On the Utility of Marrying GIN and PMD for Improving Stack Overflow Code Snippets

Sherlock A. Licorish  
Department of Information Sciences, University of Otago  
Dunedin, New Zealand  
sherlock.licorish@otago.ac.nz

Markus Wagner  
School of Computer Science, The University of Adelaide  
Adelaide, Australia  
markus.wagner@adelaide.edu.au

ABSTRACT
Software developers are increasingly dependent on question and answer portals and blogs for coding solutions. While such interfaces provide useful information, there are concerns that code hosted here is often incorrect, insecure or incomplete. Previous work indeed detected a range of faults in code provided on Stack Overflow through the use of static analysis. Static analysis may go a far way towards quickly establishing the health of software code available online. In addition, mechanisms that enable rapid automated program improvement may then enhance such code. Accordingly, we present this proof of concept. We use the PMD static analysis tool to detect performance faults for a sample of Stack Overflow Java code snippets, before performing mutations on these snippets using GIN. We then re-analyse the performance faults in these snippets after the GIN mutations. GIN’s RandomSampler was used to perform 17,986 unique line and statement patches on 3,034 snippets where PMD violations were removed from 770 patched versions. Our outcomes indicate that static analysis techniques may be combined with automated program improvement methods to enhance publicly available code with very little resource requirements. We discuss our planned research agenda in this regard.

KEYWORDS
Static analysis, Genetic improvement, Hybridisation

1 INTRODUCTION/MOTIVATION
Websites hosting code online such as Stack Overflow and Hacker-Rank have become the cornerstone for software developers seeking solutions to their coding challenges [9]. These portals are particularly useful as they allow the community to openly critique solutions. While this mechanism is anticipated to help with improving code quality, evidence has shown that many faults remain in code available in online portals [8]. Stack Overflow code, in particular, is extensively reused, at times introducing unsuspecting vulnerabilities in the systems where such code is copied [7].

In helping to remedy faulty code, static analysis is used extensively for understanding the quality of code on Stack Overflow and other online portals. For instance, PMD and SpotBugs have provided insights around how much contributors adhere to code readability, reliability, performance and security rules [4, 8]. These tools may help the software engineering community to quickly understand the quality of code, and whether or not provided solutions conform to coding standards. However, they are ignored at times due to the large amount of rule violations that are returned.

Notwithstanding the challenges apparent in static analysis and automated program improvement techniques, we believe that they may be combined to good effect for helping to improve code available online. In demonstrating this proof of concept, we use the PMD static analysis tool [10] to detect performance faults for a sample of Stack Overflow Java code snippets, before performing mutations on these snippets using GIN [2]. We then re-analyse the performance faults in these snippets after the GIN mutations, observing indicators that static analysis techniques may be combined with automated program repair methods to good effect.

Our contributions in this poster paper are twofold. We demonstrate the utility of static analysis and automated program improvement, and more granular, we provide empirical evidence for how PMD and GIN may be combined for code improvement. We also propose several future research directions, in presenting an agenda for how the two domains considered may be further explored.

2 INITIAL STUDY
2.1 Data
We have used the 8010 snippets that were provided by [8]. The data were extracted from Stack Overflow for 2014, 2015, and 2016 and were said to represent suitably long compilable code from answers where a high level of reuse was evident. Java code was studied due to its popularity, and various forms of preprocessing were done to ensure a reliable dataset. For instance, code snippets were checked for the ‘import’, ‘package’, or ‘class’ keyword and saved unmodified. Code snippets that did not contain these words were encased in a public class structure, and files were saved with the java extension with a unique name.

As GIN’s samplers target the modification of methods, we discard files that do not contain any methods. Moreover, we discard files that do not compile or that contain Java features that are currently not supported by GIN, e.g. certain multi-threading concepts. The resulting 3034 independent files contain 3607 methods. This number is worth highlighting, as most analyses of program improvement spaces to date consider only single programs.

To perform our static analyses, we employ PMD [10]. For Java, it has 324 rules (rulesets/internal/all-java.xml) organised in...
eight sets: Best Practices, Code Style, Design, Documentation, Error Prone, Multi-Threading, Performance, and Security. Analysing the 3034 files resulted in 30,668 PMD violations.

Table 1 shows a summary of the 30,668 PMD rule violations; in total, 135 different rules were violated. Table 2 lists for each of the seven categories the three most frequently violated rules. We spell out the rules to provide the reader with an idea of the types of rules that PMD contains. PMD’s documentation also contains longer descriptions as well as examples and suggestions for mitigation.

