

A Line-Level Explainable Vulnerability Detection Approach for Java

Balázs Mosolygó[1], Norbert Vándor[1], Péter Hegedűs[1,2(✉)], and Rudolf Ferenc[1]

[1] Software Engineering Department, University of Szeged, Szeged, Hungary
hpeter@inf.u-szeged.hu
[2] FrontEndART Ltd., Szeged, Hungary

Abstract. Given our modern society's level of dependency on IT technology, high quality and security are not just desirable but rather vital properties of current software systems. Empirical methods leveraging the available rich open-source data and advanced data processing techniques of ML algorithms can help software developers ensure these properties. Nonetheless, state-of-the-art bug and vulnerability prediction methods are rarely used in practice due to numerous reasons. The predictions are not actionable in most of the cases due to their level of granularity (i.e., they mark entire classes/files to be buggy or vulnerable) and because the methods seldom provide explanation why a fragment of source code is problematic. In this paper, we present a novel Java vulnerability detection method that addresses both of these issues. It is an adaptation of our previous method for JavaScript that is capable of pinpointing vulnerable source code lines of a program together with a prototype-based explanation. The method relies on the word2vec similarity of code fragments to known vulnerable source code lines. Our empirical evaluation showed promising results, we could detect 61% and 41% of the vulnerable code lines by flagging only 43% and 22% of the program code lines, respectively, using two of the best detection configurations.

Keywords: Software security · Vulnerability prediction · Explainable prediction model · Empirical study

1 Introduction

Software systems have become a fundamental part of our every-day lives. They not only control critical infrastructure (power plants, air traffic, manufacturing)

This research was supported by the Ministry of Innovation and Technology of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021-NVA funding scheme and the framework of the Artificial Intelligence National Laboratory Program (MILAB). The research was partly supported by the EU-funded project AssureMOSS (Grant no. 952647). Furthermore, Péter Hegedűs was supported by the Bolyai János Scholarship of the Hungarian Academy of Sciences and the ÚNKP-21-5-SZTE-570 New National Excellence Program of the Ministry for Innovation and Technology.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2022
O. Gervasi et al. (Eds.): ICCSA 2022 Workshops, LNCS 13380, pp. 106–122, 2022.
https://doi.org/10.1007/978-3-031-10542-5_8

or handle and store sensitive data (bank card details, health records, personal documents) but serve our convenience as well (smart TV, smart watches, smart homes, etc.).

The abundance of research results in the area promises a decent practical solution for managing bugs and security issues at the development phase (i.e. before the system goes public). Bug and vulnerability prediction models can be used during the development, even integrated within the CI/CD pipelines, to detect problematic or vulnerable code introduced by the developers on-the-fly. Having such an early alarming mechanism could help in fixing critical problems fast in an early phase, before malicious users even have the chance to exploit them.

In this paper, we focus on a line-level vulnerability detection method that is tailored to Java programs, which addresses both of the above mentioned shortcomings of existing approaches. The proposed technique builds on, adapts and improves our very promising previous work [10] on detecting vulnerable lines in JavaScript programs together with a prototype based explanation. We use the project KB [14] manually validated vulnerability dataset containing hundreds of vulnerability fixing commits (that can be mapped to a CVE [9] entry) as a basis of creating a set of known vulnerable line repository (VLR). Using the word2vec [8] embedding technique on the lexical tokens of these lines, we build a golden set of vectorized features of vulnerable lines. Then, we scan the subject systems line by line and determine the line in our golden set that is the most similar to the analyzed line. If the cosine distance is below an empirically established threshold, we declare the line to be vulnerable.

To adapt our JavaScript approach to Java, we had to perform the following steps:

- Create an entirely new VLR, a golden set of vulnerable Java lines;
- Adapt our code lexing approach to produce Java tokens and re-train the word2vec embeddings on a large Java corpus;
- Develop an approach for reducing the size of the VLR to maintain practical performance while keeping the prediction performance.

We ran an empirical evaluation on the created method, where we scanned 1282 commits' lines form 205 Java projects. Our method proved to be generalizable, meaning that we were able to adapt it to Java programs and re-run a similar experiment to evaluate its performance. We found that our method works slightly worse for Java programs, but it could be improved by fine tuning the dictionary of tokens we take into consideration. It shows that different tokens play a major role in vulnerability prediction in the different languages, which is quite intuitive. We were able to reduce the large size of our Java VLR (from 10,000 to 3,200) to maintain practical applicability of the method without losing significant predictive power. In the two best setups, our line-level prediction model was able to identify 61% and 41% of the vulnerable code lines by flagging only 43% and 22% of the program code lines, respectively.

