Bug Detection: A Comparison of PMD, Code Naturalness, and Commit Verification

V. Patwardhan, W. Young
School of Engineering and Applied Science
The University of Virginia
Rice Hall, 85 Engineer's Way
Charlottesville, Virginia, USA
{vvp2da, wty5dn}@virginia.edu

Abstract *- As the size, complexity, and ubiquity of software systems increase, identifying and patching buggy code remains the principal means of ensuring system security and stability. Entropy has emerged as a useful statistic by which to measure the naturalness—and by extension, correctness—of code, though determining the entropy of code in a software system provides little context within which one can discern the nature of a potential bug. PMD, a widely-used and readily available bug detection tool, provides an easy mechanism for fine-tuning bug detection and substantial context for reported violations, making it a good candidate to capture the context lost in entropic scoring. We investigate the claim that by combining entropic scores with matching PMD violation context, developers can better understand—and thus prioritize and patch—buggy code. We evaluate a corpus of four software projects from the Defects4j repository using PMD and existing entropy information. We find that while relative entropy scores are preserved between buggy and nonbuggy code, the nature of PMD prevents it from being the sole source of bug context, though steps can be taken to expand its usefulness.*

Index Terms - bug detection, entropy, PMD, software engineering

1 INTRODUCTION

Buggy code is a serious problem in the business of designing and distributing software products, accounting for up to 40% of system failures [6]. Increasing size and complexity in software systems comes at an interesting juncture in the computing environment, when the ubiquity of software systems in everyday products is exceeded only by the collective demand for the security and stability of these systems. It comes as no surprise, then, that bug detection is an immensely important research topic.

For as long as software has existed, so too have efforts to identify, understand, and patch bugs. The time and resources invested in these efforts is evident in the body of bug-detection tools and methods available for use by programmers throughout the software industry. In recent years, the bug detection landscape has become saturated with a myriad of tools and techniques designed to identify control-flow, stylistic, logical, and syntactical errors, though these tools come with varying tradeoffs with respect to runtime, compatibility, and accuracy.

Though some bug detection techniques are proprietary solutions developed at major software companies, many are widely available for use by programmers of all skill levels and may even come prepackaged in popular integrated development environments (IDE), such as FindBugs™ and PMD. Research tends to suggest, however, that no single bug detection system is yet capable of maximizing the three pillars of software testing—ease of use, compatibility, and accuracy—forcing developers to strike an imperfect balance among the three based on the availability of development time and resources [6].

Entropy has recently emerged as a promising means for identifying and prioritizing potentially buggy lines of code, with the potential to save developers significant time and energy. Entropic scoring is an intuitive means of detecting irregularities in code, grounded in the notion that correct code is highly repetitive [1].

Language models similar to those utilized in the areas of natural language processing (NLP) and text mining are capable of tokenizing and classifying code in order to determine the naturalness of code; this notion of *naturalness* is codified in the form of an *entropy score* associated with each line of code in a software system. Prior work [2] showed that unnatural code (exhibiting a high entropy value) is more likely to be wrong, advancing the notion that entropy is a useful means by which to measure the bugginess of code.

While entropic scoring may be effective at flagging potentially buggy code, it provides little in the way of context and therefore must be used in conjunction with other bug-detection tools. Entropy serves as an efficient means to triage and prioritize review efforts based on real-world development constraints, leading developers directly to code flagged by at least one other bug detection test, sorted in order of decreasing entropy. Intuitively, combining the output of multiple bug-detection tools to produce a single doubly-verified compilation of bug data sorted by entropy should serve to increase the fidelity of bug detection and output highly actionable buggy code. We seek to verify this intuition.

