Information and Software Technology 112 (2019) 164–175

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

A bug finder refined by a large set of open-source projects

Jaechang Nam[a,1,*], Song Wang[b], Yuan Xi[b], Lin Tan[b]

[a] Handong Global University, Pohang, Gyeongsangbuk-do, Korea
[b] University of Waterloo, Waterloo, ON, Canada

ARTICLE INFO

Keywords:
Static bug finder
bug detection rules
bug patterns

ABSTRACT

Context: Static bug detection techniques are commonly used to automatically detect software bugs. The biggest obstacle to the wider adoption of static bug detection tools is false positives, i.e., reported bugs that developers do not have to act on.
Objective: The objective of this study is to reduce false positives resulting from static bug detection tools and to detect new bugs by exploring the effectiveness of a feedback-based bug detection rule design.
Method: We explored a large number of software projects and applied an iterative feedback-based process to design bug detection rules. The outcome of the process is a set of ten bug detection rules, which we used to build a feedback-based bug finder, FeeFin. Specifically, we manually examined 1622 patches to identify bugs and fix patterns, and implement bug detection rules. Then, we refined the rules by repeatedly using feedback from a large number of software projects.
Results: We applied FeeFin to the latest versions of the 1880 projects on GitHub to detect previously unknown bugs. FeeFin detected 98 new bugs, 63 of which have been reviewed by developers: 57 were confirmed as true bugs, and 9 were confirmed as false positives. In addition, we investigated the benefits of our FeeFin process in terms of new and improved bug patterns. We verified our bug patterns with four existing tools, namely PMD, FindBugs, Facebook Infer, and Google Error Prone, and found that our FeeFin process has the potential to identify new bug patterns and also to improve existing bug patterns.
Conclusion: Based on the results, we suggest that static bug detection tool designers identify new bug patterns by mining real-world patches from a large number of software projects. In addition, the FeeFin process is helpful in mitigating false positives generated from existing tools by refining their bug detection rules.

1. Introduction

There are a multitude of static bug detection techniques, many of which have been adopted by the industry [1–4]. A major challenge for static bug detection techniques is the large number of false alarms (false positives) that occur, i.e., reported bugs that developers do not have to act on. Researchers have proposed techniques to filter out false positives by prioritizing all reported warnings and focusing on warnings with top priority [5–8], building statistical models to classify false positives and true warnings [6,9–14], combining static and dynamic analysis to filter out false positives [15–18], etc. However, 30–90% of reported warnings by static bug detection tools are still false positives [5,6,9,15,19]. Such a large number of false positives often make developers reluctant to use bug detection tools entirely owing to the overhead of alert inspection [20].

To address the false positive issue of static bug detection tools, we propose an iterative and feedback-based process to design bug detection rules that report fewer false positives [21]. The process consists of an iterative manual method to design bug detection rules from bug-fixing patches, and to further refine these rules by using false positives (feedback) from a large number of software projects. In addition, we implement a Feedback-based bug Finder, FeeFin, using this process from scratch. This paper is the extended version of its short summary for an ICSE2018 poster [21].

The scope of our study is represented in Fig. 1. The grey area (***A***) shows all the bugs that are not detected or fixed in the world. A recent study by Habib and Pradel shows that bugs not detected by bug detection tools amount to 95.5% in their experiments [22]. The circle ***B*** represents bugs that can be detected by existing static bug detection tools. The intersection between ***A*** and ***B*** shows true positives. However, as reported in previous studies [9,20], the remaining area of ***B*** often contains false positives. In this study, we implemented our own bug detection tool, FeeFin, that can detect bugs with fewer false alarms and bugs missed by other tools, as denoted by the circle ***C***.

The goal of this study is to explore the effectiveness of feedback-based bug detection rule design in reducing false positives and detecting

* Corresponding author.
E-mail addresses: jcnam@handong.edu (J. Nam), song.wang@uwaterloo.ca (S. Wang), y25xi@uwaterloo.ca (Y. Xi), lintan@uwaterloo.ca (L. Tan).
1 This work has been conducted since Jaechang Nam was at the University of Waterloo as a postdoctoral fellow.

https://doi.org/10.1016/j.infsof.2019.04.014
Received 28 July 2018; Received in revised form 27 April 2019; Accepted 30 April 2019
Available online 1 May 2019

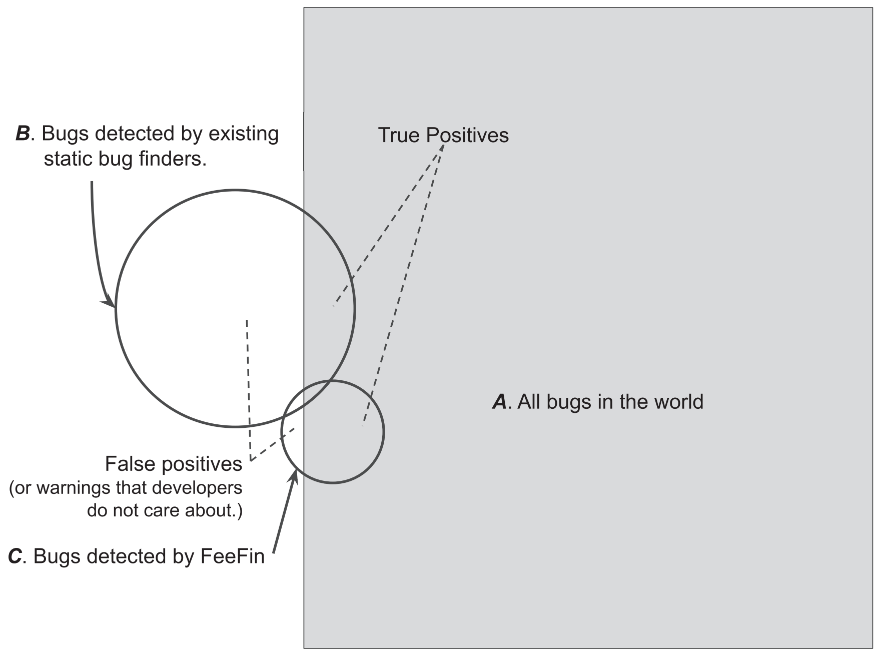
B. Bugs detected by existing static bug finders.
True Positives
A. All bugs in the world
False positives (or warnings that developers do not care about.)
C. Bugs detected by FeeFin

Fig. 1. The scope of our case study.

new bugs. To achieve this goal, we conduct a case study on 1880 Java projects to evaluate FeeFin in terms of bug detection with fewer false positives.

The contributions of our study are as follows:

- **Feedback-based bug detection rule design**: Our bug detection rule design is based on an iterative manual process that repeatedly refines detection rules by using false positives from hundreds of software projects.
- **FeeFin**: We implemented a feedback-based static bug finder, FeeFin. Initially, we found ten simple, yet effective bug detection rules by analyzing 1,622 bug-fixing patches. Of the ten rules, six detection rules are new and the bugs detected by them are unique to FeeFin, while the other four produce considerably fewer false positives than their corresponding rules in existing bug detection tools.
- **Case study and empirical evaluation of the feedback-based bug detection rule design**: We conducted a case study to evaluate the detection performance of FeeFin rules on a large number of Java projects. We considered FeeFin in two different scenarios, i.e., Just-in-time (JIT) and Snapshot FeeFin. JIT FeeFin detects bugs before or at the time developers commit source code changes, while Snapshot FeeFin detects bugs in the latest snapshots of projects before release (see Section 2.3). In our preliminary evaluation, JIT FeeFin successfully detected 160 known bugs with only one false positive from 599 Java projects on GitHub. In the latest versions of these 599 projects and an additional 1281 (a total of 1,880) Java projects, Snapshot FeeFin detected 98 previously unknown bugs, 63 of which have been reviewed by developers at the time of writing: 57 have been confirmed as true bugs. False positives were decreased from 5 to 0, as the detection rules were iteratively refined in the projects. In addition, we investigated our detection results with four existing bug detection tools—*PMD, FindBugs, Facebook Infer, and Google Error Prone*. We found that FeeFin has the potential benefits in identifying new bug patterns, as well as in improving existing bug patterns.

2. Approach

Fig. 2 shows the overview of our FeeFin process [21]:

1. Manual patch analysis: First, we manually analyze patches to identify common bug patterns. While one can use known bug patterns [23] or use bug patterns mined automatically [24,25], we manually analyze patches to collect valid bug patterns and understand the possible heuristics to fix a bug.
2. A feedback-based detection rule design: We design detection rules for the identified bug patterns, apply the rules to a set of software projects, and manually analyze the false positives on these projects. We keep refining the rules iteratively until the false positives are filtered out.
3. FeeFin: We implement FeeFin based on the rules from the feedback-based detection rule design.