The remaining of the paper is organized as follows. In Sect. 4 we list the works related to our approach. Section 2 describes the methodology we used to

build the prediction model. We present and explain the results of our empirical evaluation in Sect. 3. Section 5 lists the possible threats to validity of our work, while Sect. 6 concludes the paper.

2 Methodology

2.1 Background

In a previous work [10], we have created a method to effectively mark vulnerable lines in JavaScript programs. We used a simple word2vec based solution, where we took the average vectors of words found in code lines and checked whether there were similar code lines in a pre-constructed vulnerable line repository. The solution includes several rules that we created to refine the prediction, such as not leting the method mark lines that are only one word long, or preferring lines that consisted of more unique tokens.

In this paper, we present the results of our efforts to transfer our method to Java. Examining our findings both in terms of creating a valuable tool and gaining a deeper understanding of the complex structures of programming languages.

2.2 Motivation

Advantages of Java and Limits of JavaScript. Our JavaScript results, while promising, were not indicative of the true potential of our approach, since the available data was limited. JavaScript is not widely used for critical systems, as such, it contains relatively few known and documented vulnerabilities. This limited our abilities to provide our method with vulnerable lines to be used as prediction bases, and to run tests of the proper size.

Project KB [14] provides a large and validated knowledge base of vulnerabilities in Java programs.[1] Using this, a more realistic testing environment can be set up, making the results more likely to reflect the real capabilities of our approach.

Testing the Generalizability. Changing the examined language is a step towards understanding our method's limitations in terms of generalizability. Our approach does not clearly rely on any language specific properties of the examined JavaScript projects. Knowing how easily the model can be adapted, if it is even needed, can also show a direction for future improvements. Not relying on language specific information would keep a generalizable model flexible, which would be important to keep in mind.

However, if the method proves ineffective in its new application, the results it produces can still be used to push our progress forward. Failure in this case could be caused by the method using structural information yet unnoticed. If these hidden properties indeed exist, they could be used to augment our current method or be taken into account during later projects.

[1] https://sap.github.io/project-kb/.

2.3 Highlights of the Original Algorithm

In this section, we will discuss the method [10] we have developed for JavaScript in detail to introduce the terminology already established in our previous work. The process of vulnerability prediction can be broken down into 3 phases (for an overview, see Fig. 1), two of which do not need to be repeated every time.

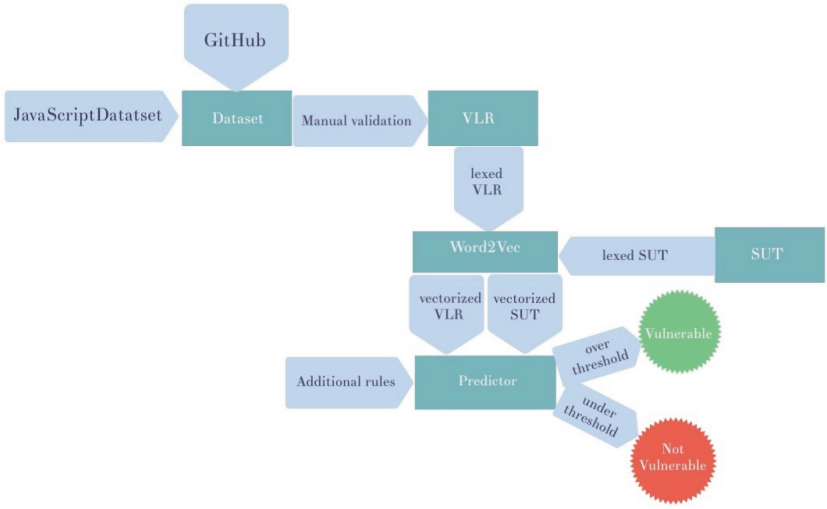

Fig. 1. Overview of our process

Phase 1. The first, and the most crucial step when it comes to getting accurate predictions is creating a word2vec model to be used to represent lines as vectors. This will be used to determine whether lines are similar to one an other. Word or token type count should be reduced to a point where it becomes manageable, since the complexity of code in term of words used can vary greatly. We used ANTLR[2] as a tokenizer, which allowed us to create the corpora for training word2vec models with different levels of abstraction. We used the GitHub Java Corpus [1] for training the word2vec model as a representative sample of Java code from open source projects.[3]

Phase 2. As mentioned before, our method looks for similarities between lines that are known to be vulnerable and the line that is being processed. To do this, we need a set of lines that are known to be vulnerable to serve as a ground truth. So naturally, the next step is to create a Vulnerable Line Repository or VLR for short that will serve as the basis of our prediction. As mentioned in Subsect. 2.2, we used the manually curated data published in project KB for this purpose.