In order to test our claim, we evaluated a corpus of four software projects from the Defects4j repository. We began by running PMD on all four selected projects, resulting in a large body of PMD rule violations, which we interpreted throughout this work as potential bugs. We then proceeded to compare the PMD output against the entropy data acquired from the experiments of Ray & Hellendoorn *et al* [2]. Matches between the two data sets were checked against a compilation of manually verified bugs in an effort to eliminate false-positives. We also recorded information pertaining to the entropy and context of each matched bug. To ensure thorough analysis of the data, we elected to evaluate the matches between the manually verified bugs and PMD rule violations in the same manner as the previously-described approach.

Our results show that while relative entropy between buggy and nonbuggy code is preserved, the effectiveness of comparing entropy information with PMD rule violations is limited by our ability to test an entropy dataset with sufficient coverage. We finish our assessment of the test data with an analysis of each cross-comparison. The paper concludes with a discussion of potential threats to validity and exploration of future work.

2 BACKGROUND

2.1 Defects4j

Defects4j is a database of existing faults in select Java projects. It was created with the purpose of providing these reproducible, annotated bugs to the public in order to advance software testing research [5]. Defects4j contains a corpus of projects and their respective data. Each of the five projects has a codebase divided into monthly snapshots. These snapshots contain the state of the entire project at a specified point in time. All code files as they existed at the time of capture are present in each snapshot directory. Each project also contains a large .csv file of existing faults. The process by which this bug data was obtained is as described in [2] and summarized below.

Each commit's description in a project repository is analyzed to determine if it is a bug fix. This information is then used in conjunction with git-blame to determine which commit introduced the buggy code. The line number of the bug and file name it appears in for each of these commits are stored. Finally, the bug becomes associated with the chronologically-following snapshot. This results in a large data set with numerous columns of information. However, all key pieces of information can be gleaned from purely the snapshot data, so the stored data from the two commits is discarded for the purposes of this work.

Defects4j makes five statements about the bugs listed in the repository:

1. Each bug was fixed in a single commit.
2. This commit only contained the fix for a given bug, and did not include any irrelevant code changes or refactoring.
3. Each bug was fixed by modifying a line or set of lines in the java source code as opposed to configuration or test files.
4. Each bug was confirmed as fixed.
5. At least one test in the test set for the project which previously failed prior to the commit passed after implementing changes to the source code.

2.2 PMD

PMD is a general-purpose source code analyzer capable of detecting common programming flaws across a wide range of programming languages, including popular languages such as Java, C, C++, and Python. PMD is available for use by default in integrated development environments (IDE) like Eclipse, JEdit, NetBeans, TextPad, and Emacs, resulting in its status as a widely available, powerful tool for conducting bug detection. PMD possesses the attractive feature of enabling users to rapidly write and deploy custom rulesets [4].

Due to its wide availability and native integration in software development frameworks, we make two assumptions:

1. The rulesets are diverse
2. The rulesets are correct

Our first assumption follows naturally from the proliferation of PMD throughout the software development landscape—as the user base of a tool expands, one expects that the number of contributions to the tool base will increase in turn. Our second assumption is related to the first, but distinct enough to warrant attention. It may not necessarily follow that all community-developed rulesets are "correct." By correctness, we refer to the idea that PMD accurately identifies all instances of a given rule violation. In such instances, failure could result from a rule that fails to capture the full scope of a given violation.

The power of PMD rests in its customizability. When running tests, users are able to select from a range of prepackaged rulesets in addition to any number of custom-defined rulesets. Since PMD runs on raw source code, the structure and syntax of PMD rules follow relatively plain English. In order to minimize stylistic violations yet capture as many serious rule violations as possible, we implemented a custom ruleset, imaginatively named *myRules.xml*. The ruleset tests for a number of violations pertinent to our analysis of code bugginess. Figure 1 (below) details the coverage of *myRules.xml*. Notably, we excluded rulesets concerned with reporting unnecessary and unused code, comment-style violations, and variable naming conventions (length, etc) in order to suppress superfluous violations in the output data.

