references.bib

A Reproduction of "DeepBugs: A Learning Approach to Name-based Bug Detection"

Caleb Tung
Purdue University
West Lafayette, Indiana, USA
tung3@purdue.edu

Dario Rios Ugalde
Lockheed Martin
Arlington, Texas, USA
dugalde@purdue.edu

Young Jin Jung
Purdue University
West Lafayette, Indiana, USA
jung199@purdue.edu

Jordan Matthew Winkler
Purdue University
West Lafayette, Indiana, USA
jwinkler@purdue.edu

Abhimanyu Agarwal
Purdue University
West Lafayette, Indiana, USA
agarw184@purdue.edu

ABSTRACT

INTRODUCTION: Reproducibility and replication of existing work is important in the research community. For works that are not reproduced, the validity of the work may be questioned, since the results are unvalidated. The research community has labeled the large amount of unreproduced work as the "reproducibility crisis". Researchers often do not undertake replication studies because there are relatively few incentives to do so. In this project, we replicate *DeepBugs* by Pradel and Sen, a novel automated name-based bugs detection framework. We re-implement the DeepBugs algorithm from scratch and test it with the original dataset, verifying the original authors' claims. STATE-OF-ART: The research community is encouraging more replication studies, with journals and workshops dedicated to replication work. Prior to this paper, there were no replication studies for DeepBugs. CONTRIBUTION: In this project, we perform a replication study of the DeepBugs name-based bug detection framework. This provides validation for those who wish to implement DeepBugs into their work. METHOD: We re-implement the DeepBugs artifacts for training set generation, source code vectorization, and neural network training and testing. The team uses Python, while the original work uses mostly JavaScript. RESULTS: The team successfully replicated DeepBugs by following the process outlined in the original work. The resulting reimplementation of the framework has an accuracy only 2% lower than that of the original.

CCS CONCEPTS

• **Software and its engineering** → **Software maintenance tools**.

KEYWORDS

defect detection, deep learning

1 INTRODUCTION

In this project, we will perform a replication study of *DeepBugs: A Learning Approach to Name-based Bug Detection* [**?**]. We select this paper because (1) our team is interested in automated bug detection methods, and (2) the paper is particularly novel when compared to other automated bug detectors. Instead of using hand-crafted heuristics to judge where source code contains bugs, the DeepBugs framework fully automates the detection of name-based bugs via deep learning. Moreover, since the paper first appeared in a 2018 OOPSLA issue of *The Proceedings of the ACM on Programming Languages*, it has gained considerable popularity. According to the ACM, 40 other peer-reviewed works cite the paper, while Google Scholar registers the total citation count at 99.

In our replication study, we will implement the algorithms outlined in the DeepBugs paper and test them using the original authors' data sets to see if we can reproduce the same results.

2 MOTIVATION

In a 2016 survey by Springer [**?**], 52% of scientists across multiple fields of study reported that they felt there was a "significant crisis" in the reproducibility of published research. Another 38% felt there was a "slight crisis." This "reproducibility crisis" describes the consensus among the scientific community that an outsized portion of published research is not reproduced. "Our analysis...suggests a large proportion of CS research faces the same threats to replication as those encountered in other areas," writes Cockburn, et al in a recent edition of *Communications of the ACM*. Non-reproducible research is problematic because (1) it is more challenging for other researchers to use the research technology in their own projects, and (2) there exists no independent verification of the research's validity. Therefore, reproducing research should be important to the scientific community.

Unfortunately, few replication studies (formal studies that attempt to independently reproduce an existing work and document its results) are attempted. This is due to many reasons, including: reproducing a study might be expensive, attempting to question an author might be seen as malicious or reflecting incompetence, spending the time to reproduce a study may not further one's own research, etc. Additionally, scientists unable to reproduce work might be inclined to write off a failure as some personal mistake and move on. Incentives to publish positive replications are already low; journals can be even more reluctant to publish negative findings [**?**].

