Machine-Learning Supported Vulnerability Detection in Source Code

Tim Sonnekalb
tim.sonnekalb@dlr.de
Institute of Data Science, German Aerospace Center (DLR)
Jena, Germany

ABSTRACT

The awareness of writing secure code rises with the increasing number of attacks and their resultant damage. But often, software developers are no security experts and vulnerabilities arise unconsciously during the development process. They use static analysis tools for bug detection, which often come with a high false positive rate. The developers, therefore, need a lot of resources to mind about all alarms, if they want to consistently take care of the security of their software project.
We want to investigate, if machine learning techniques could point the user to the position of a security weak point in the source code with a higher accuracy than ordinary methods with static analysis. For this purpose, we focus on current machine learning on code approaches for our initial studies to evolve an efficient way for finding security-related software bugs. We will create a configuration interface to discover certain vulnerabilities, categorized in CWEs. We want to create a benchmark tool to compare existing source code representations and machine learning architectures for vulnerability detection and develop a customizable feature model. At the end of this PhD project, we want to have an easy-to-use vulnerability detection tool based on machine learning on code.

CCS CONCEPTS

• **Security and privacy** → **Software security engineering**; • **Software and its engineering** → Application specific development environments; • **Computing methodologies** → *Supervised learning by classification*; Neural networks.

KEYWORDS

software security, vulnerabilities, vulnerability detection, source code analysis, machine learning on code

ACM Reference Format:
Tim Sonnekalb. 2019. Machine-Learning Supported Vulnerability Detection in Source Code. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3338906.3341466

Publication rights licensed to ACM. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of a national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.
ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-5572-8/19/08…$15.00
https://doi.org/10.1145/3338906.3341466

1 INTRODUCTION

Accounting security issues for software at an early stage of development is meaningful and can save costs of all subsequent support after release. Considering the security aspect should be a part of every software development life cycle. The security-related problems already start with the choice of the programming language, as some are immune for certain vulnerabilities, as is Java for buffer overflows, which is by far the most common vulnerability in using C [18]. Security must be considered already at the design stage of the software, as creating fundamental designs for the structure, organization, data flow, etc. reduces bugs right from the beginning [18].

However, a project in the agile software development often starts on a small scale with an initial idea and sometimes evolves over time into a larger version that employs a whole software development team. At this point, vulnerabilities in already written code and running software must be found. A proven approach is to run static or dynamic analyzers to identify common mistakes by analyzing and/or executing the code with different inputs. The static analysis is based on a predefined set of rules to discover vulnerabilities such as buffer overflows or calls to insecure library functions [9]. However, these tools generally have a high level of false positives [5], making them difficult to use for developers who often lack the resources to handle all the false alarms of these tools.

Nowadays, machine learning is used in many areas to automate applications by extracting patterns from preprocessed data. Machine learning on code (MLoC) in general is the analysis of source code with code as input data. There are some open challenges in this field, described in section 2, to increase the accuracy and to put this concept into practice.

According to a recent survey by Ghaffarian and Shahriari [5], the discovery of vulnerabilities based on machine learning can be broken down into three major areas: vulnerability detection based on software metrics, anomaly detection and vulnerable code pattern recognition. In our work, we focus on the latter. Our goal is, therefore, a supervised machine learning process that extracts patterns of vulnerable code snippets and re-identifies them in unseen source code. Anomaly detection as unsupervised learning method is less relevant to this project, because it comes as well as static analysis with a high false-positive rate. However, we want to focus on improving the usability of these tools. Finally, the vulnerability knowledge could be divided into known patterns and unknown patterns. The latter can only be discovered by an anomaly-based method. We will focus on known patterns categorized for the most common weaknesses, or CWEs [12], a project for classifying frequent vulnerabilities.

2 RELATED WORK

The vulnerability analysis on source code can be divided in three types: lexical, syntactic and semantic analysis [7]. By looking at source code analysis with machine learning at different stages, we can distinguish three waves [1]: the first wave consists of basic tools with hand-crafted features. The second wave follows a lexical analysis, which has already been studied extensively, treating code as text and organizing code into classes using natural language processing techniques. The ongoing development, taken into account the semantics of programming languages, is referred as the third wave of MLoC and pointed out as future work. Once the weak spots in the code have been identified, additional measures such as automatic bug fixing can be applied. Fist works can also be found in the machine learning field [11]. Bug fixing could be done history driven or based on deductive reasoning. Code repair is not part of this work, we focus on the detection of weak spots.