When configured properly, PMD takes as input a source file or directory, an output format (.csv, .txt, etc), any number of rulesets, and various flags related to its sensitivity and output verbosity. We configured PMD to consider all possible violation priorities (1 to 5) and provide the following output in a .csv file, one per project:

1. filename
2. line number
3. issue number (resets to 0 for each file)
4. ruleset (of violation)
5. rule violation
6. description of rule violation

2.3 Entropy

Our end-goal with the entropy data is to correlate a high entropy value with a buggy line of code. The reasoning behind this is simple: Large code bases more than likely have many instances of similar code. Common coding constructs such as for-loops and declarations appear throughout and therefore have a low entropy value. It is less likely that something so commonplace in the code

	Rulesets included in *myRules.xml*
Ruleset	Description
Basic	Collection of good programming practices
Clone	Collection of rules that find questionable usages of clone()
Empty	Searches for empty statements of any kind
Finalizers	Catches problems that may occur with finalizers
J2EE	Rules to detect J2EE usage errors
Controversial	Rules that many programmers may consider controversial
JavaBeans	Catches instances of bean rules not being followed
JUnit	Rules that deal with problems encountered in JUnit tests
Logging - Jakarta Commons	Finds questionable uses of the Jakarta Commons framework
Logging - Java	Finds questionable uses of the Java logger
Strict Exceptions	Guidelines about throwing and catching exceptions
String & String Buffer	Rules that deal with manipulation of the class String
Security Code Guidelines	Rules that perform comparison against security guidelines
Type Resolutions	Rules to resolve Java class files for comparison

Figure 1

actually contains a bug. On the other hand, a particular method implementation, or even more specifically, a single line of code, has the potential to differ greatly from the rest of the code. If it appears less frequently (or even just once), it stands to reason that it would be less likely to be caught. Therefore, we postulate that less frequently appearing lines of code, which correspondingly exhibit high entropy values, will be buggy more often than those lines with lower entropy values [2].

3 METHODOLOGY

3.1 Running PMD

We began the experiment by running PMD on the raw Java source code files in each of four Defects4j projects: *joda-time*, *jfreechart, commons-math,* and *commons-lang.* We configured PMD to capture all potential priorities of rule violations (1 to 5) with the understanding that *myRules.xml* would suppress many unnecessary violations. The PMD output from each run was directed to a .csv file for parsing, pruning, and analysis, along with all pertinent PMD context (refer to §2.2 for details).

3.2 Pruning

Analyzing the Defects4j data set showed us that it includes a surprising number of duplicates. The same buggy line appeared in multiple entries across multiple snapshots. A single line could appear upwards of thirty times in the data set. It is possible that each of these referred to a unique bug, but in lacking context regarding the bugs, our ability to reason about the data was limited.

Our response to this dilemma was to test two separate configurations. The two models under which we can perform comparison are the *snapshot sensitive* model ("unique snapshots") and the *snapshot-agnostic* model. The snapshot sensitive model is, as the name suggests, sensitive to snapshots. According to this model, each entry in the Defects4j data set is assumed to be a valid unique bug. It logically follows that we expect a larger volume of matches when the comparison is run.

The snapshot-agnostic model instead ignores all snapshot context and differentiates bugs by their file and line number only. This eliminates many of the potential duplicates we observed in the data, thus we expect to observe fewer matches when this model is used for comparison.

Implementing this pruning model was relatively straightforward—we merely needed to scrub snapshot context from the full pathname of each file using a basic *split* manipulation available in most scripting languages' string libraries (for example, `str.split('token')` in Python).

3.3 Parsing

Once pruning was completed, we were left with three distinct data sets—PMD output, entropy information, and the Defects4j snapshot data. Each dataset possessed two common fields, pathname and line number, therefore making it quite easy for us to reach a decision regarding the fields on which we would perform our analysis. Considering that we needed to perform matching on upwards of one MLOC, no analysis would have been possible without finding an efficient and effective method to parse the data. To ensure our parsing model was maximally unbiased, we independently developed two parsers, one in Perl and one in Python. Testing showed that the parsers produced identical statistical output when run on the same datasets.