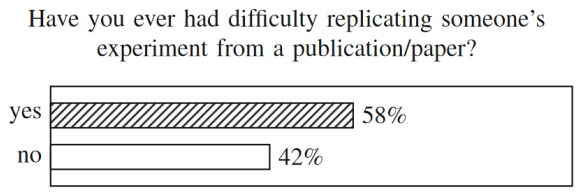

Fig. 68.1 First survey question

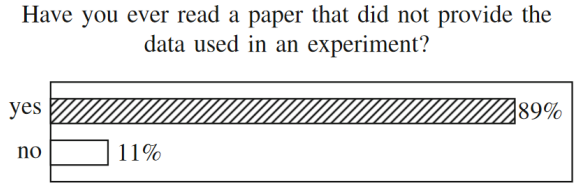

Fig. 68.2 Second survey question

Figure 1: (image credit ITNG 2020 [?])

This all contributes to the "file drawer effect": replication work that would otherwise be useful goes unpublished [?]. Therefore, the lack of replication studies continues.

By performing the replication study ourselves, we wished to contribute to the software engineering community and add a positive effect to mitigate the "reproducibility crisis." Hopefully, our replication study motivates other software engineers to participate in replication studies. Narrowly, we attempt to give a better insight to other researchers who seek to implement DeepBugs into their system.

3 BACKGROUND AND RELATED WORK

3.1 Automated bug detection

Automated bug detection is an active area of research. There is a wealth of statistical information in identifier names. [?] DeepBugs specifically addresses name-based bug detection unlike several popular static analysis tools, such as Google Error Prone [?] and FindBugs [?] that mostly ignores identifier names. Name-based bugs are bugs that can be tracked by linguistic similarity. An example of this is the association of the variable x with width and y with height. In a program, this could lead to a switch-argument bug. Switch-argument bugs are part of a class of bugs called single-statement bugs, which have been reported to consist possibly as much as 33% of bugs in some Java projects. [?]

The reason many program analysis tools ignore identifier names is because such detection is challenging. Name-based analysis must encode the meaning of identifier names, which can be based on subconscious cultural knowledge. Although lexical reasoning is often used to achieve this, existing name-based analysis relies on manually designed algorithms that use hard-coded patterns and tuned heuristics which require significant amount of time and human effort [? ? ? ?]. In contrast, DeepBugs uses a deep learning approach to detect bugs automatically. This is done by training the neural network on a large corpus of code examples.

Before the identifiers can be trained on, they must first be extracted from code. Conditional random fields and statistical machine translation have been used to do this in works such as JSNice [?] and JSNaughty [?]. DeepBugs used Word2Vec [?] to automatically generate vector representations of code snippets to feed into a neural network. Furthermore, in contrast to previous work, DeepBugs focused on implicitly typed languages, where information about the code semantics is conveyed through identifier names by programmers instead of types. The overall framework complements any designed bug detector that relies on processes inducing significant manual work, thereby replacing human effort by transitioning to computational means to achieve the same.

3.2 Similar models for analyzing and finding bugs

The DeepBugs model learns from positive examples and negative examples, focusing on bug patterns that can be expressed via probabilistic means. On the other hand, Bugram [?] uses an n-gram model of code to detect bugs with positive examples only (i.e., anomaly detection). Specification mining [?] uses mined features and specifications to detect unusual bugs in code [? ? ?] while learning only from correct examples and flagging any inconsistencies and errors.

While DeepBugs finds bugs and pinpoints the bug location, Wang, et al.'s approach uses a deep belief network [?] to flag buggy code files for further inspection. Ultimately, DeepBugs differs from existing name-based bug detectors in that it (1) exploits semantic similarities that may not be identified by lexical comparison of identifies, (2) improves generalizability by replacing manually tuned heuristics with a deep neural network.

3.3 Background on replication studies

According to the ACM, replication of an existing study should be performed by a "different team, with the same experimental setup" [?]. And according to BSEL, complete reporting is also highly important due to possible high variance of the parameter of interest. [?] More concretely, the replication study should be conducted without deploying the original authors' artifacts and should instead use independently generated artifacts. The original study is verified if those new artifacts achieve the same results as those found in the original paper. A reproduction study differs from a replication study: a reproduction study checks whether the original results can be obtained using the author's artifacts, while a replication study requires the artifacts to be generated from scratch. Our study will reuse the original authors' code data set [?] but will implement the DeepBugs algorithms using our own source code. Therefore, our project would qualify as a replication study.