The process involves manual steps to refine detection rules by analyzing false positives from a large number of software projects. This manual process could be a common step when developing static bug detection tools. For example, Chen et al. [26] summarized and refined six anti-patterns to detect log related bugs from three projects, whereas Jin

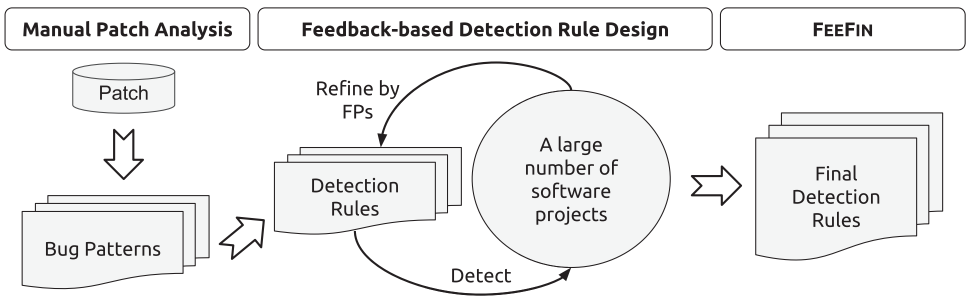

Fig. 2. Overview of the FeeFin process (FPs = false positives) [21].

et al. [27] summarized and refined four bug patterns by using performance bugs collected from five projects. When applied to new projects, although these tools did refine their rules, they still reported non-trivial false positive rates, i.e., 40% from [26] and 30% from [27]. One possible reason is that refining rules based on a small number of subjects is not sufficient to generalize the rules and could easily make the rules overfitted to the subjects. To address the issue, we refine detection rules on *a large number of software projects*. In this section, we show the rule design processes of the FeeFin process in detail. To verify whether FeeFin is effective in mitigating false positives, after implementing FeeFin, we conduct a case study (Section 3 and Section 4) to evaluate its performance.

2.1. Manual patch analysis

To collect common bug patterns, we manually analyzed bug patches collected from a software repository, git. Common bug patterns are 'error-prone coding practices', e.g., 'simple and common mistakes' [28]. Because developers introduce and fix bugs by submitting patches to the software repositories, analyzing patches, i.e., bug-fixing and bug-introducing changes, is an efficient way to find the common bug patterns. To identify bug-introducing changes, one first needs to link bug reports in an issue tracking system to bug-fixing changes in a version control system. After that, the original source code changes that induce the bug-fixing changes are considered bug-introducing changes. Bug-introducing changes can be automatically identified by the SZZ algorithm [29]. However, this process can be noisy, e.g., the issue reports labeled as bugs may be actually feature requests but not bug reports. We started to analyze 258 and 577 patches that fix bug-introducing changes collected between 2010-09-17 and 2011-02-28 from Lucene and between 2007-09-12 and 2009-09-14 from Jackrabbit respectively. The datasets of both projects have bug issue reports and patch data manually verified by Herzig et al. [30], and have been widely used as trustworthy datasets in defect prediction [31,32].

After analyzing these patches, we found that common bug patterns could likely be identified in small patches whose number of changed lines is around five. This may be because large patches are difficult to understand and investigated for patterns without project-specific domain knowledge. In addition, it is obvious that analyzing smaller patches rather than larger ones can save manual effort. Based on this observation, we also analyzed the small patches in Hadoop-common and HBase, two of the most popular projects in the Apache Software Foundation (ASF), to identify bug patterns. Hadoop-common is a representative project for distributed computing and HBase is the Hadoop database. We define a small patch as having no more than five changed lines. On average, we could analyze approximately 40 patches per hour.

In total, we analyzed 1,622 small patches and finally identified ten bug patterns from Lucene, Jackrabbit, Hadoop-common, and HBase using the feedback-based detection rule design (Section 2.2). While analyzing patches, we identified new bug patterns as well as those that overlapped with the ones used by existing bug detection tools. Because the goal of this study is to evaluate if the feedback-based detection rule design is effective in reducing false positives, we investigate the detection results using both new and existing bug patterns in our case study.

We have identified the following ten *initial* bug patterns [21]. Because detecting bugs using these initial patterns may generate many false positives, Section 2.2 shows how we refine these initial patterns by using false positives from a large number of software projects. Six of these bug patterns are completely new, while the rest (patterns (2), (3), (8) and (9)) may overlap with existing bug patterns. Although these four patterns are not new, the corresponding refined rules that we designed significantly reduce false positives (details are in Table 4). In the example code, '-' denotes a buggy line deleted in a patch. We also show added lines ('+') in a patch to explain how a bug is fixed.

1. ***CompareSameValue*** compares the same values from a getter and a field. The example always returns true as the getter, `getVersion`, actually returns the field, `VERSION`.

```
public static final byte VERSION = 1;
public void readFields(DataInput in) throws IOException {
  byte version = in.readByte();
...
- } else if (getVersion() == VERSION) {
+ } else if (getVersion() == version) {
...
public byte getVersion() {
  return VERSION;
}
```

2. ***EqualToSameExpression*** compares the same expression with '=='. The example always returns true and this redundancy is removed in a fix.

```
- if(replicaInfo.getStamp() == replicaInfo.getStamp()) {
+ if(block.getStamp() == replicaInfo.getStamp()) {
```

3. ***IllogicalCondition*** loads a known null object into conditional expressions that can cause a null pointer exception (NPE) because of an illogical condition. In the example, when the left operand of '||' in the deleted line is false, `prefix` is null and will cause an NPE in the right operand. The '||' is replaced into '&&' in the added line of the patch for avoiding this NPE when calling `prefix.value()`.

```
- if(prefix != null || prefix.value()!=null)
+ if(prefix != null && prefix.value()!=null)
```

4. ***IncorrectDirectorySlash*** causes an inconsistent path with an additional slash. In the example, depending on whether `args[1]` ends with a slash, `mDir` could have an inconsistent path. This is fixed by calling `getAbsolutePath()` which always returns a path that does not end with a slash.

```
- File mDir = new File(args[1] + "-tmp");
+ File mDir = new File(args[1]);
+ mDir = new File(mDir.getAbsolutePath() + "-tmp");
```

5. ***IncorrectMapIterator*** is the wrong iteration of a map by values instead of an entry set. In the example, developers incorrectly used the values of a map to iterate the map.

```
Map m = this.getMap();
- for(Iterator i=m.values().iterator();i.hasNext();){
+ for (Iterator i=m.entrySet().iterator();i.hasNext();){
Map.Entry e = (Map.Entry) i.next();
String uri = (String) e.getKey();
```

6. ***MissingLForLong*** causes an integer overflow while defining a long integer with the multiplication of actual values. The example shows its fix by adding a suffix, L, for a long type on the largest value.

```
- final long DEFAULT_MAX_FILE_SIZE = 10*1024*1024*1024;
+ final long DEFAULT_MAX_FILE_SIZE = 10*1024*1024*1024L;
```

7. ***RedundantException*** handles the same exception in both a try-catch-finally block and a method declaration with 'throws'. This may miss the intended `catch` or `finally` blocks. In the example, the `finally` block is not executed when there is an IOException from initializing `os` or `is` before the `try` block as the method `copy()` throws the same exception. This bug leads to a memory leak and is fixed by moving the statements, causing the IOException in the `try` block.

```
public void copy() throws IOException {
-     IndexOutput os = createOutput(dest);
-     IndexInput is = openInput(src);
+     IndexOutput os = null;
+     IndexInput is = null;
      try {
+       os = createOutput(dest);
+       is = openInput(src);
...
      } catch (IOException ioe) {...
      } finally {
       IOUtils.closeSafely(pExp, os, is);
```

8. ***RedundantInstantiation*** redundantly initializes an object, which can cause a performance issue. The example shows a redundant object instantiation of `conn` which is removed in a patch.

```
- Connection conn = getConnection();
- conn = new Connection(url,user);
+ Connection conn = new Connection(url,user);
```

9. ***SameObjEquals*** compares the same object, method, variable, etc., with `equals()`. The comparison in the example always returns true and the redundancy is removed in a fix.

```
- return other.getUUID().equals(other.getUUID());
+ return getUUID().equals(other.getUUID());
```

10. ***WrongIncrementer*** is the misuse of an outer incrementer in an inner loop. In the example, developers incorrectly used the incrementer `i`. If the sizes of `a` and `b` are not the same, this would cause an index-out-of-bounds exception.

```
for(int i=0; i < a.length; i++){
 for(int j=0; j < b.length; j++){
- a[i] = b[i];
+ a[i] = b[j];
```

2.2. A feedback-based detection rule design

A Feedback-based detection rule design is an iterative manual process that improves the bug detection rules by using false positives from detection results on a large set of software projects as feedback. With the false positives generated by a bug detection rule, we can easily garner the direct problems of the rule and further improve it by fixing these problems. This is a benefit gained from mining software repositories. To obtain feedback, we used 599 Java open-source projects—427 from ASF and 127 from Google—mirrored on GitHub. These projects were chosen because ASF and Google are representative organizations that operate various Java open-source projects.