We took extensive advantage of the languages' dictionary implementations to optimize matching performance. Entries in the dictionary appeared in the form:

```
{key(path), value(line)}
```

with a slight modification to account for the extra context associated with each PMD entry:

```
{key(path), value([line, violation])}
```

where *path* refers to the full pathname of the file (including or excluding snapshot context, depending on the configuration). This configuration allowed for fast and simple matching among the data sets, and enabled us to capture the PMD violation context associated with each matched bug.

Unique Snapshot							
Project	Entropy Dataset				PMD Dataset		Snapshot Dataset
	Buggy		Nonbuggy				
	Lines Interpreted	Average Entropy	Lines Interpreted	Average Entropy	Violations Reported	Types of Violations	Manually Verified Bugs
joda-time	32	5.9786	2436	4.8089	530344	65	11667
jfreechart	7	4.5309	2188	4.0892	124451	61	5345
commons-math	32	5.336	884	4.9803	202142	61	128122
commons-lang	33	5.2477	4832	4.0226	339630	67	42028

Snapshot-agnostic							
Project	Entropy Dataset				PMD Dataset		Snapshot Dataset
	Buggy		Nonbuggy				
	Lines Interpreted	Average Entropy	Lines Interpreted	Average Entropy	Violations Reported	Types of Violations	Manually Verified Bugs
joda-time	32	5.9786	2436	4.8089	99154	65	2036
jfreechart	7	4.5309	2188	4.0892	34844	61	1996
commons-math	32	5.336	884	4.9803	54722	61	39430
commons-lang	33	5.2477	4832	4.0226	55229	67	10423

Figure 2

In addition to pathname, line number, and violation context, we also recorded (in comparisons against the entropy data) the average entropy of both buggy and nonbuggy lines, the true match percentage of buggy lines, the number of lines verified as buggy *without* any corresponding matches, and the most commonly encountered PMD violations. In the next section, we delve into the relationships among the three datasets as well as their implications.

4 DISCUSSION

4.1 Bug Breakdown

An overview of the data sets is represented in Figure 2. One of the few points to note about this information is that it shows a noticeably higher entropy for buggy lines than for nonbuggy lines of code. However, the sample size of the number of buggy lines with corresponding entropy data is small, so the average may be misleading. This figure also illustrates the difference between the snapshot-agnostic model and the snapshot-sensitive model. The agnostic model significantly reduced the problem size by removing likely duplicate bugs from the data. Therefore, with this model, if an entry from the Defects4j data matches something from another source, we have a higher confidence that it is a legitimate match.

4.2 Entropy vs PMD

The entropy data was combined in the parser with the output from PMD to yield the results in Figure 3a. The first intuition about these results is that they are sparse. The number of matches was not nearly significant enough to draw any meaningful conclusions from. This is attributed to the sparseness of the entropy data

Entropy Dataset vs. PMD Dataset					
	Buggy		Nonbuggy		
Project	Matched Lines	Average Entropy	Matched Lines	Average Entropy	Match Percentage
joda-time	2	5.6967	292	4.68	6.25%
jfreechart	0	0	0	0	0%
commons-math	2	5.9052	35	5.0726	6.25%
commons-lang	0	0	498	3.284	0.00%

(a)

PMD Dataset vs. Snapshot Dataset				
Unique Snapshot				
Project	Matched Lines	Match Percentage	Verified Bugs w/o PMD Violation	Nonbuggy PMD Violations
joda-time	397	3.40%	11270	529947
jfreechart	47	0.88%	5298	124404
commons-math	6620	5.17%	121052	195522
commons-lang	6487	15.43%	35541	333143
Snapshot-agnostic				
Project	Matched Lines	Match Percentage	Verified Bugs w/o PMD Violation	Nonbuggy PMD Violations
joda-time	182	8.94%	1854	98972
jfreechart	102	5.11%	1894	34742
commons-math	5091	12.91%	34339	49631
commons-lang	2319	22.25%	8104	52910

