TSE* (2017).
[29] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. PraPR Website. https://github.com/prapr/prapr. Accessed May-27-2019.
[30] Divya Gopinath, Muhammad Zubair Malik, and Sarfraz Khurshid. 2011. Specification-based Program Repair Using SAT. In *TACAS (TACAS'11/ETAPS'11)*. Springer-Verlag, Berlin, Heidelberg, 173–188.
[31] Greg4cr. 2018. Defects4J – version 1.4.0. https://github.com/Greg4cr/defects4j/tree/additional-faults-1.4. Accessed Jan-18-2019.
[32] Zhongxian Gu, Earl T. Barr, David J. Hamilton, and Zhendong Su. 2010. Has the Bug Really Been Fixed? *(ICSE'10)*. ACM, New York, NY, USA, 55–64.
[33] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. 2018. Towards practical program repair with on-demand candidate generation. In *ICSE (ICSE'18)*. 12–23.
[34] JetBrains. 2018. Kotlin Language Documentation. http://kotlinlang.org/. Accessed Jan-21-2019.
[35] T. Ji, L. Chen, X. Mao, and X. Yi. 2016. Automated Program Repair by Using Similar Code Containing Fix Ingredients. In *COMPSAC*, Vol. 1. 197–202.
[36] Yue Jia and Mark Harman. 2011. An analysis and survey of the development of mutation testing. *IEEE TSE* 37, 5 (2011), 649–678.
[37] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping program repair space with existing patches and similar code. In *ISSTA*. ACM, 298–309.
[38] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *ISSTA (ISSTA 2014)*. ACM, New York, NY, USA, 437–440.
[39] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are Mutants a Valid Substitute for Real Faults in Software Testing?. In *FSE (FSE 2014)*. 654–665.
[40] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *ICSE*. IEEE Press, 802–811.
[41] Thomas Laurent, Mike Papadakis, Marinos Kintis, Christopher Henard, Yves Le Traon, and Anthony Ventresque. 2017. Assessing and improving the mutation testing practice of PIT. In *ICST (ICST'17)*. IEEE, 430–435.
[42] Xuan Bach D Le, David Lo, and Claire Le Goues. 2016. History driven program repair. In *SANER*, Vol. 1. IEEE, 213–224.
[43] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE TSE* 38, 1 (2012), 54–72.
[44] Xiangyu Li, Marcelo d'Amorim, and Alessandro Orso. 2016. Iterative user-driven fault localization. In *Haifa Verification Conference*. Springer, 82–98.
[45] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. 2019. DeepFL: Integrating Multiple Fault Diagnosis Dimensions for Deep Fault Localization. In *ISSTA*. to appear.
[46] Xia Li and Lingming Zhang. 2017. Transforming Programs and Tests in Tandem for Fault Localization. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 92 (Oct. 2017), 30 pages.
[47] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2017. *The Java Virtual Machine Specification, Java SE 9 Edition* (1st ed.). Addison-Wesley Professional.
[48] Francesco Logozzo and Thomas Ball. 2012. Modular and Verified Automatic Program Repair. *SIGPLAN Not.* 47, 10 (Oct. 2012), 133–146.
[49] Fan Long and Martin Rinard. 2015. Staged program repair with condition synthesis. In *FSE (FSE'15)*. 166–178.
[50] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *POPL (POPL'16)*. 298–312.
[51] Fan Long, Stelios Sidiroglou-Douskos, and Martin C. Rinard. 2014. Automatic runtime error repair and containment via recovery shepherding. In *PLDI (PLDI'14)*. 227–238.
[52] Yiling Lou, Junjie Chen, Lingming Zhang, Dan Hao, and Lu Zhang. 2019. History-driven Build Failure Fixing: How Far Are We?. In *ISSTA*. to appear.
[53] Siqi Ma, David Lo, Teng Li, and Robert H Deng. 2016. Cdrep: Automatic repair of cryptographic misuses in android applications. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ACM, 711–722.
[54] Matias Martinez, Thomas Durieux, Jifeng Xuan, Romain Sommerard, and Martin Monperrus. 2015. Automatic repair of real bugs: An experience report on the defects4J dataset. *arXiv preprint arXiv:1505.07002* (2015).
[55] Matias Martinez and Martin Monperrus. 2016. ASTOR: A Program Repair Library for Java (Demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. 441–444.
[56] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: scalable multiline program patch synthesis via symbolic analysis. In *ICSE (ICSE'16)*. 691–701.
[57] Ben Mehne, Hiroaki Yoshida, Mukul R Prasad, Koushik Sen, Divya Gopinath, and Sarfraz Khurshid. 2018. Accelerating Search-Based Program Repair. In *ICST (ICST'18)*. 227–238.
[58] Martin Monperrus. 2018. Automatic Software Repair: A Bibliography. *ACM Comput. Surv.* 51, 1, Article 17 (Jan. 2018), 24 pages.
[59] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. 2014. Ask the mutants: Mutating faulty programs for fault localization. In *IEEE 7th ICST*. IEEE, 153–162.
[60] Steven S. Muchnick. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
[61] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: program repair via semantic analysis. In *ICSE (ICSE'13)*. 772–781.
[62] Martin Odersky. 2014. The Scala Language Specification. http://www.scala-lang.org. Accessed Jan-21-2019.
[63] Mike Papadakis and Yves Le Traon. 2012. Using mutants to locate" unknown" faults. In *ICST (ICST'12)*. IEEE, 691–700.
[64] Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: Mutation-based Fault Localization. *Softw. Test. Verif. Reliab.* 25, 5-7 (Aug. 2015), 605–628.