[2] https://www.antlr.org/.
[3] https://groups.inf.ed.ac.uk/cup/javaGithub/.

This data set contains 1,282 commits from 205 open-source Java projects, which fix 624 publicly disclosed vulnerabilities (i.e. CVEs). The testing environment was set up by randomly splitting up the vulnerability database so that 90% of it would go in the VLR and 10% could be used for testing.

The knowledge base does not directly contain the vulnerable lines we need, only the projects and commits that contain their fixes. To extract the information we need, we simply clone the repository, checkout to the given commit, get its parent, and using the `git diff` command get the lines that have been removed or changed. We ignore any lines that originate from test files or that are one token long. Assuming that only the given vulnerability has been fixed in a commit, this heuristic would collect all lines that contributed to the vulnerable behavior of the code. To minimize the chance of a commit inducing other changes next to vulnerability fixes, we looked through every commit to check if they were merges or containing additional code changes other than vulnerability fixes based on their commit messages. After removing the duplicate lines, our VLR contained around 10,000 vulnerable lines.[4]

Phase 3. Once finished with the preparation, we can apply our method for predictions. This is done by taking a file from the System Under Test (SUT) and comparing each of its lines' word2vec representation to those in the VLR. A line's word2vec value is determined by the average of its words' vectors.

Using this distance, we can calculate a confidence value that will be used to decide whether a line should be marked vulnerable or not. The confidence value is calculated as follows:

$$Conf(line_{code}) = 1 - min_e(cos(\boldsymbol{v}(line_{code}), \boldsymbol{v}(line_{VLR_e})))$$

Using the confidence value a prediction can already be made. For this however, a threshold value is required to decide which line to be considered vulnerable. We will refer to this threshold as the method's *trip value*. The confidence value of the method will fall between 0 and 1, so it naturally follows that the trip value will also be within this range.

Nonetheless, we do not use this confidence score directly to make the prediction as our previous empirical evaluations showed, that this alone produces lots of false positives. Therefore, we have created multiple so called "rules", which can be used to decrease the false positive rate of our method. They are not specific to JavaScript, therefore they can be applied in this new environment without major modifications. The rules are applied after the initial prediction phase, and modify the base confidence score to decrease the method's false positive rate.

We have tested all of our previous rules [10], from which we kept the three most efficient ones: **no_one_word_line**, **prefer_complex**, and **surrounded**. The rules are applied by taking the average of the rules' score and the word2vec prediction's values, while the `no_one_word_line` rule is applied as a direct filtering or in combination with every other rule. The application of `prefer_complex` rule and the `surrounded` rule together did not produce productive results, therefore it will not be discussed.

[4] https://doi.org/10.5281/zenodo.5761680.

2.4 Changes to Adjust Our Method to the New Language

During the first tests we ran in the new environment, it became clear that the method as it was in JavaScript needs adjustments to work well in Java.

Using the adapted model, we examined a reduction in the model's ability to properly determine similarity between lines. Lines that, to the human eye, have no connection have started getting matched as being close to one another. The reason for this behavior was an increase in the average lines' complexity that got lost during the pre-processing stage of prediction. Java code tends to be more verbose, therefore more readable to human eyes. Our simple method of comparing the averages of lines' words was not prepared for the sudden increase in token count, especially without any change in the amount of unique tokens. Even a task as simple as printing out a string to the default output takes 9 tokens in Java, as opposed to the 6 in JavaScript. This was a drawback in our original case since even though the line became more complex and verbose, our model did not gain any extra information. To address this issue, we had to investigate our first research question:

> **RQ1:** Does extending the dictionary used by the word2vec model help in capturing the nuances of the language at hand?

Our VLR in JavaScript contained a few hundred lines and the files examined were also relatively short. In this new setting however, the size of our knowledge base grew to over 10 times of its previous size. The examined files also increased in size, while retaining the same vulnerable line count. Not only is it more difficult to find the lines we are looking for, but it takes significantly more time since our method is essentially a matching algorithm with an $O(N * M)$ time complexity, where N is the number of lines in the SUT and M is the size of the VLR. Reducing the time it takes to create predictions is crucial not only for the development of the method but for its usability as well. To increase the performance without changing the fundamentals of our algorithm, we had to reduce the size of the VLR. This however, can not be done by simply removing some lines randomly, since that might impact performance. Therefore, we faced our second research question:

> **RQ2:** Is it possible to reduce the size of the VLR without losing predictive power?