(b)

Entropy Dataset vs. Snapshot Dataset					
	Buggy		Nonbuggy		
Project	Matched Lines	Average Entropy	Matched Lines	Average Entropy	Match Percentage
joda-time	27	6.014	25	5.0486	84.38%
jfreechart	0	0	0	0	0%
commons-math	10	6.6514	66	5.5123	31.25%
commons-lang	1	8.2962	213	4.1908	3.03%

(c)

Figure 3

set to begin with. That shortcoming aside, the little data we do have shows a correlation between noticeably higher entropy and buggy lines of code.

4.3 Snapshot vs PMD

In our second test, the Defects4j snapshot data was compared with the results from PMD in a similar fashion as that described in §4.2. These results are tabulated in Figure 3b. The first of many things to note is that the sample size of matches is much larger than with Entropy. The Defects4j data set and the PMD data set are significantly larger than the entropy data set, so it is expected that there are significantly more matches. Taking a look at the "Nonbuggy PMD Violations" column, you can see the number of violations bugs caught by PMD that were supposedly missing in the Defects4j data. However, we believe this stems from the nature of PMD itself. As mentioned previously, most of PMD's many rulesets catch stylistic and programming best-practice errors which would not be represented in the Defects4j data, so it is not unusual to see a large number in that column.

4.4 Entropy vs Snapshot

Figure 3c shows the results of the comparison between the entropy data and the Defects4j data. Relatively speaking, these data are much better than those from the comparison in Figure 3a between the entropy data and the PMD data. The difference here can be attributed to the difference between Defects4j data and PMD data. PMD finds more stylistic errors, some involving loop simplification and brace placement. Entropy's purpose is to show how unique a line of code is, so it is understandable that our entropy data set did not have matching entries for lines with stylistic errors that (depending on the programmers' unique styles) may appear all throughout the project.

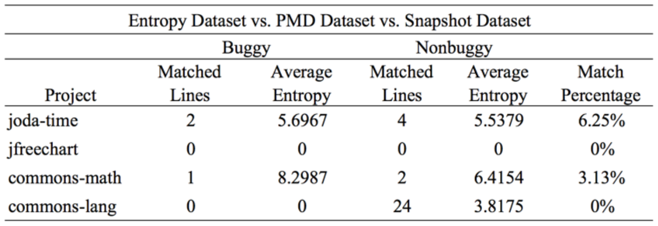
Taking a look at the entropy of the matched lines, a distinct increase can be seen for buggy lines of code across all of the projects. Two of our most significant results, the nonbuggy matches in *commons-lang* from Figures 3b and 3c, both corroborate the idea that nonbuggy lines are correlated with a lower entropy.

4.5 Snapshot vs Entropy vs PMD

Comparing each data set that was available to us at once gave us the results in Figure 4. As you can see, there were very few matches across all of the data sets. The single largest limiting factor in this case was certainly the intersection of the entropy and PMD data sets. This type of holistic comparison was not fruitful, and it was unadvised to draw conclusions from these results. With an expanded entropy data set, the intersection of all data sets might provide more useful results.

4.6 PMD Violations

PMD was run on all of the files in the Defects4j repository with the ruleset we outlined previously. Figure 5 shows the ten most frequent violations caught by PMD. The most frequent violation by a factor of five was *JUnitAssertionsShouldIncludeMessage*. This violation and many others in this list are stylistic errors or style guidelines that do not affect the functionality or code flow of the program. This violation in particular likely generated so many results due to an API update to *assert()* while these projects were still under development. The developers likely made a conscious decision to ignore that error, but this is purely speculation.