Through the given example for *EqualToSameExpression*, we can see how this process is effective even if it is based on manual effort.

2.2.1. Initial rule design

After identifying an initial bug pattern, we implement the detection rule for the pattern by forming a detection question that can reveal whether the source code contains this bug pattern. For the bug pattern *EqualToSameExpression* we generated the following initial question:

- Q1: Does a condition statement compare the same expression with ‘==’?

If the answer to this question is “yes”, we consider a potential bug detected. Based on this question, we implemented a detection rule. We improved the detection rule incrementally by studying the rule’s effectiveness on the chosen set of projects.

2.2.2. Rule revision 1

To revise the initial rule, we first apply the rule to past commits in the 599 Java projects. Then, we analyze the detection results that do not have corresponding bug-fixing commits. If the detected results from the past commits do not have bug-fixing commits, they are suspected as false positives because they have not been fixed yet [9]. Using the initial rule implementation for *EqualToSameExpression*, we detected 136 matches from past commits. Of these matches, we only examined the detection results that were not fixed yet. One of these matches is as follows:

```
1 | double thisDouble, otherDouble;
2 | ...
3 | if ((thisDouble != otherDouble)
4 |   && ((thisDouble == thisDouble) ...
```

Comparing the same variable defined by a double type checks whether the variable is NaN or not. When the expression in Line 4 returns false, `thisDouble` is NaN. This is not a false positive and is also valid for variables defined as a float type. Thus, we added the question ‘Q2: Is `x == x` not used for checking if `x` is NaN when its type is float or double?’. After revising our rule for *EqualToSameExpression* based on Q2, the number of matches decreased to 94, significantly reducing false positives.

2.2.3. Rule revision 2

Among the 94 matches, we also found the following detection result that has not been fixed yet:

```
1 | do {
2 |   ...
3 | } while (1 == 1);
```

In Line 3, a developer intentionally uses the condition `1 == 1` instead of `true`, which is not a bug. Thus, we added the question ‘Q3: Are developers unintentionally using an always true condition with the same number literals, such as 1 == 1?’. After improving the rule based on Q3, the final number of detected potential bugs is 90.

The final bug detection rule refined by using false positives from the 599 projects for *EqualToSameExpression* is Q1-yes, Q2-yes, and Q3-yes. When the answers for all these questions is ‘yes’ for a detected case, we consider the detected case a true positive. Otherwise, it is considered to be a false positive. Each question represents one iteration and took approximately 10–30 min to be implemented. To turn bug patterns into rules, there must be no false positives in bugs detected after refining the rules. This feedback-based rule design process is applied for all bug patterns and their detection rules were repeatedly revised until no false positives that do not violate Java common coding practices were found in the detected matches. Detailed rules for the ten bug patterns are available online [33]. Note that the rules in [33] are finalized by false positives from the 599 projects and the additional 1281 projects used in our case study as the FeeFin process keeps refining detection rules.

For some bug patterns, it is challenging to come up with explicit detection rules to reduce false positives. For example, the following patch shows the potential bug pattern *InconsistentNullChecker* learned from Hadoop-common:

```
  final BalancerDatanode d = datanodeMap.get(datanodeUuid);
- if (datanode != null) {
+ if (d != null) {
  block.addLocation(d);
```

In the buggy code, the null check is on a variable, `datanode`, but the variable used in the relevant code block is `d` instead. This bug pattern considers these inconsistent null checks and their use a potential bug.

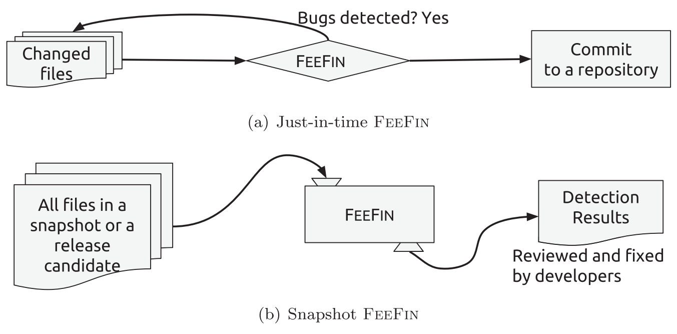

Fig. 3. Two FeeFin scenarios.

Table 1
Detected known bugs by JIT FeeFin.

Bug pattern	# True positive	# False positive
CompareSameValue	1	0
EqualToSameExpression	30	1
IllogicalCondition	10	0
IncorrectDirectorySlash	3	0
IncorrectMapIterator	2	0
MissingLForLong	7	0
RedundantException	6	0
RedundantInstantiation	6	0
SameObjEquals	51	0
WrongIncrementer	44	0
Total	160	1

To detect this type of bug, we form the initial question “Is an object in a null check consistent with the object used in the relevant code block?”. However, while applying the implementation of this detection rule, we found many false positives: correct code where different objects are used for the null checks and their use. Without project-specific knowledge, it is difficult to determine which inconsistencies are bugs. We actually identified more potential patterns in addition to the ten patterns introduced in Section 2.1. However, we had to drop potential bug patterns that possessed this type of uncertainty. Finally, we implemented the ten bug patterns as in Section 2.1. This implies that validating a rule on a large set of software projects helps a tool designer decide whether the rule for a bug pattern should be included in the bug detection tool.

2.3. *FeeFin*

Rule implementation for FeeFin is based on the abstract syntax tree (AST) and heuristics that embody the questions formed from the feedback-based detection rule design. We used the Eclipse JDT core 3.10.0 for implementing our AST parser.

FeeFin can be applied in two detection scenarios.

2.3.1. *Just-in-time (JIT) FeeFin*

Fig. 3(a) shows the overview of JIT FeeFin. In this scenario, detection is conducted before the changes are committed to a version control system (e.g., git) or while developers write source code in an integrated development environment (IDE) tool. We simulate this scenario on the past commits of 599 Java projects from GitHub to validate our FeeFin implementation for known bugs (Section 2.4).

2.3.2. *Snapshot FeeFin*

Snapshot FeeFin detects bugs in source code files in a snapshot or a release candidate, as shown in Fig. 3(b). Most existing static bug detection tools for Java work as in this scenario and analyze byte code or code snapshots. In this study, we apply Snapshot FeeFin on 1880 Java projects on GitHub.

2.4. *Preliminary result from past commits*

To estimate the detection performance of our FeeFin implementation, we verified the detection results on all past commits of the same 599 projects for known bugs, which have fix commits.

Table 1 shows the statistics of the detected known bugs from the 599 projects after verifying the bugs with their related fix commits. Specifically, we found 160 true positives and only one false positive. In other words, the false positive rate of FeeFin on the past commits of 599 ASF and Google’s Java open-source projects is only 0.625% in this preliminary result.

We verify a detected bug as a true positive in the following way:

1. We first find a fix commit that changes the buggy code lines. The fix commit can be identified automatically by tracking the commit history [34].
2. The commit may include a cosmetic change instead of a bug-fixing change. Because the cosmetic change still contains buggy code, we track a fix commit again from the cosmetic change.
3. If the path is renamed in the repository, the abovementioned process cannot find a fix commit with the original path. However, the same bug is also detected in the renamed path as the bug still exists in the renamed path. Thus, we run ‘git blame’ on the renamed path with the ‘-C’ option, which can track the original path from which a buggy line was copied (an example is given below). In this way, we can find the fix commit even if the path is renamed.
4. When we find the fix commit, we analyze whether the changes in the fix commit actually correct the buggy code, because the buggy code can be removed simply by refactoring rather than directly fixing the bug. We classify the bug as a true positive only when the changes directly fix the buggy code.

The following example shows an *IllogicalCondition* bug (Line 3) and its patch (Line 5). However, the patch makes a cosmetic change:

```
1 | - if (relatesTo != null || "".equals(relatesTo)) {
2 |   ...
3 | + if ((relatesTo != null) || "".equals(relatesTo)) {
```

This cosmetic change still contains the same bug. Thus, the buggy code was detected again by FeeFin in the renamed path. By using ‘git blame -C’, we confirmed that the deleted path was actually renamed from ‘core’ to ‘kernel’ in the second sub-directory name, and the fix is as follows:

```
1 | + if ((relatesTo != null) && !"".equals(relatesTo)) {
```

As shown in this example, some patches only contain a cosmetic change and a renamed path so that bugs can still remain in the code file. In the JIT scenario (Section 2.3.1), FeeFin keeps detecting bugs to alert developers until actual fix patches are committed. Table 1 shows the number of detected bugs in this JIT scenario.