Reducing the Size of the VLR. The problem of the overly large VLR as mentioned in Subsect. 2.4 causes significant slowdown. To combat this, we aimed to decrease the size of the VLR without significantly reducing the amount of information it contains. We did this by removing lines that are not likely to take productive part in the prediction process.

We created a second repository of lines, this time saving those that were not vulnerable. This was done by collecting the lines that replaced the vulnerable ones in fix commits. Since these lines were created for removing issues, they can be considered to be non-vulnerable. Simply adding random lines from

the repositories would have not been a good strategy as they may contain yet unknown vulnerabilities. This risk is present with the fixing lines as well since they may have some hidden flaws, which only get uncovered later in the project development, however the probability of this should be minimal.

Once the new repository was set up, we used its contents to eliminate lines from the VLR that were too similar to a non-vulnerable lines. This should not only reduce the VLR size but may decrease the amount of false positives the method generates as the lines that are very similar to a non-vulnerable line cannot trigger a false prediction anymore.

New Tokens in the Lexer. Introducing new tokens that are general enough to appear regularly but do not inflate the dictionary is challenging. For a start, we used the tokens with which the method performed the best in JavaScript (adapted to Java). Our additions to an otherwise standard lexing method were special string literals, which aim to differentiate between cases where the content of the string is not necessarily relevant to its function, and cases where its value directly influences functionality as it would be the case with strings containing SQL commands, for example.

In Java the original set of tokens did not prove to be satisfactory when it comes to capturing enough of the lines' content to allow for meaningful predictions. We expanded the tokens that may abstract too much information. The most common token was without question the `Identifier` token. Identifiers are used commonly in every language, but in Java's object oriented environment most of the functions are created through classes, methods, and in general variables, all parsed into the token `Identifier` once the lexing is done.

To overcome the issue, we have introduced 4 sub-classes of identifiers to preserve as much of the functionality as possible, without too much of a dictionary size explosion. These are the `VariableIdentifier`, `MethodIdentifier`, `ObjectIdentifier`, and `ArrayIdentifier`.

Since unlike in C++ for example, certain operations can only be executed on a specific subset of objects, we used the token following an `Identifier` instance to determine its type. An `Identifier` will be classified as a `MethodIdentifier` if its following token is a (. Similarly, `ObjectIdentifiers` and `ArrayIdentifiers` need to be followed by a `.` and a [, respectively. The `VariableIdentifier` tokens were harder to extract, we identified them by their next token being in the possibilities displayed in Table 1. This heuristic does not guarantee that we will be able to classify every `Identifier`, however, it introduces significant variety that our method leverages to produce results of a higher quality.

Table 1. Tokens we use to identify a VariableIdentifier

=	+=	−=	&=
*=	/=	|=	≙
%=	«=	»=	»>=
++	−		

2.5 Metrics Measured

In this section, we will briefly discuss the metrics we measured to evaluate our models. Table 2 contains a short description for all of them.

Table 2. Our measures and their short descriptions

`file_lines`	The total number of lines
`flagged_lines`	The number of lines flagged as vulnerable
`vuln_lines`	The number of lines confirmed to be vulnerable
`flagged_vuln_lines`	The number of confirmed vulnerable lines flagged as vulnerable
`%_flagged`	The percentage of `file_lines` flagged as vulnerable (i.e., efficiency)
`%_is_vuln`	The percentage of `flagged_lines` confirmed to be vulnerable (i.e., precision)
`%_vuln_flagged`	The percentage of `vuln_lines` being flagged (i.e., recall)

It is important to mention that our goal was not necessarily to find all vulnerable lines, rather to decrease the amount of lines that need to be checked, while still touching most if the issues.

2.6 Overview

In this section, we have discussed the method we have used previously in the context of JavaScript, and the one we have introduced here to increase the performance of the method in the context of Java. Taking these additional steps was necessary as we wanted to answer a third research question:

> **RQ3:** Can the performance of our previous method for JavaScript be reproduced in Java?

3 Results

In this section, we discuss the results of the empirical evaluation of our proposed method with the changes mentioned in Sect. 2.

3.1 The Original Method for JavaScript

At first, as mentioned before we have tried to port the method as it was originally published in the context of JavaScript vulnerability detection without any major modifications. We used the parameters that performed best in the previous environment to test whether additional changes to the method were at all required. The results were less than ideal, see the method variant with the **O_** prefix (O as in Original) in Table 3.