It is worth noting that replication studies remain largely unwelcome in academic venues; for example, only 3% of 1,500 top psychology journals welcome replication work as

submissions [**?**]. Fortunately, in recent years, a few research communities have tried to increase incentive for replication studies. Targeted at software engineering research, the ROSE Festival exclusively features replication work [**?**]. Similarly, the Association for Information Systems publishes an entire journal, *Transactions on Replication Research*, dedicated to reproducibility studies [**?**].

Our replication study of DeepBugs will be unique; to our knowledge, there exists no such study for that work.

4 REIMPLEMENTING DEEPBUGS

DeepBugs is structured as shown in Fig. **??**. DeepBugs first converts a piece of source code (their source code is taken from the 150k JavaScript Dataset [**?**]) into an Abstract Syntax Tree (AST). The tree is parsed and converted into condensed tokens that represent the relevant code symbols. The symbols are then turned into vectors so that they can be passed into a neural network to classify if the code is buggy or not.

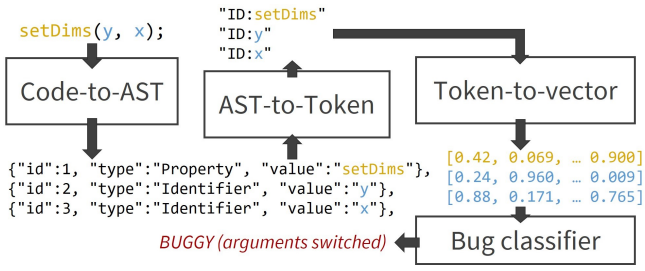

Figure 2: DeepBugs first converts source code into Abstract Syntax Trees (ASTs) and then into semantic-encoded vectors. The vectors are input to a deep learning model that determines if the source code is buggy. Here, a developer has written a function call for `setDims(width, height)` using `setDims(y, x)`. DeepBugs predicts this as "buggy" because the system understands that `x`, `width` are semantically similar, as are `y`, `height`. Thus, it predicts the arguments are swapped; the order should be `(x, y)`, not `(y, x)`.

The DeepBugs framework is designed to detect any name-based bug pattern. Like the original authors, we implement the framework for the "swapped argument" bug case. Our reimplementation of DeepBugs is described in the following four steps.

4.1 Step 1: Code-to-AST

The original authors used the Acorn JavaScript parser [**?**] to convert JavaScript source code files into a Json formatted Abstract Syntax Tree (AST). For simplicity of testing the swapped-argument bug pattern, we removed AST nodes unrelated to name-base bugs. Our simplified format only uses the `''children''` AST attribute, which identifies the AST nodes that are acted upon by a given statement or call block. For example, calling some method `object.methodName(arg1, arg2)` would yield the following simplified AST structure:

```
{''id'':1, ''type'':''CallExpression'', ''children'':[2,5,6]},
{''id'':2, ''type'':''MemberExpression'', ''children'':[3,4]},
{''id'':3, ''type'':''Identifier'', ''value'':''object''},
{''id'':4, ''type'':''Property'', ''value'':''methodName''},
{''id'':5, ''type'':''Identifier'', ''value'':''arg1''},
{''id'':6, ''type'':''Identifier'', ''value'':''arg2''}
```

This allows us to quickly switch the ordering of the elements in the `''children''` list and generate swapped-argument bugs.

4.2 Step 2: AST-to-Token

For a given bug pattern, not every node in the AST necessarily needs to be evaluated. Further, since each node contains many characters dedicated to represent the node's schema, each relevant group of nodes should be compressed into small tokens that represent the entire group. The tokens can then be vectorized and fed to a neural network for bug classification.