Entropy Dataset vs. PMD Dataset vs. Snapshot Dataset

	Buggy		Nonbuggy		
Project	Matched Lines	Average Entropy	Matched Lines	Average Entropy	Match Percentage
joda-time	2	5.6967	4	5.5379	6.25%
jfreechart	0	0	0	0	0%
commons-math	1	8.2987	2	6.4154	3.13%
commons-lang	0	0	24	3.8175	0%

Figure 4

5 THREATS TO VALIDITY

5.1 Defects4j Data

The Defects4j annotated bug data is potentially misleading. Each listed bug is assumed to be true. However, it is quite possible that some bug entries present across snapshots are not in fact unique. This was the best data set available to us at the time of this experiment, but this threat to validity limits the conclusions that we can draw from our results.

This data also lacks one other quality that potentially threatens the validity of our results. The form of the data we received does not contain any information regarding the details of the bug itself. The bug referred to by an entry in the data set could be anything from a misplaced semicolon to improper inheritance and everything in between. The results we receive from PMD are very specific and detail the context of the reported bug. Because we lack this same context in the Defects4j data, we cannot be absolutely certain that an entry and its match in the PMD output actually refer to the same bug.

5.2 Entropy Data

The size of the entropy dataset pales in comparison to the size of the Defects4j data and the PMD data. Given the smaller pool of entries, it is reasonable to assume that fewer matches than may necessarily exist were found between this data set and others. Moreover, with such a small number of matches, it is reasonable to question whether a meaningful and strong conclusion can be drawn from our data.

Most Frequent PMD Violations (all projects)				
Rule Violation	Occurences	Priority	Ruleset	Description
JUnitAssertionsShouldIncludeMessage	4060	3	JUnit	JUnit assertions should use the three-argument version of assertEquals()
EmptyCatchBlock	802	3	Empty Code	Avoid swallowing exceptions without acting upon them
UseAssertTrueInsteadOfAssertEquals	745	3	JUnit	Use True/False rather than Equals when asserting a boolean literal
OnlyOneReturn	683	3	Controversial	A method should have only one exit point occuring after all statements
JUnitTestContainsTooManyAsserts	598	3	JUnit	Too many asserts are indicative of an overly-complex test
BeanMembersShouldSerialize	533	3	JavaBeans	If a class is a bean it needs to be serializable
AvoidLiteralsInIfCondition	391	3	Controversial	Avoid using hard-coded literal in conditional statements
AvoidDuplicateLiterals	292	3	String	Code containing duplicate String literals can be improved with constants
CollapsibleIfStatements	90	3	Basic	Two consecutive "if" statements can be consolidated with boolean
NullAssignment	62	3	Controversial	Assigning a "null" to a variable outside of declaration is bad form

Figure 5

The entropy data lacked one additional quality which hurt our ability to back many conclusions drawn. Unlike the Defects4j data, the entropy data lacked snapshot context. We were only able to match using two parameters, the filename and line number of the bug, and as a result introduced the potential for error. Without snapshot context, it remains entirely possible that the bug referred to by the entropy data is different from that present in Defects4j.

6 RELATED WORK

Ray & Hellendoorn *et al* tackle a similar problem as us in the paper "On the 'Naturalness' of Buggy Code" [2]. This is motivated by the idea that real working software systems on the scale of thousands of lines of code are both natural and highly repetitive. This same train of thought can be extended to then investigate the "naturalness" of a piece of code and determine if the code becomes increasingly suspect as it becomes more unnatural. The authors use the same repository of bugs for their comparison and calculate the naturalness, or entropy, of the code using their own method. Our goals are similar in that we both try to attribute a higher entropy, or unnaturalness, to a buggy line of code. The results of their study indicated that entropy could be used to guide other bug detecting software in the right direction, and more notably that the entropy of a line of code decreased by a statistically significant value when its contained bug was fixed.

