Designing Bug Detection Rules for Fewer False Alarms

ABSTRACT

One of the challenging issues of the existing static analysis tools is the high false alarms rate. To address the false alarm issue, we design bug detection rules by learning from a large number of real bugs from open-source projects from GitHub. Specifically, we build a framework that learns and refines bug detection rules for fewer false positives. Based on the framework, we implemented ten patterns, six of which are new ones to existing tools. To evaluate the framework, we implemented a static analysis tool, FeeFin, based on the framework with the ten bug detection rules and applied the tool for 1,800 open-source projects in GitHub. The 57 detected bugs by FeeFin has been confirmed by developers as true positives and 44 bugs out of the detected bugs were actually fixed.

CCS CONCEPTS

• **Software and its engineering → Automated static analysis**; **Software testing and debugging**;

KEYWORDS

Static bug finder, bug detection rules, bug patterns

ACM Reference Format:

. 2018. Designing Bug Detection Rules for Fewer False Alarms. In *Proceedings of International Conference of Software Engineering, Gothenburg, Sweden, June 2018 (ICSE'18)*, 2 pages.
https://doi.org/10.475/123_4

1 INTRODUCTION

Static bug detection tools has been widely adopted in industry [1–5, 14]. Google has a program analysis ecosystem, TRICORDER [14] and Facebook has its own static analysis tool, Facebook Infer [4]. There are various commercial static analysis tools as well [1–3, 5]. The widespread adoption of static bug detection techniques provides solid evidence that static code analysis is economically beneficial to help developers find real bugs and improve software quality during software development and maintenance phases.

However, false alarms from the static analysis tools prevent developers to actively use them [7, 8, 10–12, 15]. Since the large number of false alarms from static analysis tools causes code inspection overhead so that developers are reluctant to use static analysis tools while developing software products [10]. One of the major reasons that static analysis tools generate too many false alarms is the incomplete rules that are designed with limited buggy cases. For example, when developing bug detection rules, bugs were collected from the small number of projects [9, 13].

ICSE'18, June 2018, Gothenburg, Sweden

ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

To address the false alarm issue, we conducted a case study that investigates whether large scale, iterative rule refinement by using bug histories from *hundreds of open-source projects* is effective. We conjecture the scope of our study as shown in Figure 1. The grey area (**A**) shows all bugs that are not detected and fixed in the world. 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 in previous studies [8, 10], the rest area of **B** contains false positives. While conducting the case study, we implemented our own bug detection tool, FeeFin, that can generate few false alarms as in the circle **C**.

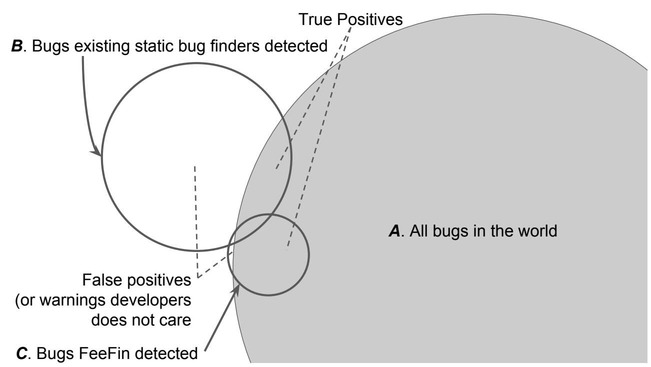

Figure 1: The Scope of Our Case Study

2 APPROACH

To implement FeeFin, we take the following steps as shown in Figure 2.

(1) Manual Patch Analysis: Collect potential bug patterns by manually analyzing patches from open-source projects: We manually analyzed 1,622 patches, whose number of the modified lines are at most five, from four open-source projects, Lucene, Jackrabbit, Hadoop-common, and HBase.
(2) Feedback-based Detection Rule Design: Iteratively refine bug detection rules by using false positives from hundreds of open-source projects after FeeFinwas applied on them.
(3) FeeFin: Implement final detection rules from (2).

These steps are repeated whenever FeeFin generates false positives. In this study, we identified ten bug patterns and refined detection rules based on false positives from FeeFin detection results. The ten bug patterns are as follows: *CompareSameValue, EqualToSameExpression, IllogicalCondition, IncorrectDirectoySlash, IncorrectMapIterator, MissingLForLong, RedundantException, RedundantInstantiation, SameObjEquals, WrongIncrementer.* The detailed descriptions of the bug patterns and rules are available online [6].