Allamanis et al. represent programs as graphs [2], where edges correspond to syntactic and semantic relationships. They evaluate their approach in open source C# projects to predict variable names based on their usage and to predict the correct variable name at the corresponding program location. They give selected examples for the correct prediction of variable usage and also test their model on unseen projects. This work is relevant to the third wave of MLoC and there is still plenty of room for improvements in accuracy and F1 score.

In the work of Tufano et al. [17], different source code representations are used to detect code clones with a deep learning approach. Identifiers, abstract syntax trees (AST), control flow graphs (CFG), or byte code, are used as representations, each providing an orthogonal view of the code snippet and demonstrating the effectiveness of each, but also creating a combined model with ensemble learning. The authors have shown that both single and combined representations work with a very high accuracy in this experiment. We also want to test single and combined representations of source code, but for a another application: vulnerability detection.

The actual detection of vulnerabilities in source code is researched in [15] using an identifier representation in a deep learning environment. They created a data set from a variety of sources and labeled them, based on the results of statical analysis tools considering the top five CWE categories and empirically developed a best practice pipeline consisting of a random embedding of the source code tokens, learned the features through a one-dimensional CNN and used it as input to a random forest classifier that decides if the code is secure or vulnerable. This work could be assigned to the second wave of MLoC. By creating an embedding that pays more attention to the token semantics than just a random embedding, this method could achieve a higher classification accuracy. In addition, with a tree representation of the code, this method could be further improved.

A graph based representation brings better performance for vulnerability detection, as shown in the work of Kronjee et al. [9] for CFGs to detect SQL injections and cross-site scripting in PHP applications. They have shown that the chosen machine learning algorithm is not crucial because it provides approximately similar AUC-PR (area under curve for precision recall) values for the same vulnerability. We also want to include CFGs as code representation in our work, but for C/C++ fragments.

A path-based approach called Code2Vec [3] is used to provide a framework that creates a fixed-length continuous vector from code snippets of any size to detect code similarities. However, the results show that the prediction accuracy depends less on the path than on the variable names of the start and end nodes in the considered path. Other limitations of this work are the non-universal closed-label vocabulary, which is limited to the training data. The work is though a promising concept. We want to find out, how well this approach works for vulnerability discovery in our future work.

Binary code representations are used for pattern analysis [6, 10, 19] by identifying potentially dangerous usage patterns of the C standard library to predict the probability of a crash when executing certain commands and to test them with various fuzzing tools. However, due to high prediction errors, we move away from the idea of analyzing byte code and look instead at code, which is represented in a high-level format.

MLoC, in general, is also relevant to various other applications, e.g., for code summarization, code completion, or prediction of function and variable names. The challenge in this field is to find a good representation of the learning data, taking into account the programming language semantics and creating a contextual word embedding depending on the environment of the particular token, since tokens can have different meanings [4].

3 BENCHMARKING STATE-OF-THE-ART METHODS

The presented approaches in the literature differ greatly. These approaches often use different source code representations as input. In addition, they use various architectures such as dense neural networks, random forests, convolutional neural networks (CNN), long short-term memory (LSTM), etc. Their evaluations are often not comparable due to a use of different measurements.

In form of a systematic literature review, we want to reappraise the state-of-the-art. This should include a benchmark tool as shown in Figure 1. To do this, we are re-implementing existing machine learning pipelines for vulnerability detection and comparing them for their classification ability for CWEs. The evaluation is carried out with the same data as well as uniform measurements in order to obtain a ground truth for further studies.

This tool converts the source code snippets into three representations: token streams, ASTs and CFGs. Tokens and ASTs are often used in state-of-the-art methods. With CFGs, we can model the execution order of statements so that vulnerabilities often correspond to paths in the CFG [20]. Therefore, CFGs are also important for vulnerability modeling. These three representations are used for each of the approaches and are finally classified for the most prevalent CWE classes according to the Top 25 CWEs [12].

Furthermore, we want to focus on approaches from the MLoC field that were originally developed for a different application. We want to know how well they are suitable for our application: vulnerability detection. To do that, we expand our benchmark tool in Figure 1, select the most promising approaches in the MLoC field and evaluate them in the same way. Based on one of the three representations, the different approaches are evaluated individually.