More concretely, for the swapped-argument bug pattern, we are only interested in function names and argument names for any 2-argument function call. To generate tokens, we do the following: For a given AST node, only attempt to generate a token from its `value` field. If the node does not have a `''value''` field, derive one from its `''children''`. The rules we use to generate tokens are similar to those used by the original authors:

- A `CallExpression` with multiple `Identifier` children can be tokenized separately, with the first child tokenized as the function name, and the remaining children tokenized as the arguments.
- A `MemberExpression` with both an `Identifier` child and a `Property` child can be tokenized into the object name and the method/property name, respectively.
- Variable names and function names are tokenized as identifiers, with `''ID:<variable name here>''`.
- JavaScript literals of all types are tokenized as `''LIT:<value>''`.
- Other language elements like operands, `if/else` blocks, `try/catch` blocks, and `for` loops are ignored.

Using that system, we crawl the entire corpus of code in the 150k JavaScript Dataset, converting each file's AST into a list of tokens. This dramatically compresses the size of our data, distilling only important components. For example, the 6-node AST provided in the Step 1 example (Sec. **??**) would be tokenized into the following list:

`[''ID:object'', ''ID:methodName'', ''ID:arg1'', ''ID:arg2'']`.

4.3 Step 3: Token-to-Vector

To vectorize a token for the neural network bug classifier, the original authors use the Word2Vec [**?**] model. Word2Vec can capture lexical similarities between two words, based on the context of the other words surrounding the two words. For example, the letter `x` and the word `width` do not necessarily carry similarities on their own. However, in the context

of `setDims()`, they take on similar meanings (i.e., "width along x-dimension"). Under the CBOW (Collected Bag Of Words) variant, Word2Vec considers a window of words both before and after the word in question to decide the word's vector representation. In our example, this would mean that Word2Vec's output vectors for `''ID:x''` and `''ID:width''` would be similar.

To build our Word2Vec model, we first use our AST-to-Token extractor from Step 2 (Sec. **??**) to list all tokens, in order of appearance, for a given JavaScript file's AST. The list of tokens can then be used as a "paragraph" of words to feed Word2Vec. We then implement and train our CBOW Word2Vec instance using a window size of 200 tokens (i.e., 100 tokens before the current token, 100 tokens after the current token). To sanity-check our model, we compare the vector similarities of `''LIT:true''` with `''LIT:false''`, `''LIT:0''`, and `''LIT:1''` using cosine distance (dot product). We observe that the two true-values are similar to each other (i.e., distance is close to 0), and the two false-values are similar to each other. Meanwhile, the true-values are not similar to the false values (i.e., distance is much farther than 0). This is expected behavior, suggesting that the Word2Vec model is reasonably trained.

4.4 Step 4: Bug Classifier

As shown in Fig. **??**, the DeepBugs bug classifier is a simple neural network that feeds the input through a 20% dropout regularization layer, a 200-node hidden layer with ReLU activations, another 20% dropout layer, and a final sigmoidal node to determine the buggy/not buggy prediction. The small size of the network facilitates faster training for different bug patterns.

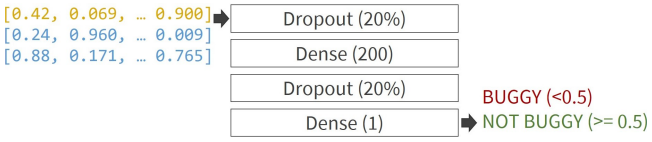

Figure 3: The Deepbugs classifier is a small neural network that can be rapidly retrained to predict different name-based bugs. As shown, we train our replicated classifier for swapped-argument bug detection.

Depending on the bug pattern being tested for, the input to the neural network may differ. For the swapped-argument bug, the vectors for the function name and two arguments are concatenated before feeding into the network.

5 COMPARING WITH THE ORIGINAL

Like the original authors, we evaluate our replicated DeepBugs using switched-argument bugs identified in the 150k JavaScript Dataset [**?**]. The dataset is already partitioned into training and evaluation categories.