[65] Mike Papadakis and Nicos Malevris. 2010. Automatic mutation test case generation via dynamic symbolic execution. In *ISSRE (ISSRE'10)*. IEEE, 121–130.

[66] Yu Pei, Carlo A. Furia, Martin Nordio, Yi Wei, Bertrand Meyer, and Andreas Zeller. 2014. Automated Fixing of Programs with Contracts. *IEEE TSE* 40, 5 (2014), 427–449.

[67] Alexandre Perez, Rui Abreu, and Marcelo d'Amorim. 2017. Prevalence of Single-Fault Fixes and Its Impact on Fault Localization. In *ICST*. 12–22.

[68] Jeff H. Perkins, Sunghun Kim, Samuel Larsen, Saman P. Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin C. Rinard. 2009. Automatically patching errors in deployed software. In *SOSP (SOSP'09)*. 87–102.

[69] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The Strength of Random Search on Automated Program Repair. In *ICSE (ICSE 2014)*. ACM, New York, NY, USA, 254–265.

[70] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An Analysis of Patch Plausibility and Correctness for Generate-and-validate Patch Generation Systems. In *ISSTA (ISSTA 2015)*. ACM, New York, NY, USA, 24–36.

[71] Ripon K Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R Prasad. 2017. ELIXIR: effective object oriented program repair. In *ASE (ASE'17)*. IEEE Press, 648–659.

[72] Alireza Salahirad, Hussein Almulla, and Gregory Gay. 2019. Choosing The Fitness Function for the Job: Automated Generation of Test Suites that Detect Real Faults. *STVR* (2019). to appear.

[73] David Schuler and Andreas Zeller. 2009. Javalanche: efficient mutation testing for Java. In *Proc. of the the 7th joint meeting of the European soft. eng. conf. and the ACM SIGSOFT symp. on The foundations of soft. eng.* ACM, 297–298.

[74] Eric Schulte, Stephanie Forrest, and Westley Weimer. 2010. Automated program repair through the evolution of assembly code. In *ASE (ASE'10)*. ACM, 313–316.

[75] Scitools. 2019. Understand. https://scitools.com/. Accessed Jan-18-2019.