3 RESULT

We applied the FeeFin on 599 open-source projects of Apache Software Foundation and Google in GitHub. After finishing the rule

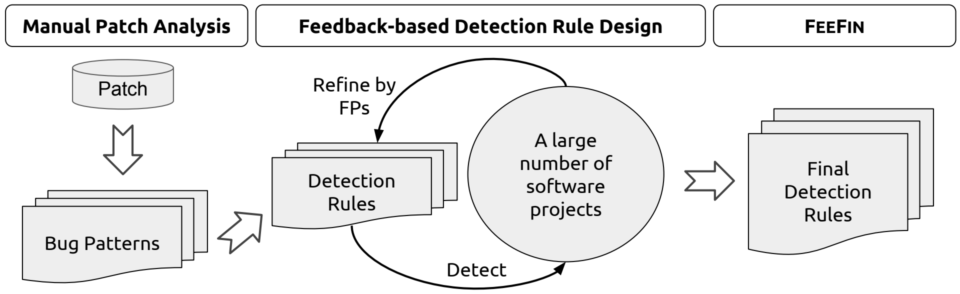

Figure 2: Overview of the FeeFin framework (FPs = false positives)

Table 1: New bugs detected by Snapshot FeeFin. Bug patterns that did not detect any new bugs were excluded. (# DB: detected bugs, # RT: reported bugs to issue tracking systems, # TP: true positives confirmed by developers, # FP: false positives confirmed by developers, # WC: waiting for confirmation, # FX: fixed bugs by developers)

Bug Pattern	# DB	# RT	# TP	# FP	# WC	# Fix
Group 1 (599 Projects)						
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	11	8	0	3	5
Subtotal	**63**	**51**	**27**	**5**	**19**	**20**
Group 2 (948 Projects)						
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 (333 Projects)						
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**	**97**	**57**	**9**	**31**	**44**

refinement, we applied FeeFin on the same 599 projects to check if the rule refinement was correctly conducted by detecting known bugs (i.e., already fixed bugs). FeeFindetected 160 bugs and had only one false positive.

To check if FeeFin can effectively detect unknown bugs, we first collected the new bugs detected by FeeFin on the 599 open-source projects (Group 1). We then applied FeeFin on top 1,281 GitHub open-source projects (Group 2 and Group 3) as in Table 1. FeeFin with ten bug patterns could detect 129 potential bugs. Among them, 97 cases were reported to issue tracking systems and 54 were confirmed by developers as true positives and only 9 were false alarms. The rest cases were still waiting for developer confirmation. Out of the 54 true positives, 40 bugs were already fixed by developers.

REFERENCES

[1] 2017. AppScan. (2017). http://www-03.ibm.com/software/products/en/appscan
[2] 2017. Coverity. (2017). https://scan.coverity.com/
[3] 2017. Fortify. (2017). https://saas.hpe.com/en-us/software/sca
[4] 2017. Inferbo: Infer-based buffer overrun analyzer. (2017). https://research.fb.com/inferbo-infer-based-buffer-overrun-analyzer/
[5] 2017. Klocwork. (2017). http://www.klocwork.com/
[6] 2017. FeeFin. (2017). http://feefin.github.io
[7] A. Aggarwal and P. Jalote. 2006. Integrating Static and Dynamic Analysis for Detecting Vulnerabilities. In *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, Vol. 1. 343–350. https://doi.org/10.1109/COMPSAC.2006.55
[8] Quinn Hanam, Lin Tan, Reid Holmes, and Patrick Lam. 2014. Finding Patterns in Static Analysis Alerts: Improving Actionable Alert Ranking. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR 2014)*. ACM, New York, NY, USA, 152–161. https://doi.org/10.1145/2597073.2597100
[9] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and Detecting Real-world Performance Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 77–88. https://doi.org/10.1145/2254064.2254075
[10] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why Don't Software Developers Use Static Analysis Tools to Find Bugs?. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 672–681. http://dl.acm.org/citation.cfm?id=2486788.2486877
[11] Sunghun Kim and Michael D. Ernst. 2007. Which Warnings Should I Fix First?. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC-FSE '07)*. ACM, New York, NY, USA, 45–54. https://doi.org/10.1145/1287624.1287633
[12] Ted Kremenek, Ken Ashcraft, Junfeng Yang, and Dawson Engler. 2004. Correlation Exploitation in Error Ranking. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering (SIGSOFT '04/FSE-12)*. ACM, New York, NY, USA, 83–93. https://doi.org/10.1145/1029894.1029909
[13] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2013. Detecting Bad Smells in Source Code Using Change History Information. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE'13)*. IEEE Press, Piscataway, NJ, USA, 268–278. https://doi.org/10.1109/ASE.2013.6693086
[14] Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. 2015. Tricorder: Building a Program Analysis Ecosystem. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 598–608. http://dl.acm.org/citation.cfm?id=2818754.2818828
[15] Song Wang, Devin Chollak, Dana Movshovitz-Attias, and Lin Tan. 2016. Bugram: Bug Detection with N-gram Language Models. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, New York, NY, USA, 708–719. https://doi.org/10.1145/2970276.2970341