First, we extract every 2-argument function call group from the ASTs generated for the dataset. Specifically, we extract every function call that fits either of the following patterns:

- `functionName(arg1, arg)`
- `objectName.methodName(arg1, arg2)`

Note that in the second pattern, the function name is extracted for `methodName`, not `objectName.methodName`. Attempting to vectorize `objectName.methodName` would produce two vectors instead of one, because Word2Vec would need to vectorize `objectName` and `methodName` separately. The neural network has a fixed input size, so we must ensure that every extracted sample produces exactly three vectors, instead of a mix of three- and four-vector samples.

For every 2-argument function call we extract, we assume that the argument ordering is correct. Thus, the extracted function calls comprise our positive data samples. We make that assumption because the 150k JavaScript dataset is built using code from various public, open-source projects, so the source code is less likely to be buggy.

To create negative data samples, we simply switch the order of the `(arg1, arg2)` for all extracted 2-argument function calls. Thus, our training and testing data is comprised of 2× the number of extracted samples:

- `functionName(arg1, arg2)` - POSITIVE
- `functionName(arg2, arg1)` - NEGATIVE
- `objectName.methodName(arg1, arg2)` - POSITIVE
- `objectName.methodName(arg2, arg1)` - NEGATIVE

Each data sample is then vectorized using the following procedure. Via the AST-to-Token method described in Step 2 (Sec. **??**), we convert each data sample into a tuple of three tokens (e.g., `(''ID:functionName'', ''ID:someParam'', ''LIT:true'')`). Each of the three tokens is individually vectorized using the Word2Vec model we trained in Step 3 (Sec. **??**). The final, classifier-ready sample is comprised of the three vectors, concatenated into a larger vector.

To train our DeepBugs classifier from Step 4 (Sec. **??**), we use the vectorized data samples from the 150k JavaScript dataset's training partition. Training is completed rapidly (10 epochs, batch size 100, RMSprop optimizer with binary cross-entropy loss).

As shown in Tab. **??**, our results are very similar to those reported by the original authors. We compare multiple aspects of the replication with the original.

There are a few differences between our replication and the original, however. In the original paper, the authors claim to use a vocabulary size of 10,000 unique tokens. In actuality, the Word2Vec model they provide and the model we train both retrieve the same 9,994 tokens. We believe that this difference was merely because of rounding up the number. However, there are no significant implications of this. The vectors generated are also different; the original authors do not provide their random seed for their Word2Vec model. We verify that their method works by using the vectors as input to our DeepBugs model, hence we confirm that not having the exact same seed for the vector generation does not have significant impact on our results compared to the original author's work. Our model flags many of the bugs that the

Step	Comparison Metric	Original VS Replication
1 & 2	Token vocabulary size	Same: 9,994 VS 9,994
	Extracted token values	Identical
3	Generated vectors	Different (authors did not provide the seed)
4	Training Loss	Close: ∼0.002 VS ∼0.002
	Test Error	Close: ∼0.04 VS ∼0.06

Table 1: On the switched-argument case from the 150k JavaScript Dataset, our DeepBugs replication successfully captured similar performance to the original authors' work.

DeepBugs Component	Replication VS Original Authors
Code-to-AST	-Both use Acorn to generate ASTs -Beyond that, we reimplement everything in Python using our own logic
AST-to-Token	-We implement everything in Python from scratch
Token-to-Vector	-Both use the Gensim language framework -We implement everything in Python from scratch
Bug Classifier	-Both use Tensorflow/Keras -We implement everything in Python from scratch -We train the model ourselves
Dataset	-Both use the 150k JavaScript Dataset -Our scripts to prepare the dataset for classification are reimplemented in Python from scratch

Table 2: Our replication uses the same dataset and a few of the same third-party libraries as the original authors, but we implement everything else from scratch based on the instructions from the paper. We do not use any artifacts produced by the original authors.

original authors' did. Our accuracy is not exactly the same; although our training loss drops to roughly 0.002, like their model, our model performs with 2% worse accuracy.

Because our work is a replication, we do not reuse artifacts generated by the original authors. Our implementations are based on our understanding of the authors' instructions in their paper. The DeepBugs authors wrote most of their code in JavaScript, while our work exclusively uses Python. We use the same dataset to verify the authors' results. A comparison between our and the authors' implementations are shown in Tab. **??**.

