Toward a Reliability Measurement Framework Automated Using Deep Learning

John Heaps
University of Texas at San Antonio
john.heaps@utsa.edu

Xueling Zhang
University of Texas at San Antonio
xueling.zhang@utsa.edu

Xiaoyin Wang
University of Texas at San Antonio
xiaoyin.wang@utsa.edu

Travis Breaux
Carnegie Mellon University
breaux@cs.cmu.edu

Jianwei Niu
University of Texas at San Antonio
jianwei.niu@utsa.edu

ABSTRACT

We propose a framework to detect software bugs based on code pattern detection. Our framework will mine and generate bug patterns, detect those patterns in code, and calculate a vulnerability measure of software. While our framework performs well, we realize that it requires heavy manual tasks that render the framework infeasible to use in practice. However, we believe that recent advancements in machine learning will allow us to apply deep learning techniques to source code, which will help automate our framework for better practicality in the real world.

ACM Reference Format:
John Heaps, Xueling Zhang, Xiaoyin Wang, Travis Breaux, and Jianwei Niu. 2019. Toward a Reliability Measurement Framework Automated Using Deep Learning. In *Hot Topics in the Science of Security Symposium (HotSoS), April 1–3, 2019, Nashville, TN, USA*. ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3314058.3317733

1 INTRODUCTION

The security and reliability of software systems are essential in making decisions for real-world problems. Many process-based approaches and models have been developed to quantify software vulnerabilities and reliability. However, these approaches, while useful, have many limitations. Our framework develops a pattern-based vulnerability measurement model, which checks software artifacts for the existence of negative patterns to produce a vulnerability measure and we hope to extend to probabilistically estimate whether a given cyber system preserves a given set of security or reliability properties. The project's components include a *Learning Engine* to mine bug repositories to create bug patterns; a *Pattern Detector* to detect the existence and invocation of patterns; and a *Vulnerability Model* for the estimation of the vulnerability of a software project based on the detected patterns.

Further, we recognize that our framework (as well as many other frameworks we have studied) requires a large amount of manual processes with software code to perform properly and stay up-to-date. We propose to use deep learning to mitigate some of these manual processes to make our model feasible to real-world applications. However, while using deep learning on natural language has recently gained much attention, their is quite limited literature and research that has applied deep learning to software code. We discuss our first steps toward utilizing deep learning for processing software code, which is creating high quality word embeddings so learning tasks can be performed.

HotSoS, April 1–3, 2019, Nashville, TN, USA

ACM ISBN 978-1-4503-7147-6/19/04.
https://doi.org/10.1145/3314058.3317733

2 APPROACH

2.1 Framework

The *Learning Engine* acts as a repository of known patterns which will be used to identify bugs in code during pattern detection. To create this repository, we utilized the Common Weakness Enumeration (CWE) database and Github, focusing on Java-related bugs and projects. For each bug we collected bug categories, descriptions, solutions, and sometimes sections of code. Then, for the most common code bugs we manually extracted concrete code patterns from them to be used during Pattern Detection.

To conduct the *Pattern Detection* in our framework, we utilized SpotBugs[1], a fork of the static analysis tool FindBugs [2]. Given code patterns, the tool analyzes Java bytecode and detects the existence of the patterns.

After identifying bug patterns, they further needed to be linked to abstract quality aspects to calculate a vulnerability measure. Specifically, we consider five major aspects of code reliability: *Data*, *Behavior*, *Performance*, *Security*, and *Design*.

Finally, the detection of a bug pattern does not indicate a bug exists, only that it is highly probable it exists. We integrate targeted testing techniques in order to test if the section of code identified by the pattern actually produces an error, showing it truly is a bug.

For our *Measurement Model* we use Formula 1 to generate a vulnerability value, normalized to the range [0, 1]. Here, *Detected* is the set of bug instances found in the code using patterns. *Risk(b)* denotes the risk value of a given bug *b*, which is calculated using the quality aspects and testing results during Pattern Detection. *R* is a constant representing the average risk sum per software project.

$$Vulnerability = 1 - \frac{R}{R + \sum_{Detected} Risk(b)} \quad (1)$$

With this formula, vulnerability scores above 0.5 indicate above average vulnerability, and lower than 0.5 indicate below average vulnerability.

[1] SpotBugs URL: https://spotbugs.github.io/

2.2 Word Embeddings of Code Elements

Like most sophisticated vulnerability and reliability measurement frameworks we found, many processes in our framework must be performed manually. Since the magnitude of these manual processes makes such frameworks infeasable for real-world use, we hope to utilize deep learning to mitigate some or all of the manual processes to make our framework practical.