Rutar *et al* [3] performs a comparison of a number of the bug finding tools available for Java. This was done in order to determine the relative effectiveness of each of these tools, but was done in a manner different from ours. Their comparison was on individual buggy lines of code, blatantly obvious errors, of differing severity. The primary difference between their study and ours is they lack anything similar to our Defects4j data. They did not annotate their list of bugs to determine which results were false positives and false negatives, and instead compared the output of one analyzer to another to determine precision. PMD was included in this study and they found that it performed reasonably. It took an above-average length of time to run in order to find an above-average number of violations. However, these violations were not as diverse as those generated by other tools, reinforcing our claim that PMD's default rulesets are too specific for our purposes.

Lu *et al* [6] followed a path similar to ours when they created their bug-detection benchmark, *Bugbench*. As we've stressed numerous times until now, one of the most important aspects of this study and many others on bug detection is the data set on which you operate. Bugbench realizes this and provides a standard means of measuring and evaluating the performance of bug testing software. It contains a bank of bugs and their contexts across a number of projects across the web, and the creators plan to create something similar to a web crawler to automatically fetch more.

7 FUTURE WORK

We identify three qualities that we desire in a data set if we are to continue exploring this hypothesis. One of the main differences between our current data and desired data is the size of the entropy dataset. We would benefit from significantly more data points when it comes to the entropy of buggy and nonbuggy lines in order to draw adequate conclusions—perhaps an entropy score corresponding to every line in the repository. In the same vein, we also require context information surrounding the bugs in question in order to definitively confirm that a match found between data sets refers to the same bug. Finally, we seek a facility to confirm that each bug listed in the entropy (or in our case, Defects4j) datasets is truly a bug.

In order to achieve these goals, the most feasible solution appears to be manufacturing a data set. This involves building a corpus similar to Defects4j by parsing through GitHub or any other contemporary repository collection. We must ensure from the context of the fix commit that each bug we list is truly a bug and mark its context as well. In order to produce enough entropy data, the best solution would be to create our own program to calculate the entropy of each line in a code base. Our calculation can be performed on the Defects4j data and verified against our existing entropy data to confirm its accuracy. This would provide the most flexibility and maximize the amount of entropy data we have.

8 CONCLUSION

Bugs in software systems constitute a serious problem for the producers and consumers of any products and services that rely on stability and security in operation. Efforts to efficiently and quickly identify and patch bugs have resulted in a constantly-growing availability of tools designed to assist developers in quickly verifying the integrity of software systems, though the large volume and variable effectiveness and compatibility of these tools poses a number of cost-benefit challenges. Entropy has recently emerged as an effective means for classifying and prioritizing bug fixes, but it provides little benefit as a standalone statistic and is best combined with multiple bug-detection frameworks to obtain optimal results.

In this research work, we sought to verify this intuition and assess the benefit of cross-checking the output of PMD, a popular code-testing tool, with entropy data. While our results corroborated the notion that entropy and bugginess are indeed linked, the inherent limitations imposed by the size of our dataset and the nature of PMD as a style-checking tool prevented us from drawing meaningful results. Many opportunities exist for expanding upon this work, and the portable nature of our parsing framework allows for larger, more comprehensive datasets to be immediately tested in a minimal amount of time, opening the door for future exploration of these concepts.

9 REFERENCES

[1] A. Hindle, E. Barr, M. Gabel, Z. Su, and P. Devanbu: On the Naturalness of Software. In *ICSE, 2012*.

[2] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu: On the "Naturalness" of Buggy Code. Presented at *ICSE, 2016.* ACM, 2016.

[3] N. Rutar, C. B. Almazan, and J. S. Foster: A comparison of bug finding tools for Java. In *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium*. IEEE, 2004.

[4] PMD: *Don't Shoot the Messenger.* GitHub repository, accessed 2016. https://pmd.github.io

[5] R. Just, D. Jalali, and M. Ernst: Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 2014.

[6] S. Lu, et al: Bugbench: Benchmarks for Evaluating Bug Detection Tools. In *Workshop on the evaluation of software defect detection tools*. Vol. 5. 2005.