Because our results are very similar and were produced using artifacts that we generated ourselves, we are confident that the DeepBugs paper's results are valid; our replication is successful.

6 CONCLUSIONS

Although there are many tools to help software engineers identify and mitigate bugs, such as fuzzers, linters, and unit test frameworks, deep learning provides another avenue to develop tools which can support the software development process. The original DeepBugs study addressed the problem of name-based bug detection through deep learning. The study was able to produce detectors with high accuracy. Following the procedure outlined in the study, our team was able to replicate results with similar accuracy and precision. In the future, our team plans to share our results with the software engineering community by submitting the results to a suitable venue (such as the Journal of Open Source Software). To provide further insights into DeepBugs' capabilities, we currently consider to further test DeepBugs beyond the 150k JavaScript Dataset, either with a different corpus of JavaScript code or with different programming languages entirely (this will require modification of a few DeepBugs elements). Implementing DeepBugs for a variety of languages can assist software engineers in the detection of these name-based bugs and increase the quality of software across the board.

7 APPENDIX A: ETHICS ASSESSMENT

7.1 Dario

Just as software engineers must submit their code for review to their peers, so too must the authors of research papers submit their work to the academic community. In a similar fashion, the authors offer up their work for the scrutiny of their peers to find validation. This process has ethical implications that go beyond the opinions of the reviewers. In this case, our attempt to replicate the Deepbugs study found similar results to that specified in the original study. As such, the time our group spent this semester serves to fortify the claims made in the original study and provide additional, independent proof to the validity of this work.

7.2 Abhimanyu

Conducting replication and reproduction studies of previously conducted research is one of the hallmarks of good science. Reproducibility provides scientists with evidence that research results are objective and reliable and not due to bias or chance [**?**]. Replication and reproduction cumulatively help in addressing ethical issues such as data falsification. In this project, the team worked on implementing the work done in DeepBugs. This helped in verifying and validating results put forth by the author of DeepBugs. Fabrication of results by the authors could lead to major ethical issues.

It is noteworthy that while the original work made use of JavaScript extensively, our implementation is in Python. It is entirely possible to see different results due to difference in

programming languages being used as the code written would see variation. As engineers following ACM Code of Ethics, one must be responsible and avoid public harm. The results put forth by DeepBugs along with the result validation done gives credibility to the original author's work. An automated bug detection tool like DeepBugs would certainly prove to be useful to the engineering society.

An ethically ideal software engineer would be an individual who is honest, responsible, and courageous. These aspects are crucial because these would impact the choice a software engineer makes in difficult ethical circumstances. Moreover, these traits may not be accounted by the code of ethics that most engineers are expected to follow, they essentially come from within due to the moral values one has always been bounded with. Being honest with others and towards your work is extremely important. An honest software engineer takes the responsibility and comes out to be accountable for the work done.

Throughout our research study, we followed the ACM Code of Ethics and ensured that the replication study carried out eliminates any concerns on reproducibility. This idea ensures public welfare and ensures that no software without a strong fundamental foundation and improper testing is not proposed for public use.

7.3 Caleb

A replication study lends credibility to an existing piece of research. In our case, we validated the results of the DeepBugs paper. By reimplementing the original authors' work entirely in Python (they used JavaScript as well), we provided more ways for the public to use the paper, making it even more accessible. Although this does support aspects of the ACM/IEEE Code of Ethics – we help ensure that the authors stay honest with the public – there is another ethical consideration: is the original work something that deserves the credibility we give it via replication?

I believe the original DeepBugs paper is a good contribution to science and engineering. By automating new types of bug detection, the paper provides another tool for engineers to use in their ongoing war against defects in software. Further, I find it difficult to imagine any lasting ramifications of the paper that would violate any point of the Code of Ethics. Therefore, I find this paper to be worthy of elevating via replication.