To better understand the one false positive reported by FeeFin as shown in Table 1, we show its detailed fix commit:

```
1 | >>>>>>> change the LOGIN progress using native MVP instead of gwt-
  |     presenter
2 | =======
3 | >>>>>>> change the LOGIN progress using native MVP instead of gwt-
  |     presenter
```

The entire code file contains conflicted code caused by code merging. The Eclipse JDT parser used in the FeeFin implementation interpreted the code in the patch (Line 1–3) as the following condition: `gwt - presenter == gwt - presenter`. For this reason, JIT FeeFin detects an *EqualToSameExpression* bug. This issue is directly caused by the broken Java code containing messages that show a merge conflict.

Table 2
1,880 Java projects used in the case study.

Group	Used for	Projects	Num.
1	Implementation, Evaluation	ASF Java projects mirrored on GitHub	472
		Google Java projects mirrored on GitHub	127
2	Evaluation	Top GitHub Java projects by Stars	948
3	Evaluation	Top GitHub Java projects by Forks	333

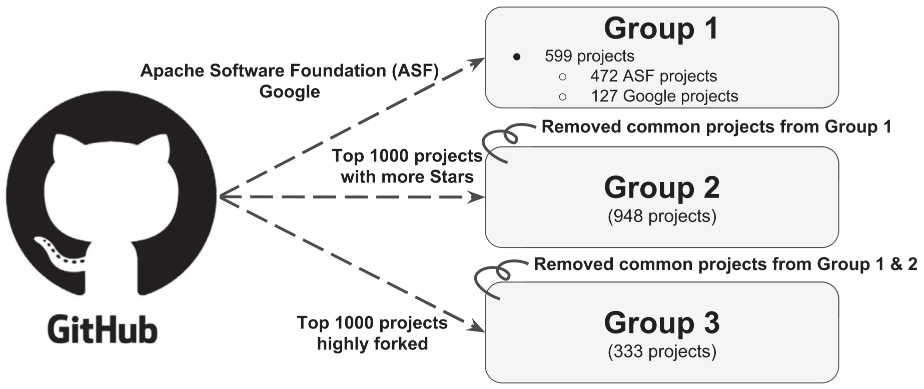
Group 1
599 projects
472 ASF projects
127 Google projects
Apache Software Foundation (ASF) Google
Removed common projects from Group 1
Top 1000 projects with more Stars
Group 2
(948 projects)
Removed common projects from Group 1 & 2
GitHub
Top 1000 projects highly forked
Group 3
(333 projects)

Fig. 4. Project selection for Group 1, 2, and 3.

Although this detection is clearly a false positive, a developer can be made aware of the broken Java code file by the merge conflict. In this sense, its detection might still be valid to a developer.

This preliminary result is from the same projects where the feedback-based detection rule design was conducted. Because of this, the preliminary result could be overfitted or underfitted by the known bugs and false positives. Thus, in Section 3, we design a case study to evaluate FeeFin in terms of 'unknown' bugs on the latest snapshots of 1880 Java projects, including the 599 projects used in the JIT scenario.

3. Case study design

Table 2 shows the number of all projects used in FeeFin implementation and/or evaluation. Note that, to detect *unknown bugs*, we use the latest versions of all these projects.

- **Group 1:** After implementing FeeFin and verifying its detection results on the 599 projects of Group 1, we also run FeeFin to detect *unknown bugs* in this group.

To evaluate FeeFin implemented on Group 1, we also run FeeFin on other two groups of GitHub projects. To identify top projects, we used the number of stars or forks that can represent the popularity of each project on GitHub. Fig. 4 describes how we select open-source projects from GitHub. Because a GitHub JSON query returns an ordered list of up to 1000 projects, we obtain top 1000 projects by the number of stars and another top 1000 projects by that of forks.

- **Group 2:** As the 1000 top projects also include some ASF and Google's projects, we applied FeeFin finally on new 948 projects in terms of stars after excluding the ASF and Google projects used in Group 1.
- **Group 3:** As FeeFin is also improved by false positives from Group 2, we chose the unique projects that are included in neither Group 1 nor Group 2 among top 1000 projects with more forks. The number of unique projects is 333 by excluding projects used in Group 1 and Group 2.

3.1. Research questions

We investigate the following research questions in this case study:

- **RQ1:** How many previously unknown bugs (i.e., new bugs) does FeeFin detect?
- **RQ2:** How are the refined rules in FeeFin for detecting new bugs robust against false positives?
- **RQ3:** What are the benefits of FeeFin bug patterns identified by the FeeFin process?

The goal of this study is to show the effectiveness of feedback-based rule design in terms of avoiding false positives. Thus, we focused on research questions that investigate false positives and new bugs detected in real world projects from GitHub. RQ1 and RQ2 investigate the detected bugs that still exist and are previously unknown in the latest source code snapshots of the Java projects. To verify if the detected bugs are true positives or false positives, we report them to issue tracking systems and wait for developers' confirmation. In addition, we investigate the benefits of FeeFin bug patterns identified by the FeeFin process (RQ3).

3.2. Evaluation steps

3.2.1. FeeFin verification for new bugs

The detected bugs that still exist in code snapshots were reported to issue tracking systems, e.g., Jira, as they are potentially new bugs (RQ1 and RQ2). Based on developers' responses, we count true bugs and false positives. We also count the number of bugs fixed by developers after we reported them.

3.2.2. FeeFin verification for new bug patterns

By the FeeFin process in Fig. 2, we identified bug patterns by manual patch analysis. To check if these patterns are new, we verified our detection results by the existing bug detection tools, namely PMD, FindBugs, Facebook Infer, and Google Error Prone. When the existing tools do not have bug patterns in FeeFin, we considered that they are new patterns and identifying bug patterns by mining software repositories have the potential to improve bug detection tools. PMD is an open-source static code analyzer, which aims for finding common programming flaws like unused variables, empty catch blocks, unnecessary object creation, etc., and duplicated code in source code [2]. PMD does not require to compile the source code and it supports Java, C, C++, C#, PHP, Ruby, JavaScript, Python, etc. FindBugs is a typical static bug finder designed for Java to find programming bugs, confusing code, bad practices, etc. [1,28]. It basically operates on Java byte code rather than source code.

Facebook Infer and Google Error Prone are two lightweight open source static bug finders that are built and used inside the companies.

Different from FindBugs and PMD, these tools focus on several types of bugs with clear bug patterns, e.g., null dereference, resource leak, etc. Thus, these tools are reported to show better detection performance [35,36].

4. Results

In this section, we report all results to address research questions. We publicly shared bug detection results in [33].

4.1. RQ1: New bugs detected by FeeFin

Table 3 shows FeeFin's bug detection results on the latest code snapshots of the Java projects in Table 2. Before reporting the detected bugs, we found that some bugs were detected in retired/inactive projects. In addition, some other detected bugs were clearly false positives (CFPs). Thus, we only reported the detected unknown bugs that require developers' confirmation for active projects because the chances of receiving developers' reply for the inactive and retired projects are low. We defined a project as an active project if its last commit or the last closed bug report was less than a year ago.

4.1.1. Group 1

For the detected bugs in Group 1, we reported 52 out of the 63 bugs to issue tracking systems.

Overall, 27 out of 52 reported bugs (# RT) were confirmed to be true bugs by developers and we are awaiting the developer responses for 20 bugs. For example, of the 11 reported bugs of the *WrongIncrementer* pattern, 8 bugs were reviewed by developers as TPs and there were no FPs. For the remaining bugs (3 WCs), we have not yet received any comments from the developers. In addition, after they were reported, 20 out of 27 TPs were fixed by developers (the pull requests we submitted were also accepted for some bugs).

Table 3
New bugs detected by Snapshot FeeFin. Bug patterns that did not detect any new bugs were excluded. (# DB: detected bugs, # RT: bugs reported to issue tracking systems, # TP: true positives reviewed by developers, # FP: false positives reviewed by developers, # WC: waiting for confirmation, # FX: bugs fixed by developers) [21].