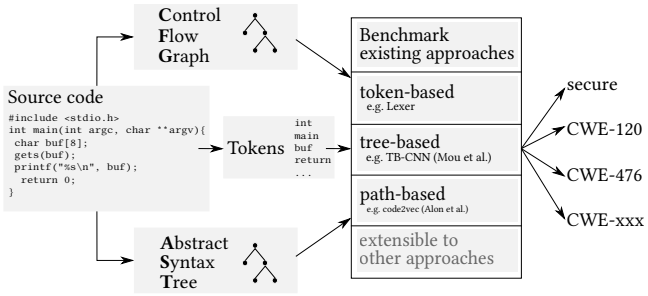

Figure 1: From source code to classification.

From the extracted tokens, ASTs and CFGs, we consider a token-based, a tree-based and a path-based approach. The main purpose of the token-based part is to find an optimal embedding that matches the context of the token and represents its semantics. Tree-based means to using a graph representation of the code such as an AST or a CFG as input to a machine learning pipeline. This could be a graph CNN following the approach of Kipf and Welling [8] or a tree CNN following the approach of Mou et al. [13]. Among the path-based parts are frameworks like Code2Vec [3], that extracts paths from an AST. In addition, this benchmark tool can also be extended to other, newly developed approaches that we will look at in the course of this project.

4 FOCUS ON ACTIONABILITY

In general, static analysis tools often come with a high false positive rate, whereas dynamic analysis tools often provide a high number of false negatives [16]. Machine learning approaches inherently have false-positive and false-negative results. The user, who wants to improve her written code, wants to get results quickly. In practice it takes 3-8 minutes to check, if she should take the alarm seriously, and about 1 out of 50 alarms is a serious vulnerability [16].

The works presented in Section 2 have certain error rates, which is why they were not comparable with static analysis tools and have not been used in practice. We want to focus on the practicability of these analytical methods. To improve our work in terms of usability, we can summarize the following requirements from the survey [5]:

- Identifying the type of a vulnerability is important to assess the severity of vulnerabilities and to prioritize them effectively for the developer(s). We want to classify among the most commonly used CWEs [12]. The existing works classify mainly binary for secure and vulnerable. The prioritization of the findings could be a score that combines classification accuracy/probability with the severity/impact when exploited by an attacker.
- The ability to pinpoint to the source location of the vulnerability is important for saving debugging time. Recent work [9, 15] classify a code snippet at the function level. Ideal would be a classification at the line-of-code level. Nevertheless, code snippets at the function level are necessary to determine the context of the weakness. Explainable AI methods should be considered to trace the exact position of the classification decision.
- Analyzing a program requires various types of representations: syntactic, AST, control flow, dependencies and data flow information. Previous work [17] found the optimal combination of representations for recognizing code clones. We want to find a suitable representation for the detection of vulnerabilities.

5 COMBINING REPRESENTATIONS TO AN IMPROVED MODEL

The results of the presented benchmark tool are intended to avoid disadvantages of the individual approaches. The results may also differ for the particular CWE type associated with the representation. A combination of approaches to a feature model could prevent false alarms and produce more accurate results by configuring the model through parameters. The configuration should be similar to static analysis tools, for which the user can customize the notifications of the tools. Another possibility is to weight the representations of code if it turns out that a representation is better suited to recognize a certain CWE. This could be the case if buffer overflows are detected with the help of CFGs [21]. The feature model should contain a number of configuration parameters, depending on the nature of the vulnerability that the user wants to focus on.

With ensemble learning techniques, different representations could be combined to vote for the created classifiers so that the lexical and semantic analysis can be combined. This has the advantage of obtaining more accurate prediction results and a more stable and robust model [14]. On the other hand, this technique also reduces interpretability of the model, which we also want to improve. In addition, the computing time increases with ensemble learning [14].

Another possibility would be to create a meta-embedding [22], which is a combined embedding of different representations in the same vector space. This ensures the traceability to the code representation, but the prioritization by weights for a vector is only possible to a limited extent. Another advantage is the additional inclusion of text components such as comments or documentation of the code, which then supports the classification.

Furthermore, a model could be created in which the representation is not embedded separately, but one chooses an architecture so that the individual features of the representation can influence each other. For example, a higher weight for a path in the CFG results in a higher weighting for the associated sub-tree in the AST. Which is the best method for combining the code representations will be empirically determined in the experiments.