Most of our manual processes deal directly with code. However, deep learning tasks cannot be applied directly to code elements as they have no meaningful underlying numerical representations for the learning task calculations to be performed on. Inspired by natural language processing (NLP), we have created word embeddings for Java code elements. These word embeddings were created in Tensorflow [1] using a modified version of the Word2Vec Skip-gram [4] algorithm.

In NLP, Word2Vec is based on the assumption that similar words are used in similar context. For example, the similar words "fast" and "quick" would be used in the same way and in close proximity to the same words in a large body of text, or *corpus*. Word2Vec creates vector representations for words, called *word embeddings*. For each word in a corpus, Word2Vec identifies the words surrounding it in the corpus as context and then performs word prediction using deep learning to modify the embedding of each word. After performing the prediction task many times over the entire corpus, the word embeddings will eventually converage and represent the relative semantic meaning of all the words. A good sign of high quality word embeddings is that similar words are grouped together in the vector space, since similar words should have very similar word embeddings.

For source code, we perform a similar process. We first convert source code to an abstract syntax tree (AST), which is a graph representation of the source code where code elements are nodes and the structure is defined by the edges. We then walk the AST and collect context for each node visited. Through our research we have found that simply using surrounding code elements as context for the current node is not sufficient to produce high quality word embeddings. Therefore, we identify context based on asking the question "What other node types or code elements in the AST are needed to determine the meaning or perform the task of the target code element?"

3 RESULTS

For our framework, we have created a website[2] where a Github repository URL can be entered. If the automatic build script is able to successfully build the repository, it will produce measurement results and bug detection outputs based on our framework. For the final vulnerability calculation, R has been determined based on the average bug occurances across 717 top Github Java projects.

In our initial attempt at creating word embeddings for code elements, we performed Word2Vec on the Java Development Kit 8 (JDK8) which included 15,355 unique code elements. We achieved a lowest loss of 2.69 and perplexity of 14.80, which is acceptable. We explored the vector space that contains our word embeddings and found that many similar code elements have been grouped together, as can be seen in Figure 1. For example, in the original vector space

[2]http://galadriel.cs.utsa.edu:25666/

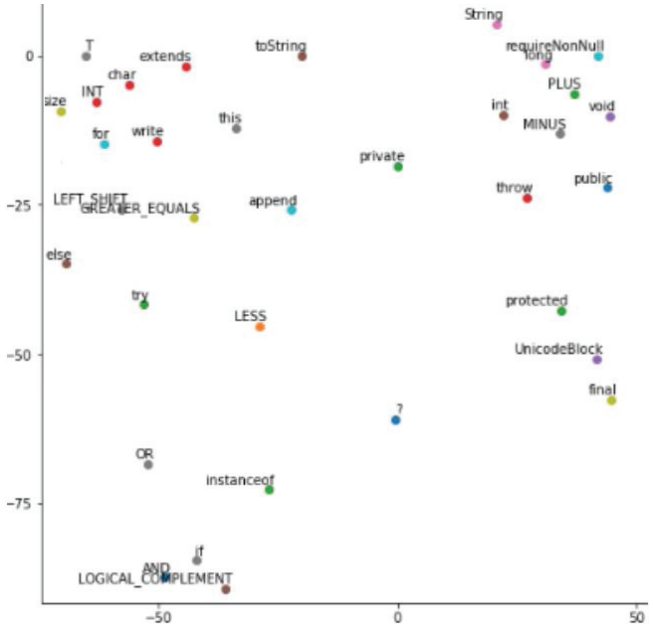

Figure 1: JDK8 embeddings for 34 common code elements; this graph visualizes 300 dimensional vectors in 2 dimensional space.

if can be found closest to *&&*, *||*, *while*, and *?* (which represents the conditional statement); and the double literal can be found closest to the char literal, String literal, int literal, and + operator.

4 FUTURE WORK

In the next steps of our research we will continue to refine and develop word embeddings for code elements. We will then use those word embeddings to automate our framework using learning tasks that include: detection of semantic vulnerabilities, enforcement of natural language policies, classification of bugs to abstract quality aspects, and generation of robust concrete bug patterns.

ACKNOWLEDGEMENTS

We would like to thank MSI STEM Research & Development Consortium (MSRDC) (Grant Award #D01_W911SR-14-2-0001-0012), the CREST Center For Security And Privacy Enhanced Cloud Computing (C-SPECC) (Grant Award #1736209), and the National Science Foundation's (NSF) Early-Concept Grants for Exploratory Research (EAGER) (Grant Award #1748109) for their contributions to this project.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 265–283.

[2] Nathaniel Ayewah, David Hovemeyer, J David Morgenthaler, John Penix, and William Pugh. 2008. Using static analysis to find bugs. *IEEE software* 25, 5 (2008).

[3] Peter Mell, Karen Scarfone, and Sasha Romanosky. 2006. Common vulnerability scoring system. *IEEE Security & Privacy* 4, 6 (2006).

[4] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).