Bug pattern	# DB	# RT	# TP	# FP	# WC	# Fix
Group 1						
CompareSameValue	5	5	0	5	0	0
EqualToSameExpression	8	6	3	0	3	2
IllogicalCondition	2	2	1	0	1	1
MissingLForLong	1	1	0	0	1	0
SameObjEquals	33	26	15	0	11	12
WrongIncrementer	14	12	8	0	4	5
Subtotal	**63**	**52**	**27**	**5**	**20**	**20**
Group 2						
CompareSameValue	6	3	2	1	0	2
EqualToSameExpression	3	0	0	0	0	0
IncorrectDirectoySlash	2	2	0	2	0	0
MissingLForLong	1	1	0	0	1	0
RedundantInstantiation	1	1	1	0	0	1
SameObjEquals	15	6	5	0	1	3
WrongIncrementer	7	6	4	1	1	2
Subtotal	**35**	**19**	**12**	**4**	**3**	**8**
Group 3						
CompareSameValue	1	1	0	0	1	0
EqualToSameExpression	2	1	1	0	0	1
IllogicalCondition	1	1	1	0	0	1
MissingLForLong	2	2	0	0	2	0
SameObjEquals	12	12	7	0	5	7
WrongIncrementer	13	10	9	0	1	7
Subtotal	**31**	**27**	**18**	**0**	**9**	**16**
Total	**129**	**98**	**57**	**9**	**32**	**44**

4.1.2. Group 2

We detected 35 new potential bugs, 19 of which were reported to issue tracking systems. 16 of the 19 reported bugs were reviewed by developers (12 TPs and 4 FPs). The remaining 3 are awaiting the developers' confirmation. 8 out of 12 TPs were fixed after we reported them.

4.1.3. Group 3

A total of 31 bugs were detected, and 27 were reported. All the confirmed bugs in Group 3 are TPs (18 out of 27 RTs). 16 of 18 TPs were fixed after we reported them. The remaining 9 are awaiting the developers' confirmation.

The detection results that were not reported (# DB - # RT) include the bugs detected in inactive and retired projects, plus CFPs. There were 2, 13, and 4 CFPs in the three groups. Most CFPs (9 out of 13) in Group 2 were code smells from SameObjEquals (e.g., a.equals.(a);). We did not report them, as developers are unlikely to fix them, as in Section 4.2.1. Furthermore, we did not add them to the FP count because (1) they can be considered to be bugs as one should not compare the same object, and (2) they are intermediate results, as we used these CFPs to further refine rules to reduce FPs in Group 3 (SameObjEquals CFPs disappeared in Group 3).

57 of the 98 reported bugs from the three groups combined were reviewed by developers as true bugs, 44 of which have already been fixed by developers after we reported them.

4.2. RQ2: False positives in the new bugs

4.2.1. Group 1

Among the reported bugs from Group 1 of Table 3, we found five false positives of *CompareSameValue*.

The following is the five false positives of *CompareSameValue*:

```
1  ...
2  public static final int SM_BINARY_KEYS = 1;
3  ...
4  public int getStorageModel() {
5    return SM_BINARY_KEYS;
6  }
7  ...
8  protected Object[] getKey(NodeId id) {
9    if (getStorageModel() == SM_BINARY_KEYS) {
10     return new Object[] { id.getRawBytes() };
```

Because the method, `getStorageModel`, returns `SM_BINARY_KEYS`, the condition in Line 9 always returns true. There are additional four locations that use the same condition in the same file. However, developers did not want to act on these issues and replied as follows:

"This is a non-final class and subclasses... other classes override this method and return a different value. This works as designed."

These cases will not cause any problem when being always overridden. Based on these false positives we could have rooms for improvement of our FeeFin rules. For example, we could improve FeeFin rules by ignoring the conditions that always return true are in a method that is overridden.

Although FeeFin rules were improved by feedback from past commits of the 599 Java projects, there were false positive cases that rarely happen as the examples above. When the bug finder designers can regularly (e.g., once per month) apply FeeFin for new commits, some false positive types can be identified quickly because FeeFin bug patterns are likely simple bugs that can be fixed in one line. When false positives were identified, improving FeeFin can be quickly conducted as well.

Table 4
Detected and overlapped results by both FindBugs and Snapshot FeeFin on the detected bugs. (# TP: the number of true positives, # FP: the number of false positives).

FeeFin			Overlap		FindBugs		
Bug Pattern	# TP	# FP	# TP	# FP	# TP	# FP	Bug Pattern
EqualToSameExpression	3	0	1	0	1	0	*SA_(LOCAL|FIELD)_SELF_COMPARISON*
IllogicalCondition	1	0	1	0	3	26	*NP_LOAD_OF_KNOWN_NULL_VALUE*
RedundantInstantiation	1	0	1	0	1	19	*DLS_DEAD_LOCAL_STORE*
SameObjEquals	11	0	5	0	5	0	*SA_(LOCAL|FIELD)_SELF_COMPARISON*
Total	16	0	8	0	10	45	**Total**

4.2.2. Group 2

Of the 19 reported bugs, there were four false positives. Two out of the four reviewed by developers were real issues in the same project but developers did not want to fix as they do not affect users but developers debugging the project:

"... Yes we could clean this up a bit, however, this... is designed for people who know what they are doing and are doing debug... "

The third FP in *WrongIncrementer* reviewed by a developer is not a bug. However, as the developer commented, the code requires to be cleaned up:

"I'll admit to the code being very unclear... there's no actual bug, but it's a gnarly bit of code... I'd love to see this gigantic block cleaned up in the future to be more clear."

The FP in *CompareSameValue* is similar to the false positive example in Group 1. The condition that always returns true is in a method defined in an interface. Methods defined in interfaces usually are overridden by classes where implement the interfaces. Based on this FP, we further improved our FeeFin.

4.2.3. Group 3

We reported 27 new bugs to issue tracking systems. As we iteratively improve the FeeFin rules, there were no FPs reviewed by developers yet.

The key idea of the FeeFin process is to learn from various false positives from a large number of software projects to refine detection rules. The notable result in Table 3 is that there are no false positives in Group 3 as FeeFin rules are refined in Group 1 and Group 2. As discussed in Section 2, the studies by Chen et al. and Jin et al. [26,27] refined their rules by false positives from 3 or 5 projects only. This could easily make the rules overfitted to the evaluated projects. Because of this, their approaches had 30%–40% false positives when they are applied to new projects. Some of our rules also suffer from the same issue. For example, the rule, *CompareSameValue*, has 5 false positives on the 599 projects in Group 1. Although this rule was refined by using these false positives, another false positive still exists in Group 2 among the bugs reviewed by developers. This finding implies the importance of iteratively verifying the rules on a large number of projects to address the false positive limitation of static bug finders.

There are no false positives reviewed by developers in the last group as FeeFin rules are iteratively improved by using the false positives reviewed by developers.

4.3. RQ3: The benefits on bug patterns by the FeeFin process

4.3.1. Bug patterns identified by manual patch analysis

To investigate whether identifying detection rules by mining software repositories is helpful to find new bugs, we checked whether existing tools can detect bugs FeeFin detected. In Section 2.1, we introduced that six out of ten bug patterns we manually verified are new ones. We considered that they are new when the existing tools such as PMD, FindBugs, Facebook Infer, and Google Error Prone cannot detect the same bugs. Of the 57 true positive bugs in Table 3, 23 bugs were detected by new bug patterns (eight from Group 1, six from Group 2, nine from Group 3) such as *CompareSameValue* and *WrongIncrementer*. As we identified bug patterns from scratch, these results show manual patch analysis by mining software repositories is worth to find new bug patterns.

In addition, bug patterns already in the existing tools missed some bugs FeeFin detected. Of the 57 true positives from Table 3, 34 bugs were detected by existing bug patterns such as *EqualToSameExpression, IllogicalCondition, RedundantInstantiation*, and *SameObjEquals*. The 17 cases out of the 34 bugs were detected by *SameObjEquals* but the existing tools could not detected them. For example, FindBugs could not detect a case that calls 'equals(...)' from the same object method call, e.g., `getUUID().equals(other.getUUID())`. This means the feedback-based detection rule design can improve the existing bug patterns.

Our FeeFin process in Fig. 2 has the potential benefits to identify new bug patterns and to improve existing bug patterns.

4.3.2. Detection result comparison between FeeFin and findbugs

We conducted additional investigation between FeeFin and FindBugs with the same rules that detected the same bugs. Table 4 shows the detection results by FindBugs and FeeFin on the projects. Note that we compared only four patterns shared by FeeFin and FindBugs so that our results are limited on specific samples. Because the same rules between FeeFin and other existing tools are limited, we focused on FindBugs that has more similar rules to ones in FeeFin than other existing tools [1]. Thus, we choose FindBugs as a representative tool to compare.

Among the ten patterns used in this case study, we investigated four patterns that correspond to the patterns of FindBugs. The four patterns are *IllogicalCondition, SameObjEquals, RedundantInstantiation*, and *EqualToSameExpression*, and their corresponding ones from FindBugs are

- *NP_LOAD_OF_KNOWN_NULL_VALUE* for *IllogicalCondition*,
- *DLS_DEAD_LOCAL_STORE* for *RedundantInstantiation*,
- and *SA_(FIELD|LOCAL)_SELF_COMPARISON* for *SameObjEquals* and *EqualToSameExpression*.

We verified all detection results from FindBugs as we verified those from FeeFin in the case study.