6 EVALUATION PLAN

We will collect source code for training and testing at a function level for the C and C++ programming languages. Our data sources are public available source code archives, such as open source projects and Github/Gitlab repositories. After crawling various repositories, the source code functions are extracted from the files. The labeling should be done using various static code analysis tools to obtain a semi-automatic method and also to obtain a high coverage of the vulnerabilities found. Their outputs are assigned to the most common CWEs, and a classification for secure and vulnerable components is performed.

Meanwhile, some training data problems occur: (1) we have synthetic data through synthetic labeling with the analysis tools. This could be prevented by a manually curated data set with verifying but less and an insufficient number of examples of training. Further training examples could be generated by mutation on already labeled code examples (perturbation), which again makes the examples unrealistic. (2) Existing vulnerability databases with real vulnerabilities and therefore realistic labels are not available in sufficient form. Manual labeling on large scale is too costly. (3) Due to the re-usability principle of source code, examples with exactly the same functionality written in different ways are rarely available. Data augmentation techniques will help to generate more training examples. (4) Examples labeled as secure may also contain vulnerabilities, that are not detected by the static analysis tools and can therefore lead to incorrect classification decisions. We will address these issues by combining synthetic and real examples analogous to existing work [15], to have enough examples overall and a certain amount of real examples. The snippets of code should come from the same or similar domain. A solution for the labeling problem could be some kind of active learning by reporting detected vulnerabilities from developers via pull requests. We will examine how well this is suitable for our application.

7 CURRENT STATE AND FUTURE WORK

The software analysis of security-related bugs has a long history of research and is still not optimal. Machine learning aims to overcome the problems in this area, but raises several other issues as described in the previous chapters. We find that machine-learning-based state-of-the-art approaches for vulnerability detection are not sophisticated and still need much development.

Yet, we are in the initial stages of this project and are currently implementing an integration platform to benchmark the state-of-the-art methods. In this work, we address the following contributions: In addition to publications in journals and conferences, we aim to develop a benchmark framework for vulnerability detection approaches on code, which can be also used by other researchers to compare token-based and graph-based approaches in the MLoC field. At the end of this project, we want to have a combined feature model for detecting vulnerabilities with the focus on ease of use. We also expect a more precise determination of security bugs defined using classification measures such as F1 score or Matthews correlation coefficient, our feature model compared to standard models.

The author of this publication is a scientific assistant at the Institute of Data Science, German Aerospace Center (DLR) in the Secure Software Engineering group and an external doctoral student at Technische Universität Ilmenau, supervised by Prof. Patrick Mäder.

REFERENCES

[1] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. 2017. A Survey of Machine Learning for Big Code and Naturalness. *arXiv:1709.06182 [cs]* (Sept. 2017). http://arxiv.org/abs/1709.06182 arXiv: 1709.06182.

[2] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2017. Learning to Represent Programs with Graphs. *arXiv:1711.00740 [cs]* (Nov. 2017). http://arxiv.org/abs/1711.00740 arXiv: 1711.00740.

[3] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. code2vec: Learning Distributed Representations of Code. *arXiv:1803.09473 [cs, stat]* (March 2018). http://arxiv.org/abs/1803.09473 arXiv: 1803.09473.

[4] Zimin Chen and Martin Monperrus. 2019. A Literature Study of Embeddings on Source Code. *arXiv:1904.03061 [cs, stat]* (April 2019). http://arxiv.org/abs/1904.03061 arXiv: 1904.03061.

[5] Seyed Mohammad Ghaffarian and Hamid Reza Shahriari. 2017. Software Vulnerability Analysis and Discovery Using Machine-Learning and Data-Mining Techniques: A Survey. *ACM Comput. Surv.* 50, 4 (Aug. 2017), 56:1–56:36. https://doi.org/10.1145/3092566

[6] Gustavo Grieco, Guillermo Luis Grinblat, Lucas Uzal, Sanjay Rawat, Josselin Feist, and Laurent Mounier. 2016. Toward Large-Scale Vulnerability Discovery Using Machine Learning. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy (CODASPY '16)*. ACM, New York, NY, USA, 85–96. https://doi.org/10.1145/2857705.2857720 event-place: New Orleans, Louisiana, USA.