7.4 Young

Replication and reproduction studies are essential to keep the research community healthy and reliable, by primarily allowing the other researchers to adopt, verify and possibly extend the implementation with confidence. As noted earlier, the community recognizes this as a "reproducibility crisis." An ethically ideal person is courageous without manifesting the common virtue or excellence to the maximum extent possible, beyond what's ordinary. Harris Jr. describes this as a "virtue portrait" in his article. [**?**] With the technical and non-technical excellence, one can attempt to sketch a virtue portrait of the ethically ideal software engineer. It is our belief that all software engineers should attempt to picture virtue portrait on themselves while practicing software engineering especially by contemplating and possessing an utmost interest in the social outcome of one's action. We believe that our practice of replication helps mitigate the reproducibility crisis and has positive ethical social outcomes by addressing the following clauses of ACM-IEEE Code of Ethics: *1.03. Approve software only if they have a well-founded belief that it is safe, meets specifications, passes appropriate tests, and does not diminish the quality of life, diminish privacy or harm the environment. The ultimate effect of the work should be to the public good.* and *3.10. Ensure adequate testing, debugging, and review of software and related documents on which they work.*

7.5 Jordan

Replication and reproduction studies are part of healthy scientific progress. Repeatability of observation is a core part of the scientific method. The scientific method has been a staple formula for both the natural and social sciences since its inception. Reproduction studies are also important in the formal sciences. In the formal sciences, it is more so called publishing an alternate proof, instead of a reproduction study. In the formal sciences, it is important to do this for sake of understanding the result from a different perspective. It is also the case that results in formal sciences, if complex enough, may disagree. [**?**] Replication is often done "automatically" as a result of re-creating a proof when one reads a paper. But having multiple alternate published proofs adds confidence in the correctness of a result.

8 APPENDIX B: ARTIFACTS

Our code is version-controlled on GitHub. We provide a README and commented code at https://github.com/code-correctional-facility/deepbugs-jr.

9 APPENDIX C: POST MORTEM

9.1 What Went Well

The team largely feels that the project went smoothly. The replication was completed successfully, and the team gelled well and operated without conflicts.

9.1.1 **We intentionally and carefully designed procedures for effectively producing code, and stuck to them throughout our project.** At the start of the project, our team clearly laid out procedures for checking code into Git, writing unit tests, conducting code reviews, and communicating with Teams. We established an automated Kanban board and issue tracker, and we maintained a steady backlog of tasks to draw from.

9.1.2 **We planned for success!** We spent a considerable amount of time identifying tasks from the original DeepBugs paper. By conducting a comprehensive review of the process followed by the authors, we were able to identify the major milestones necessary for completing the project. The team

went on to further break down each of the milestones into a list of actionable items which would ultimately become the backlog the team used to plan out the replication. This allowed the development process to move along smoothly. During our weekly check-in meetings the team would view the board with this list and easily understand what was in progress and what the next steps were once tasks were completed.

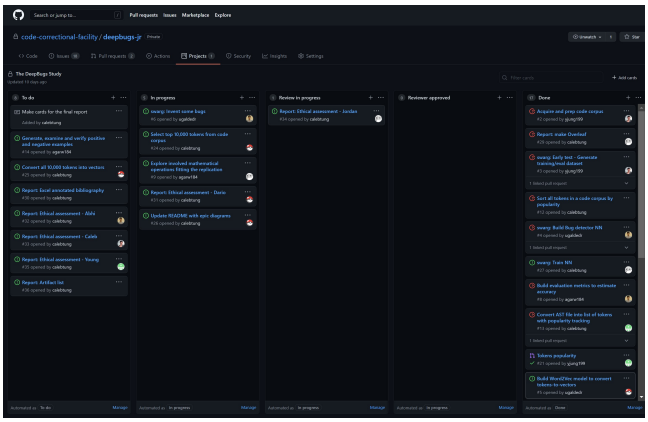

Figure 4: Example of our kanban board with automated issue tracking. As we completed pull requests, the cards automatically rearranged themselves on the board.