### 2.2 Sampling of the Edit Space

To create large numbers of patched code, we use GIN [2], an extensible and modifiable toolbox for search-based experimentation with code. GIN automatically transforms, builds, and tests Java projects. In particular, we employ GIN’s RandomSampler: it randomly generates a patch (which is composed of a given number of individual edits), it applies that patch, then tests the resulting source, and finally returns the result. RandomSampler does not perform a random walk or any iterated search via a sequence of sequential edits, other edit types, and of more complex program transformations, as these are beyond the scope of this short article.

We generate 10,000 patches with one line edit, and 10,000 patches with one statement edit. As the files are relatively small (14.7 lines on average), there is a chance of randomly sampling the very same patch again, such as the deletion of a particular statement. Therefore, instead of 20,000 unique patches, only 17,986 patches of the original code snippets were recorded.

Among these, 5640 (31.4%) are compilable, which is aligned with earlier observations made, e.g., by Langdon and Petke [6], that code is not particularly fragile. Moreover, we observe that the likelihood for a statement-level edit to compile (45.1%) is more than twice as high as it is for a line-level edit (19.8%).

### 2.3 Static Analysis with a Performance Focus

Earlier work singled out code performance issues as serious [8], thus, we now limit our proof of concept investigations to PMD’s 32 performance rules (category/java/performance.xml). According to PMD, these are “Rules that flag suboptimal code”.

Given the union of the original code snippets and the patched ones, PMD finds 3121 performance issues in 1203 of the original 3034 files. Given these 1203 files, we focus on 1185 of them (with a total of 1915 performance issues): these are 237 (with a total of 349 performance issues) of the original code snippets that originally had performance issues, plus their 1185-237=948 patched versions have a cyclomatic complexity of \( <13 \).
3 CONCLUSIONS AND FUTURE DIRECTIONS

Our outcomes show promise, and established the proof of concept: static analysis techniques and automated program improvement methods can be combined to enhance publicly available code. That said, there are several open areas that require further investigation.

Better Static Analysis. (1) Mitigating False Positives and Trivial Warnings: The sheer dimensionality of PMD’s output requires mechanisms to establish potential false positives and eliminate trivial warnings. PMD’s readability and reliability warnings have been show to be accurate [8], and should make a good starting point. However, SpotBugs may offer an alternative body of checks. (2) Improve Parsing: PMD’s effectiveness appears to be severely affected when code does not compile, hence possibly requiring a more robust approach to parsing. (3) Crowd-Sourcing Rules: It may be possible to crowd-source further performance-related PMD rules by mining repositories and question-answering sites like Stack Overflow; for example, Baltes and Wagner [1] provide a potentially useful dataset.

Better Automated Program Improvement. (1) Biased Sampling: As different edits result in different distributions of triggered rules, we conjecture that machine learning models (that take PMD output as input) can be used to bias the patch generation towards desired code properties. (2) Better Code Transformations: While the traditional operators for search-based program modification (e.g., copy, delete, replace, and swap) seem inadequate at first to address the PMD rules violations that we have encountered here (see Section 2.3), we can imagine scenarios where a single Replace or Delete can resolve a violation, e.g., in certain cases of AdEasyFromString and UseIndexOfChar. For other violations, e.g., OptimizetabletoArrayCall and AvoidInstantiatingObjectInLoops, however, custom transformations appear to be necessary – possibly, insights from the well-established field of code refactoring [3] can be beneficial. (3) Further Code Properties: While our study here focuses almost exclusively on performance-related improvements, it is straightforward to change the focus to other sets of PMD rules, to other non-functional properties, and even to functional properties.

REFERENCES

[1] Sebastian Baltes and Markus Wagner. 2020. An Annotated Dataset of Stack Overflow Post Edits. In Genetic and Evolutionary Computation Conference Companion. ACM, 1923–1925. https://doi.org/10.1145/3377929.3398108
[2] Alexander E. I. Brownlee, Justyna Petke, Brad Alexander, Earl T. Barr, Markus Wagner, and David R. White. 2019. Gin: Genetic Improvement Research Made Easy. In Genetic and Evolutionary Computation Conference (GECCO ’19). ACM, 985–993. https://doi.org/10.1145/3321707.3321841
[3] Martin Fowler. 2018. Refactoring: improving the design of existing code. Addison-Wesley Professional.
[4] Antonios Gkortzis, Daniel Feitosa, and Dionysis D. Spinellis. 2019. A Double-Edged Sword? Software Reuse and Potential Security Vulnerabilities. In ICSR.
[5] Nicolas Harrand, Simon Allier, Marcelino Rodriguez-Cancio, Martin Monper- rus, and Benoit Baudry. 2019. A journey among Java neutral program variants. Genetic Programming and Evolvable Machines 20, 4 (2019), 531–580. https://doi.org/10.1007/s10710-019-09355-3
[6] William B. Langdon and Justyna Petke. 2017. Software is Not Fragile. In First Complex Systems Digital Campus World E-Conference. Springer, 203–211.
[7] Adriaan Lotter, Sherlock A. Licorish, Bastin Tony Roy Savarimuthu, and Sarah Meldrum. 2018. Code Reuse in Stack Overflow and Popular Open Source Java Projects. In 20th Australasian Software Engineering Conference (ASWEC). 141–150. https://doi.org/10.1109/ASWEC.2018.00027
[8] Sarah Meldrum, Sherlock A. Licorish, Caitlin A. Owen, and Bastin Tony Roy Savarimuthu. 2020. Understanding stack overflow code quality: A recommendation of caution. Science of Computer Programming 199 (2020), 102516.