[76] Andrew Scott, Johannes Bader, and Satish Chandra. 2019. Getafix: Learning to fix bugs automatically. *arXiv preprint arXiv:1902.06111* (2019).

[77] Maxim Shafirov. 2017. Kotlin for Android. https://blog.jetbrains.com/kotlin/2017/05/kotlin-on-android-now-official/. Accessed Jan-9-2019.

[78] Undo Software. 2016. Increasing software development productivity with reversible debugging. https://undo.io/media/uploads/files/Undo_ReversibleDebugging_Whitepaper.pdf. Accessed: Jan. 21, 2019.

[79] Jeongju Sohn and Shin Yoo. 2017. FLUCCS: using code and change metrics to improve fault localization. In *ISSTA (ISSTA'17)*. ACM, 273–283.

[80] Shin Hwei Tan and Abhik Roychoudhury. 2015. relifix: Automated Repair of Software Regressions. In *ICSE (ICSE'15)*. 471–482.

[81] Shin Hwei Tan, Hiroaki Yoshida, Mukul R Prasad, and Abhik Roychoudhury. 2016. Anti-patterns in search-based program repair. In *FSE (FSE'16)*. 727–738.

[82] Yida Tao, Jindae Kim, Sunghun Kim, and Chang Xu. 2014. Automatically Generated Patches As Debugging Aids: A Human Study. In *FSE (FSE'14)*. ACM, New York, NY, USA, 64–74.

[83] Rijnard van Tonder and Claire Le Goues. 2018. Static automated program repair for heap properties. In *ICSE (ICSE'18)*. 151–162.

[84] Westley Weimer. 2006. Patches As Better Bug Reports. In *GPCE (GPCE'06)*. ACM, New York, NY, USA, 181–190.

[85] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware patch generation for better automated program repair. In *ICSE (ICSE'18)*. ACM, 1–11.

[86] "Wikipedia". 2019. Levenshtein Distance. https://en.wikipedia.org/wiki/Levenshtein_distance. Accessed Jan-19-2019.

[87] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE TSE* 42, 8 (Aug. 2016), 707–740.

[88] Qi Xin and Steven P Reiss. 2017. Identifying test-suite-overfitted patches through test case generation. In *ISSTA (ISSTA'17)*. ACM, 226–236.

[89] Qi Xin and Steven P. Reiss. 2017. Leveraging Syntax-related Code for Automated Program Repair. In *IEEE ASE (ASE'17)*. IEEE Press, 660–670.

[90] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. 2018. Identifying patch correctness in test-based program repair. In *ICSE (ICSE'18)*. 789–799.

[91] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise condition synthesis for program repair. In *ICSE (ICSE'17)*. IEEE Press, 416–426.

[92] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian R. Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2017. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE TSE* 43, 1 (2017), 34–55.

[93] Jooyong Yi, Shin Hwei Tan, Sergey Mechtaev, Marcel Böhme, and Abhik Roychoudhury. 2018. A correlation study between automated program repair and test-suite metrics. In *ICSE*. 24.

[94] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. 2011. Localizing failure-inducing program edits based on spectrum information. In *ICSM*. IEEE, 23–32.

[95] Lingming Zhang, Tao Xie, Lu Zhang, Nikolai Tillmann, Jonathan De Halleux, and Hong Mei. 2010. Test generation via dynamic symbolic execution for mutation testing. In *ICSM*. 1–10.

[96] Lingming Zhang, Lu Zhang, and Sarfraz Khurshid. 2013. Injecting mechanical faults to localize developer faults for evolving software. In *OOPSLA*. 765–784.

[97] Xiangyu Zhang, Haifeng He, Neelam Gupta, and Rajiv Gupta. 2005. Experimental evaluation of using dynamic slices for fault location. In *Proc. of the sixth intl. symp. on Auto. analysis-driven debug.* ACM, 33–42.