9.1.3 **Our team collaborated productively.** Several tasks on the backlog were completed by multiple members working closely together. Although we were careful to assign tasks in bite-sized chunks to stay quasi-Agile, some tasks wound up requiring more assignees. Team members would help each other by pair programming and recommending additional resources. The team did an excellent job of keeping each other in the loop when faced with any issues. This was accomplished by using Microsoft Teams as a general chat. Members would regularly ping the rest of team to ask for advice or alert others on the completion of tasks. The members were highly responsive and received feedback within a matter of minutes in most cases. We all performed manual code reviews to stay familiar with our codebase and "enforce" good practices through peer pressure.

9.1.4 **We stayed on schedule, with room for flexibility.** With our team spread across the globe (multiple US timezones and South Korea), we designed our workflow asynchronously, supplemented with regular, synchronous "tag-up" meeting times. By investing effort in maintaining a granular, bite-sized backlog, we kept work from getting bottlenecked by members needing to communicate across 13-hour time gaps. US members maintained empty slots in their evening schedules and Korea members kept their mornings open, allowing us to accommodate the few time it was necessary to synchronize outside of meeting time. Our project was thus able to adhere to our schedule, without demanding excessive commitments from team members.

9.1.5 **We developed camaraderie.** No one on the team had met each other prior to the project. By keeping our meetings casual and cracking jokes, the team successfully established a pleasant working environment that was able to complete the project without the usual "storming" period of team development.

9.1.6 **We produced quality, useful software.** We created well-documented, test-covered software: an easy-to-use replication of the DeepBugs paper. This would provide more options to users looking to try out the DeepBugs framework (the original authors used mostly JavaScript, but all of our code was written in Python).

9.1.7 **We became better engineers.** Not only was this project a terrific opportunity for us all to gain technical competency in an area of research that we had never dealt with before, but it also allowed us to exercise other software engineering muscles. In order to better understand some of the modern tools used in the software development process, our team set up our Github repository to make use of Github Actions any time code was pushed. This provided members of the team who were not familiar with the typical continuous integration process insight as to how to define and implement a continuous integration pipeline. Building on the development operations process, the python unittest module along with a basic smoke test were implemented to demonstrate a continuous integration pipeline for our project. This allowed members put material discussed earlier in the semester to practical use. We agreed to add tests for branching points to provide coverage as well as, edge cases to ensure proper functionality in the code. In the future, this project could possibly serve as reference material should any team member be required to go through setting up similar infrastructure once again.

9.2 What Did Not Go So Well

Although the project was completed successfully, our team did run into an unanticipated roadblock in the middle of development.

9.2.1 **Almost bit off more than we could chew.** Prior to the project proposal, our team had a very different vision of what our project would look like. Originally, the team had planned to conduct a systemic review on the state-of-art of automated software testing research. In addition to this, the team envisioned the creation of a tool which could direct software engineers to a variety of testing applications intended for the identification of specific bugs. After discussion with Professor Davis following the proposal, the team was reigned in and the scope of the project was narrowed to this replication. It was unfortunate that correcting the team onto this path ate some of the initial planning time leading up to the first status update. In hindsight, now that we've completed our replication, it is quite clear completing a systemic review,

generating a tool based on our findings, and compiling this report would not have been feasible within this time frame.

9.2.2 **Losing time for 10,000 tokens.** During the course of development, our team encountered an unnecessary time sink while parsing tokens for training the Word2Vec model. Dario and Young had been assigned tasks to collect the top 10,000 tokens from the source code dataset and train the Word2Vec model. Even after the two spent a week and many precious development hours on the method, the model was still not functioning correctly. After this was brought up during the weekly team meeting, the rest of the team moved to assist, conducting code reviews and familiarizing themselves with the Word2Vec paper. While browsing documentation online, Caleb discovered that the team's desired functionality was actually already implemented in the Word2Vec model framework and could be activated with a single parameter. We wound up discarding the additional code written over the week in favor of this simpler solution. This unfortunate situation could have been avoided by spending a little more time understanding the tools we used for the study. The issue cost the team valuable hours which could have been better spent supporting other tasks. Ultimately, the task was still resolved according to schedule, but any possible time savings were lost. Going forward, scheduling time to do documentation reviews for the tools being used could prevent these types of situations.