Table 4 shows the number of bugs detected by both FeeFin and FindBugs as well as that missed by each tool. The fourth column, # TP of Overlap, shows the bugs detected by both FeeFin and FindBugs. In total, 8 bugs were detected by both tools. Because the total of TPs of FeeFin is 16, 8 bugs (= 16 – 8) were detected only by FeeFin, while other 2 bugs (= 10 – 8) were detected only by FindBugs.

The following example code snippet shows one of the two bugs that FeeFin missed but FindBugs detected:

```
1  synchronized void decRef(List<Long> dvProducersGens){
2   Throwable t = null;
3   for (Long gen : dvProducersGens) {
4    RefCount<DocValuesProducer> dvp = genDVProducers.get(gen);
5    assert dvp != null : "gen=" + gen;
6    try {
7     dvp.decRef();
8    } catch (Throwable th) {
9     if (t != null)
10     t = th; } }
```

Because there is no assignment of `t`, Line 10 cannot be reached as `t` is null. Another one was a similar bug. We reported these two bugs in an issue tracking system and a developer fixed them. In the example above, Line 9 was changed into `if(t == null)`. The *IllogicalCondition* only focused on loading a null object in a consecutive condition as explained in Section 2; thus this buggy code was not detected by FeeFin.

One of the 8 bugs missed by FindBugs is as follows:

```
1  if (this.columnOffset != null)
2   return this.getLimit() == this.getLimit() &&
3   Bytes.equals(this.getColumnOffset(), other.getColumnOffset());
```

In Line 2, the left and right operands are the same method calls. The corresponding patterns in FindBugs to *EqualToSameExpression* only consider the comparisons of fields and local variables so that FindBugs could not detect the comparisons by the same method calls. For the same reason, other 7 bugs were missed by FindBugs.

FindBugs reported 45 false positives or warnings developers may not act on. Most false positive cases are loading a null object as an argument in a method or unused local variables, which are warnings provided by a modern IDE tool. As the following example, the method, `getTokenErrorDisplay`, loads the null object, `token`.

```
1  if (token == null) {
2    message = "error" + getTokenErrorDisplay(token);
```

However, loading a null object as the argument is not an issue because the null argument is correctly dealt with in `getTokenErrorDisplay`. This detection is not a bug but can be just a warning.

The one example of unused local variables is as follows:

```
1  for (Entry<String, Map<String, MetricWindow>> entry : extraMap.entrySet()
       ) {
2   String metricName = entry.getKey();
3  }
```

In Line 2, `metricName` is not used. This is also a warning and developers may complete the `for` block with the local variable later. The remaining false positives are also similar to this example.

We intended to design a tool to detect bugs that existing bug detection tools cannot detect, and our evaluation is not meant to show that our tool can detect more bugs on a representative bug set; instead, we aim to show that our tool can detect many bugs the existing bug detection tools cannot detect, thus complementary to the existing tools. In addition, our results confirmed the findings from the study by Habib and Pradel where detection tools complement each other and missed bugs can be found with better variants of existing rules [22].

We observed that FeeFin may be useful to detect new bugs that the existing tools cannot detect. By investigating the four patterns shared by FeeFin and FindBugs, we also observed the potentials that the feedback-based detection rule design may be helpful to mitigate false positives and to find new true positives. Thus, FeeFin can be complementary to an existing tool in terms of detecting new bugs.

5. Discussion

5.1. Implications

5.1.1. Improving existing bug detection rules

The FeeFin presents an iterative process to deal with possible false positives from a large number of software projects such as those on GitHub. Although addressing false positives is a manual process, it is an one-time event and we can keep avoiding the same kinds of false positives later once addressed. In our case study, some bug patterns such as *SameObjEquals* are already in existing bug detection tools. However, in real world, developers had missed some bugs as FeeFin detected real new bugs in many software projects. One possible reason would be false positives from the existing tools. If developers would apply the existing tools for their project, we could not detect the new bugs. One of the possible reasons is too many false positives from the existing tools because existing tools are limited to refine detection rules based on false positives from a large number of projects. For example, Findbugs' rules are mainly based on the authors' and its users' experience [28]. In this sense, existing tools could be improved by adopting the FeeFin process.

5.1.2. Valid simple bugs

Because FeeFin is based on the small patch analysis, FeeFin rules usually focus on simple bugs that can be fixed by a simple change in one line. If the related APIs of these kinds of bugs were not used frequently and do not provide explicit errors, they would be difficult to be detected and fixed. Most detected bugs by FeeFin have been resided in source code for a long time. The following *WrongIncrementer* bug is an example from one of Google's open-source projects:

```
1  for (int i = 0; i < 3; i++){
2  [...]
3   for (int j = 0; j < listenKids.length; j++){
4    [...]
5    ep.addToStack(listenKids[i]);
```

In Line 5, the incrementer, `i`, must be `j`. When `listenKids.length` is less than 3, the error is revealed. A developer working on this project left the following comments in the issue report we posted:

"Wow... how did you happen to notice that? [...] After trying to replicate it, I realized just how rare this is. [...] Still, it is a bug so it should be fixed."

We observed similar comments from developers about other bugs detected by FeeFin. Simple bugs that rarely happen still need to be fixed and developers actually acted on them. Thus, our results also suggest that small patches are valuable for mining new bug patterns.

5.1.3. A benefit of common knowledge from a large set of software projects.

By using false positives of a large set of software projects, we could understand and learn common developer intentions about suspicious code in practice. The example of *WrongIncrementer* in Section 2.1 leads to the rule that detects a bug when there are violations on incrementers and their related arrays. There were no false positives related to this rule while conducting feedback-based rule design using the 599 projects. If other rules for *WrongIncrementer* could generate too many false positives that are difficult to be filtered out because of uncertain developer intentions, *WrongIncrementer* would not be included or the rules are discarded in FeeFin not to generate false positives. In this study, we observed that there are few false positives from FeeFin detection results of *WrongIncrementer* for unknown bugs. In this sense, our case study result clearly implies that using a large set of software projects for feedback-based detection rule design is beneficial.

5.1.4. *Precision vs. recall*

FeeFin trades recall for precision, thus could miss bugs. When detection rules are refined by false positives, there could be a risk that true positives can be removed with false positives. In this case, precision is improved but recall worsens when true positives are not detected by the refined rules so become false negatives. However, note that FeeFin mitigates false negatives by designing new patterns to find bugs no existing detectors can find. Thus, as we discussed in Section 4.3, FeeFin is complementary to the existing detectors that can detect bugs FeeFin misses.

5.2. *Threats to validity*

5.2.1. *External validity*

We have focused on characterizing and refining bug patterns in Java-based systems. Some of our derived bug patterns may not be directly applicable to systems that are implemented in other programming languages. We plan to examine the effectiveness of our proposed bug patterns on projects written in other programming languages.

We have characterized different bug patterns in the Java code by studying the historical changes in several popular ASF Java projects such as Lucene, Jackrabbit, Hadoop-common, and HBase. We have picked these systems due to the following two reasons: (1) widely studied in literature [30–32] (2) these four systems are actively maintained and mature. However, FeeFin rules can be extended by analyzing more software projects and our case study results would be different based on the rules we identify.

In terms of new bug patterns, we only verified new bug patterns with four existing bug detection tools, i.e. PMD, FindBugs, Facebook Infer, and Google Error Prone [1,2,4,36]. These are commonly used open-source tools or tools developed by representative companies. However, there are other commercial tools adopted by industry but we could not verify our bug patterns with the commercial tools because of a licence issue. If these tools could have bug patterns considered new in FeeFin, our results would have different interpretation.

5.2.2. *Internal validity*

Static analysis tools have been adopted and actively used by developers in companies although these tools have limitations [14,37,38]. In addition, uncaught bugs dramatically increase the cost of software quality control in later development phases [39].

However, the cost benefit of detecting bugs would vary based on types of bugs. For example, detecting security defects would save million dollar cost [37,40]. In this sense, it is questionable if designing and refining detection rules for bugs that may be less severe is still important. Thus, our interpretation of the results in this study could lead to bias in terms of practical significance of our proposed approach. We proposed FeeFin to design effective and new bug patterns from scratch by mining software repositories. Although bug patterns from FeeFin could detect new bugs with higher accuracy, FeeFin may require non-trivial cost for developing and refining bug patterns. From the practical aspect, how much FeeFin can save company's debugging effort is unknown.

5.2.3. *Construct validity*

Following prior work [19,26], we manually check whether the potential bugs detected by FeeFin are true positives. Although this process is a common practice, it contains bias because the authors of this paper are not the developers of these projects. We mitigate this threat by reporting the bugs to the corresponding issue tracking systems for further confirmation, which can take a long time. So far, 57 cases have already been reviewed by developers as true bugs.