[7] Gong Jie, Kuang Xiao-Hui, and Liu Qiang. 2016. Survey on Software Vulnerability Analysis Method Based on Machine Learning. In *2016 IEEE First International Conference on Data Science in Cyberspace (DSC)*. IEEE, Changsha, China, 642–647. https://doi.org/10.1109/DSC.2016.33

[8] Thomas N. Kipf and Max Welling. 2016. Semi-Supervised Classification with Graph Convolutional Networks. *arXiv:1609.02907 [cs, stat]* (Sept. 2016). http://arxiv.org/abs/1609.02907 arXiv: 1609.02907.

[9] Jorrit Kronjee, Arjen Hommersom, and Harald Vranken. 2018. Discovering Software Vulnerabilities Using Data-flow Analysis and Machine Learning. In *Proceedings of the 13th International Conference on Availability, Reliability and Security (ARES 2018)*. ACM, New York, NY, USA, 6:1–6:10. https://doi.org/10.1145/3230833.3230856 event-place: Hamburg, Germany.

[10] D. Li, H. Chen, H. Cao, and L. Ming. 2018. Data-Driven Vulnerability Pattern Analysis for Fuzzing. In *2018 IEEE Third International Conference on Data Science in Cyberspace (DSC)*. 930–935. https://doi.org/10.1109/DSC.2018.00148

[11] K. Liu, D. Kim, T. F. Bissyande, S. Yoo, and Y. Le Traon. 2018. Mining Fix Patterns for FindBugs Violations. *IEEE Transactions on Software Engineering* (2018), 1–1. https://doi.org/10.1109/TSE.2018.2884955

[12] Bob Martin, Mason Brown, Alan Paller, and Dennis Kirby. 2011. 2011 CWE/SANS Top 25 Most Dangerous Software Errors. https://cwe.mitre.org/top25/. (2011). Online; accessed: 2019-01-07.

[13] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2014. Convolutional Neural Networks over Tree Structures for Programming Language Processing. *arXiv:1409.5718 [cs]* (Sept. 2014). http://arxiv.org/abs/1409.5718 arXiv: 1409.5718.

[14] X. Qiu, L. Zhang, Y. Ren, P. N. Suganthan, and G. Amaratunga. 2014. Ensemble deep learning for regression and time series forecasting. In *2014 IEEE Symposium on Computational Intelligence in Ensemble Learning (CIEL)*. 1–6. https://doi.org/10.1109/CIEL.2014.7015739

[15] Rebecca L. Russell, Louis Y. Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul M. Ellingwood, and Marc W. McConley. 2018. Automated Vulnerability Detection in Source Code Using Deep Representation Learning. In *17th IEEE International Conference on Machine Learning and Applications, ICMLA 2018, Orlando, FL, USA, December 17-20, 2018*. 757–762. https://doi.org/10.1109/ICMLA.2018.00120

[16] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from Building Static Analysis Tools at Google. *Communications of the ACM (CACM)* 61 Issue 4 (2018), 58–66. https://dl.acm.org/citation.cfm?id=3188720

[17] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2018. Deep learning similarities from different representations of source code. In *Proceedings of the 15th International Conference on Mining Software Repositories - MSR '18*. ACM Press, Gothenburg, Sweden, 542–553. https://doi.org/10.1145/3196398.3196431

[18] John Viega and Gary McGraw. 2011. *Building Secure Software: How to Avoid Security Problems the Right Way (Paperback) (Addison-Wesley Professional Computing Series)* (1st ed.). Addison-Wesley Professional.

[19] F. Wu, J. Wang, J. Liu, and W. Wang. 2017. Vulnerability detection with deep learning. In *2017 3rd IEEE International Conference on Computer and Communications (ICCC)*. 1298–1302. https://doi.org/10.1109/CompComm.2017.8322752

[20] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *2014 IEEE Symposium on Security and Privacy*. IEEE, San Jose, CA, 590–604. https://doi.org/10.1109/SP.2014.44

[21] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. 2013. Chucky: Exposing Missing Checks in Source Code for Vulnerability Discovery. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13)*. ACM, New York, NY, USA, 499–510. https://doi.org/10.1145/2508859.2516665 event-place: Berlin, Germany.

[22] Wenpeng Yin and Hinrich SchÃijtze. 2015. Learning Meta-Embeddings by Using Ensembles of Embedding Sets. *arXiv:1508.04257 [cs]* (Aug. 2015). http://arxiv.org/abs/1508.04257 arXiv: 1508.04257.