Listing 1: Snippet C402521.java: SwapStatement(153,154) swaps the body of the loop with one of its statements, thus mitigating the AvoidArrayLoops violation. Shown is only the relevant part of the code.

```java
// original
24  for (int i = c+1; i < nums.length; i++) {
25      b[j] = nums[i];
26      j++;
27  }
// after applying the patch
24  for (int i = c+1; i < nums.length; i++)
25      b[j] = nums[i];
```
### Table 3: PMD finds 349 performance-related rule violations in the 237 original code snippets. Examples of rule instantiations are shown as ...

| rule                              | count | description                                                                 |
|-----------------------------------|-------|-----------------------------------------------------------------------------|
| UseStringBufferForStringAppends   | 118   | Prefer StringBuilder (non-synchronized) or StringBuffer (synchronized) over += for concatenating strings. |
| AddEmptyString                    | 54    | Do not add empty strings.                                                   |
| AppendCharacterWithChar           | 35    | Avoid appending characters as strings in StringBuffer.append.               |
| RedundantFieldInitializer         | 23    | Avoid using redundant field initializer for <i>.                             |
| AvoidInstantiatingObjectsInLoops  | 19    | Avoid instantiating new objects inside loops.                              |
| AvoidArrayLoops                   | 19    | System.arraycopy is more efficient.                                          |
| UseIndexOfChar                    | 12    | String.indexOf(char) is faster than String.indexOf(String).                 |
| StringInstantiation               | 11    | Avoid instantiating String objects; this is usually unnecessary.             |
| InefficientStringBuffering        | 9     | Avoid concatenating nonliterals in a StringBuffer/StringBuilder constructor or append(). |
| AvoidUsingShortType               | 8     | Do not use the short type.                                                  |
| TooFewBranchesForASwitchStatement | 7     | A switch with less than three branches is inefficient, use a if statement instead. |
| IntegerInitialization             | 6     | Avoid instantiating Integer objects. Call Integer.valueOf() instead.        |
| UselessStringValueOf              | 6     | No need to call String.valueOf to append to a string.                       |
| ConsecutiveAppendsShouldReuse     | 4     | StringBuffer (or StringBuilder).append is called consecutively without reusing the target variable. |
| InefficientEmptyStringCheck       | 4     | String.trim().length() == 0 / String.trim().isEmpty() is an inefficient way to validate a blank String. |
| StringToString                    | 3     | String.indexOf(char) is faster than String.indexOf(String).                 |
| StringInstantiation               | 3     | Avoid instantiating String objects; this is usually unnecessary.             |
| InefficientStringBuffering        | 2     | StringBuffer (or StringBuilder).append is called consecutively without reusing the target variable. |
| SimplicityStartsWith              | 2     | This call to String.startsWith can be rewritten using String.charAt(0).      |
| ConsecutiveLiteralAppends         | 2     | StringBuffer (or StringBuilder).append is called <3> consecutive times with literals. |
| OptimizableToArrayCall            | 2     | This call to Collection.toArray() may be optimizable.                       |
| BooleanInstantiation              | 1     | Avoid instantiating Boolean objects; reference Boolean.TRUE/Boolean.FALSE or call Boolean.valueOf() instead. |

### Table 4: PMD finds 1566 performance-related issues in the patched code: 739 in the 453 compilable snippets and 827 in the 495 snippets that do not compile. The leftmost column shows the statistics for the original 237 code snippets from Table 3 as a reference.