6. Related work

6.1. *Static bug detection tools*

Many static code analysis techniques have been developed to detect bugs based on bug patterns [1,2,26,41–45]. Existing static bug detection techniques could be divided into two categories according to how bug patterns are designed, i.e., manually identified bug patterns and automatically mined bug patterns from source code.

Two widely used open-source bug detection tools for Java language, FindBugs [1] and PMD [2], detect real bugs based on manually designed bug patterns by their contributors. In addition, commercial bug detection tools with more well-designed and effective bug patterns are also available [3,4]. Most manually designed patterns focus on language-specific common bugs in software projects such as buffer overflow, race conditions, and memory allocation errors. Some other studies leverage particular types of bug patterns to detect special bugs. For example, Chen et al. [26] proposed six anti-patterns and leveraged these rules to detect log related bugs. Palomba et al. [42] propose to detect five different code smells based on five well-summarized code patterns.

For detecting project-specific bugs, many approaches leveraged rules that are mined from specific projects. Li et al. [43] developed PR-Miner to mine programming rules from C code and detect violations using frequent itemset mining algorithm. Livshits et al. [44] proposed DynaMine, which used association rule mining to extract simple rules from software revision histories and detect defects related to rule violations. Wasylkowski et al. [45] and Gruska et al. [41] proposed to detect object usage anomalies by combining frequent itemset and object usage graph models.

Different from the existing static bug detection techniques, we focused on feedback-based rule design for avoiding false positives. Our case study shows that the FeeFin rule design is helpful to refine bug detection rules to mitigate false positives. Note that, as reported in TRICORDER [38], Google also iteratively refines its static tools, e.g., Error Prone [4], by using the unactionable warnings labeled by developers. There are two differences between FeeFin and TRICORDER. First, TRICORDER uses commercial projects (i.e., projects from Google) to improve Google's static analyzers, while FeeFin refines bug detection rules by using a large scale open source projects. To the best of our knowledge, FeeFin used the largest publicly available dataset to refine its bug detection rules. Second, TRICORDER describes how to improve existing static analyzers by limited developer feedback in Google. However, FeeFin introduces approaches to both finding new bug patterns from past bugs and refining them by FPs from a large scale open source projects.

6.2. *Mining programming patterns*

Various programming patterns have been mined from source code for understanding code changes [46], code completion [47], API usage [48], fixing bugs [24,27], code smell and bug detection [49], etc.

Fluri et al. [46] use a clustering based approach to mine unknown change types in Java projects. Bruch et al. [47] proposed to improve code completion systems by learning code patterns from examples. Nguyen et al. [48] proposed an approach to learn API usage by learning from fine-grained changes. Jin et al. [27] studied 109 real-world performance bugs to discover patterns to fix performance bugs. Kim et al. [24] discovered six common repair patterns in Java by grouping bug-fixing changes. Engler et al. [49] proposed an approach to infer programmer's beliefs to find bugs by checking the inconsistent beliefs.

Different from the automated techniques mining programming patterns, we mined bug patterns manually from small patches. While the focus of our study is to reduce false positives in a bug detection tool, our manual process for bug patterns from small patches would be automated by using the existing techniques.

7. Future work

This study focused on a practical aspect (reporting fewer false alarms) of static bug detection tools. Thus, we evaluate FeeFin in terms of detecting new bugs with fewer false positives. However, other empirical aspects such as how developers provide feedback on warnings of the bug detection tools and how these warnings are selected by developers acting on also are important. We would like to explore these two directions in the future.

There are existing automated techniques such as automated bug pattern mining [25,27,43,45,50] that can be plugged into our FeeFin process. Because the goal of this study is to show how effective systematically using feedback (false positives) from a large set of projects is for detecting bugs, we did not apply them in this study. Applying automated approaches would be an interesting topic to be investigated. In addition, feedback-based detection rule design also has a room for improvement in terms of automation. In this sense, we have a plan to apply and develop an automated approach for the FeeFin process as future work.

8. Conclusion

We proposed FeeFin, a static bug finder based on feedback-based bug detection rule design. Although the FeeFin rule design requires manual effort to come up with new bug detection rules, our case study shows that FeeFin is effective as it could detect new bugs with fewer false positives. We detected 98 new potential bugs in 1880 Java projects. Of the 98 bugs, 57 and 9 were confirmed as true positives and false positives by developers respectively. We observed that the number of false positives decreases from 5 and 4 to 0 as FeeFin rules were refined and improved by the feedback-based detection rule design. We also could investigate that our FeeFin process may identify new bug patterns and improve existing bug patterns when we verified FeeFin detection results with the four existing tools, namely PMD, FindBugs, Facebook Infer, and Google Error Prone.

In this study, we observed that (1) there could be new bug patterns the existing static bug detection tools do not have, (2) manual effort from experts is still invaluable to design valid new bug detection rules that generate fewer false positives, and (3) our FeeFin process is effective to reduce false positives. The lessons learned in this study imply that (1) it is still recommended for static bug detection tool designers identify new bug patterns from real-world patches mined from a large number of software projects, and (2) the FeeFin process is helpful to mitigate false positives generated from the existing tools by refining their bug detection rules as well. Based on the findings of our study, we plan to extend FeeFin with more bug patterns and improve existing rules to reduce more false positives. In addition, we have a plan to automate the FeeFin rule design as future work.

Conflict of Interest

No conflict of interest

Acknowledgements

This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), and partially supported by Brain Korea 21 PLUS project for POSTECH Computer Science & Engineering Institute, and the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (no. 2018R1C1B6001919).

References

[1] FindBugs, 2017. http://findbugs.sourceforge.net.
[2] PMD, 2017. https://pmd.github.io.
[3] Facebook-Infer, 2017. http://fbinfer.com/.
[4] Error Prone, 2017. http://errorprone.info/.
[5] S. Kim, M.D. Ernst, Which warnings should I fix first? in: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, in: ESEC-FSE '07, ACM, New York, NY, USA, 2007, pp. 45–54, doi:10.1145/1287624.1287633.
[6] T. Kremenek, K. Ashcraft, J. Yang, D. Engler, Correlation exploitation in error ranking, in: Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering, in: FSE '04, ACM, New York, NY, USA, 2004, pp. 83–93, doi:10.1145/1029894.1029909.
[7] S. Kim, M.D. Ernst, Prioritizing warning categories by analyzing software history, in: Proceedings of the Fourth International Workshop on Mining Software Repositories, in: MSR '07, IEEE Computer Society, Washington, DC, USA, 2007. pp. 27–. doi: 10.1109/MSR.2007.26.
[8] C. Boogerd, L. Moonen, Prioritizing software inspection results using static profiling, in: 2006 Sixth IEEE International Workshop on Source Code Analysis and Manipulation, 2006, pp. 149–160, doi:10.1109/SCAM.2006.22.
[9] Q. Hanam, L. Tan, R. Holmes, P. Lam, Finding patterns in static analysis alerts: improving actionable alert ranking, in: Proceedings of the 11th Working Conference on Mining Software Repositories, in: MSR 2014, ACM, New York, NY, USA, 2014, pp. 152–161, doi:10.1145/2597073.2597100.
[10] J.R. Ruthruff, J. Penix, J.D. Morgenthaler, S. Elbaum, G. Rothermel, Predicting accurate and actionable static analysis warnings: an experimental approach, in: Proceedings of the 30th International Conference on Software Engineering, in: ICSE '08, ACM, New York, NY, USA, 2008, pp. 341–350, doi:10.1145/1368088.1368135.
[11] S. Heckman, L. Williams, A model building process for identifying actionable static analysis alerts, in: 2009 International Conference on Software Testing Verification and Validation, 2009, pp. 161–170, doi:10.1109/ICST.2009.45.
[12] U. Yüksel, H. Sözer, Automated classification of static code analysis alerts: a case study, in: 2013 IEEE International Conference on Software Maintenance, 2013, pp. 532–535, doi:10.1109/ICSM.2013.89.
[13] S. Heckman, L. Williams, On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques, in: Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, in: ESEM '08, ACM, New York, NY, USA, 2008, pp. 41–50, doi:10.1145/1414004.1414013.
[14] S. Lee, S. Hong, J. Yi, T. Kim, C.-J. Kim, S. Yoo, Classifying false positive static checker alarms in continuous integration using convolutional neural networks, in: 2019 International Conference on Software Testing Verification and Validation, 2019.
[15] A. Aggarwal, P. Jalote, Integrating static and dynamic analysis for detecting vulnerabilities, in: 30th Annual International Computer Software and Applications Conference (COMPSAC'06), 1, 2006, pp. 343–350, doi:10.1109/COMPSAC.2006.55.
[16] C. Csallner, Y. Smaragdakis, T. Xie, DSD-Crasher: a hybrid analysis tool for bug finding, TOSEM '08 17 (2) (2008) 8.
[17] P. Chen, H. Han, Y. Wang, X. Shen, X. Yin, B. Mao, L. Xie, IntFinder: Automatically Detecting Integer Bugs in x86 Binary Program, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 336–345. doi:10.1007/978-3-642-11145-7_26.
[18] B. Liang, P. Bian, Y. Zhang, W. Shi, W. You, Y. Cai, AntMiner: mining more bugs by reducing noise interference, in: Proceedings of the 38th International Conference on Software Engineering, in: ICSE '16, ACM, New York, NY, USA, 2016, pp. 333–344, doi:10.1145/2884781.2884870.
[19] S. Wang, D. Chollak, D. Movshovitz-Attias, L. Tan, Bugram: bug detection with n-gram language models, in: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, in: ASE 2016, ACM, New York, NY, USA, 2016, pp. 708–719, doi:10.1145/2970276.2970341.
[20] B. Johnson, Y. Song, E. Murphy-Hill, R. Bowdidge, Why don't software developers use static analysis tools to find bugs? in: Proceedings of the 2013 International Conference on Software Engineering, in: ICSE '13, IEEE Press, Piscataway, NJ, USA, 2013, pp. 672–681. http://dl.acm.org/citation.cfm?id=2486788.2486877.
[21] J. Nam, S. Wang, Y. Xi, L. Tan, Designing bug detection rules for fewer false alarms, in: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, in: ICSE '18, ACM, New York, NY, USA, 2018, pp. 315–316, doi:10.1145/3183440.3194987.
[22] A. Habib, M. Pradel, Bugram: bug detection with n-gram language models, in: Proceedings of the 33st IEEE/ACM International Conference on Automated Software Engineering, in: ASE 2018, 2018.
[23] FindBugs patterns, 2017. http://findbugs.sourceforge.net/bugDescriptions.html.
[24] D. Kim, J. Nam, J. Song, S. Kim, Automatic patch generation learned from human-written patches, in: Proceedings of the 2013 International Conference on Software Engineering, in: ICSE '13, IEEE Press, Piscataway, NJ, USA, 2013, pp. 802–811. http://dl.acm.org/citation.cfm?id=2486788.2486893.
[25] Q. Hanam, F.S.d.M. Brito, A. Mesbah, Discovering bug patterns in JavaScript, in: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, in: FSE 2016, ACM, New York, NY, USA, 2016, pp. 144–156, doi:10.1145/2950290.2950308.
[26] B. Chen, Z.M.J. Jiang, Characterizing and detecting anti-patterns in the logging code, in: Proceedings of the 39th International Conference on Software Engineering, in: ICSE '17, IEEE Press, Piscataway, NJ, USA, 2017, pp. 71–81, doi:10.1109/ICSE.2017.15.
[27] G. Jin, L. Song, X. Shi, J. Scherpelz, S. Lu, Understanding and detecting real-world performance bugs, in: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, in: PLDI '12, ACM, New York, NY, USA, 2012, pp. 77–88, doi:10.1145/2254064.2254075.
[28] D. Hovemeyer, W. Pugh, Finding bugs is easy, ACM Sigplan Notices 39 (12) (2004) 92–106.

[29] S. Kim, T. Zimmermann, K. Pan, E.J.J. Whitehead, Automatic identification of bug-introducing changes, in: 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06), 2006, pp. 81–90, doi:10.1109/ASE.2006.23.

[30] K. Herzig, S. Just, A. Zeller, It's not a bug, it's a feature: how misclassification impacts bug prediction, in: Proceedings of the 2013 International Conference on Software Engineering, in: ICSE '13, IEEE Press, Piscataway, NJ, USA, 2013, pp. 392–401. http://dl.acm.org/citation.cfm?id=2486788.2486840.

[31] C. Tantithamthavorn, S. McIntosh, A.E. Hassan, A. Ihara, K. Matsumoto, The impact of mislabelling on the performance and interpretation of defect prediction models, in: Proceedings of the 37th International Conference on Software Engineering - Volume 1, in: ICSE '15, IEEE Press, Piscataway, NJ, USA, 2015, pp. 812–823. http://dl.acm.org/citation.cfm?id=2818754.2818852.

[32] M. Tan, L. Tan, S. Dara, C. Mayeux, Online defect prediction for imbalanced data, in: Proceedings of the 37th International Conference on Software Engineering - Volume 2, in: ICSE '15, IEEE Press, Piscataway, NJ, USA, 2015, pp. 99–108. http://dl.acm.org/citation.cfm?id=2819009.2819026.

[33] FeeFin, 2018. http://feefin.github.io.

[34] J. Śliwerski, T. Zimmermann, A. Zeller, When do changes induce fixes? in: Proceedings of the 2005 International Workshop on Mining Software Repositories, in: MSR '05, ACM, New York, NY, USA, 2005, pp. 1–5, doi:10.1145/1082983.1083147.

[35] E. Aftandilian, R. Sauciuc, S. Priya, S. Krishnan, Building useful program analysis tools using an extensible Java compiler, in: 2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation, 2012, pp. 14–23, doi:10.1109/SCAM.2012.28.

[36] Inferbo: infer-based buffer overrun analyzer, 2017. https://research.fb.com/inferbo-infer-based-buffer-overrun-analyzer/.

[37] N. Ayewah, W. Pugh, D. Hovemeyer, J.D. Morgenthaler, J. Penix, Using static analysis to find bugs, IEEE Software 25 (5) (2008) 22–29, doi:10.1109/MS.2008.130.

[38] C. Sadowski, J. Van Gogh, C. Jaspan, E. Söderberg, C. Winter, Tricorder: building a program analysis ecosystem, in: Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on, 1, IEEE, 2015, pp. 598–608.

[39] P. Copeland, Google's innovation factory: testing, culture, and infrastructure, in: 2010 Third International Conference on Software Testing, Verification and Validation, 2010, pp. 11–14, doi:10.1109/ICST.2010.65.

[40] D. Baca, B. Carlsson, L. Lundberg, Evaluating the cost reduction of static code analysis for software security, in: Proceedings of the Third ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, in: PLAS '08, ACM, New York, NY, USA, 2008, pp. 79–88, doi:10.1145/1375696.1375707.

[41] N. Gruska, A. Wasylkowski, A. Zeller, Learning from 6,000 projects: lightweight cross-project anomaly detection, in: Proceedings of the 19th International Symposium on Software Testing and Analysis, in: ISSTA '10, ACM, New York, NY, USA, 2010, pp. 119–130, doi:10.1145/1831708.1831723.

[42] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, D. Poshyvanyk, Detecting bad smells in source code using change history information, in: Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering, in: ASE'13, IEEE Press, Piscataway, NJ, USA, 2013, pp. 268–278, doi:10.1109/ASE.2013.6693086.

[43] Z. Li, Y. Zhou, PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code, in: Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, in: ESEC/FSE-13, ACM, New York, NY, USA, 2005, pp. 306–315, doi:10.1145/1081706.1081755.

[44] B. Livshits, T. Zimmermann, DynaMine: finding common error patterns by mining software revision histories, in: Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, in: ESEC/FSE-13, ACM, New York, NY, USA, 2005, pp. 296–305, doi:10.1145/1081706.1081754.

[45] A. Wasylkowski, A. Zeller, C. Lindig, Detecting object usage anomalies, in: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, in: ESEC-FSE '07, ACM, New York, NY, USA, 2007, pp. 35–44, doi:10.1145/1287624.1287632.

[46] B. Fluri, H.C. Gall, Classifying change types for qualifying change couplings, in: 14th IEEE International Conference on Program Comprehension (ICPC'06), 2006, pp. 35–45, doi:10.1109/ICPC.2006.16.

[47] M. Bruch, M. Monperrus, M. Mezini, Learning from examples to improve code completion systems, in: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, in: ESEC/FSE '09, ACM, New York, NY, USA, 2009, pp. 213–222, doi:10.1145/1595696.1595728.

[48] A.T. Nguyen, M. Hilton, M. Codoban, H.A. Nguyen, L. Mast, E. Rademacher, T.N. Nguyen, D. Dig, API code recommendation using statistical learning from fine-grained changes, in: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, in: FSE 2016, ACM, New York, NY, USA, 2016, pp. 511–522, doi:10.1145/2950290.2950333.

[49] D. Engler, D.Y. Chen, S. Hallem, A. Chou, B. Chelf, Bugs as deviant behavior: a general approach to inferring errors in systems code, in: Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles, in: SOSP '01, ACM, New York, NY, USA, 2001, pp. 57–72, doi:10.1145/502034.502041.

[50] S. Negara, M. Codoban, D. Dig, R.E. Johnson, Mining fine-grained code changes to detect unknown change patterns, in: Proceedings of the 36th International Conference on Software Engineering, in: ICSE 2014, ACM, New York, NY, USA, 2014, pp. 803–813, doi:10.1145/2568225.2568317.