| original code snippets | patched files that compile | patched files that do not compile |
|-----------------------|----------------------------|----------------------------------|
|                       | total                      | ReplaceStatement | SwapStatement | total                      | ReplaceStatement | SwapStatement |
|                       | CopyleLine | DeleteLine | ReplaceStatement | SwapStatement | CopyleLine | DeleteLine | ReplaceStatement | SwapStatement | CopyleLine | DeleteLine | ReplaceStatement | SwapStatement | CopyleLine | DeleteLine | ReplaceStatement | SwapStatement |
| 118 UseStringBufferForStringAppends | 247 | 30 | 23 | 17 | 30 | 63 | 20 | 35 | 29 | 262 | 32 | 14 | 7 | 12 | 114 | 18 | 50 | 15 |
| 54 AddEmptyString | 121 | 2 | 14 | 10 | 15 | 22 | 23 | 20 | 15 | 157 | 4 | 16 | 14 | 42 | 10 | 37 | 18 |
| 35 AppendCharacterWithChar | 100 | 5 | 18 | 9 | 14 | 16 | 14 | 9 | 15 | 102 | 11 | 3 | 2 | 15 | 34 | 20 | 15 | 15 |
| 23 RedundantFieldInitializer | 55 | 2 | 4 | 3 | 6 | 10 | 6 | 12 | 12 | 36 | 2 | 1 | 0 | 3 | 9 | 10 | 5 | 6 |
| 19 AvoidInstantiatingObjectsInLoops | 26 | 2 | 3 | 3 | 4 | 1 | 4 | 1 | 7 | 2 | 74 | 5 | 3 | 10 | 18 | 3 | 16 | 16 |
| 19 AvoidArrayLoops | 27 | 3 | 2 | 2 | 2 | 7 | 7 | 1 | 3 | 45 | 2 | 5 | 0 | 2 | 16 | 5 | 9 | 6 |
| 12 UseIndexOfChar | 24 | 2 | 1 | 0 | 2 | 12 | 5 | 0 | 2 | 20 | 0 | 0 | 1 | 1 | 13 | 1 | 2 | 2 |
| 11 StringInstantiation | 28 | 3 | 2 | 2 | 5 | 5 | 7 | 2 | 2 | 19 | 0 | 1 | 0 | 0 | 8 | 6 | 3 | 1 |
| 9 InefficientStringBuffering | 9 | 2 | 0 | 2 | 0 | 2 | 1 | 1 | 1 | 11 | 0 | 1 | 0 | 1 | 5 | 1 | 2 | 1 |
| 8 AvoidUsingShortType | 12 | 0 | 2 | 2 | 4 | 0 | 6 | 3 | 7 | 25 | 1 | 5 | 0 | 0 | 9 | 4 | 6 | 0 |
| 7 TooFewBranchesForASwitchStatement | 12 | 0 | 0 | 0 | 1 | 3 | 4 | 2 | 2 | 18 | 0 | 2 | 0 | 0 | 3 | 4 | 4 | 5 |
| 6 IntegerInitialization | 8 | 0 | 0 | 2 | 1 | 1 | 1 | 2 | 1 | 14 | 1 | 0 | 0 | 1 | 7 | 3 | 2 | 0 |
| 6 UselessStringValueOf | 14 | 0 | 1 | 0 | 1 | 8 | 1 | 1 | 2 | 7 | 0 | 3 | 0 | 0 | 4 | 0 | 0 |
| 4 ConsecutiveAppendsShouldReuse | 12 | 1 | 2 | 0 | 5 | 2 | 2 | 0 | 0 | 18 | 5 | 2 | 0 | 0 | 5 | 1 | 4 | 1 |
| 4 InefficientEmptyStringCheck | 6 | 0 | 1 | 0 | 2 | 0 | 0 | 0 | 3 | 3 | 0 | 0 | 0 | 0 | 2 | 1 | 0 | 0 |
| 3 StringToString | 3 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 2 | 0 | 0 | 0 | 0 | 2 | 0 | 0 |
| 3 InsufficientStringBufferDeclaration | 7 | 0 | 2 | 1 | 0 | 0 | 1 | 2 | 1 | 5 | 0 | 1 | 0 | 0 | 4 | 0 | 0 |
| 3 SimplifyStartsWith | 8 | 0 | 0 | 2 | 3 | 0 | 0 | 2 | 1 | 3 | 0 | 0 | 2 | 0 | 0 | 0 | 1 | 0 |
| 2 ConsecutiveLiteralAppends | 3 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 1 | 2 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| 2 OptimizableToArrayCall | 4 | 0 | 0 | 0 | 0 | 2 | 2 | 0 | 0 | 2 | 0 | 0 | 0 | 2 | 0 | 0 |
| 1 BooleanInstantiation | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |

349 total rule violations

number of patched files: 453 28 42 36 54 82 67 71 73 495 47 39 16 57 31 49 298 85 156 87
[9] Sarah Meldrum, Sherlock A. Licorish, and Bastin Tony Roy Savarimuthu. 2017. Crowdsourced Knowledge on Stack Overflow: A Systematic Mapping Study. In 21st International Conference on Evaluation and Assessment in Software Engineering (EASE’17). ACM, 180–185. https://doi.org/10.1145/3084226.3084267

[10] PMD. 2021. PMD 6.41.0. https://pmd.github.io/latest/pmd_rules_java.html [Online, accessed on 22 January 2022].