We hypothesized that the reason for these results is the method's inability to correctly process the lines it is comparing. This could be because of the dictionary not allowing for an expressive enough representation. Java being a more verbose language than JavaScript leads to longer, more complex lines, and as such the gap between our lexed lines' information content and the original may increase. To address this, we increased the dictionary's size as described in Sect. 2.4 to decrease the reduction in information content between the processed and unprocessed lines.

During this initial run, we also encountered a major issue with the approach. Namely, its time complexity increased rapidly as the VLR's or the SUT's size increased. As the SUT's size cannot be influenced during the method run, the only way to speed up the process was to reduce the size of the VLR. Ideally, this reduction should occur without losing important information. We used a heuristic to collect the VLR's content as described in Sect. 2.4 so that we are able to remove certain lines that the heuristic wrongly added to the VLR, leading to a possible decrease in our method's false positive rate.

3.2 The Augmented Dictionary

The poor initial performance of our model as mentioned in Sect. 3.1 showed that we needed to increase our method's ability to process the lines' content accurately. Since the most common token as mentioned in Sect. 2.4 was `Identifier`, we chose to allow for more variety within them. Our new tokens have helped our method create more accurate predictions, as can be seen in Table 3 with the method variant having the **L_** prefix. Here, **L_** denotes lower similarity value. (See Sect. 3.3)

Since the new dictionary has lead to a clear improvement over the original version we started off using, we took it as our baseline during the tests for the reduction methods. As we will discuss in Sect. 3.3, the reduction created a significantly smaller VLR that was identical in predictive power than the full version. As a result of that, the original prediction (with the new tokens without VLR reduction) is not represented separately.

The improvement in performance in our opinion can be attributed to the increased understanding our method gains as the result. We hypothesize that a correctly extended dictionary leads to a better representation and as a result a more accurate prediction. Lines containing more of their original informational content should help us pair lines that are truly similar in function. The drawback, however, is that choosing an overly excessive dictionary might lead to an unnecessary increase in model size and complexity, which would in turn lead to increased prediction times. Balancing the dictionary's and in turn the model's complexity to keep the lexed lines as close to the original ones as necessary, while keeping the dictionaries size to a minimum could lead to further improvements in predictive performance without too much sacrifice on the usability front.

Based on the experiences, we can answer our first research question:

RQ1: Increasing the size and therefore complexity of the dictionary when lexing and creating the word2vec model increases the methods performance drastically. This is the case because the lexed lines created using our original dictionary fail to capture the complexity of the new environment (i.e. Java language).

3.3 The Impact of the Reduced VLR

As mentioned in Sect. 2.4, our VLR has been generated based on a heuristic. We ignored lines from files that can not contain relevant information, such as non-java files or even test files. However, chances of including lines that are not truly vulnerable should not be ignored. Adding non-vulnerable lines to the VLR may increase the method's false positive rate, and due to the time complexity of the matching algorithm any unnecessary inclusions should be avoided.

Table 3. The results produced by the method with different levels of reduction and using different filtering rules

method_variant	flagged_lines	vuln_lines	flagged_vuln_lines
O_nr	92973	21066	2087
L_nr	37439	4934	3385
H_nr	60006	16556	1907
O_no	79331	21066	1643
L_no	31865	4934	3012
H_no	50620	16556	1600
O_pc	57839	21066	1311
L_pc	20844	4934	2296
H_pc	63254	16556	1794
O_srd	42366	21066	1045
L_srd	16664	4934	2032
H_srd	8945	16556	518
method_variant	%_flagged	%_is_vuln	%_vuln_flagged
O_nr	40.64	2.24	9.91
L_nr	50.79	9.04	68.6
H_nr	33.67	3.18	11.52
O_no	34.68	2.07	4.03
L_no	43.23	9.45	61.04
H_no	28.41	3.16	9.66
O_pc	25.28	2.27	6.22
L_pc	28.28	11.02	46.53
H_pc	35.5	2.84	10.84
O_srd	18.52	2.47	4.96
L_srd	22.6	12.19	41.18
H_srd	5.02	5.79	3.13

To deal with both of these issues, we created a process, described in Sect. 2.4, to decrease the size of the VLR by eliminating lines that are likely to contain patterns not related to vulnerabilities.

Before any steps were taken to reduce its size, the VLR contained over 70,000 lines. Most of these, however, were identical, so a simple elimination of those elements greatly reduced its size. After this initial step, the non-vulnerable line based reduction process could be started. It works similarly to the prediction method, we check if any of the VLR's lines are close to the non-vulnerable ones, and if so, they are removed from the VLR. Two lines are considered close, if the distance between them is smaller than a predefined value, which we simply call the similarity value. Two separate tests were carried out, one with a similarity value of 0.01 and one with 0.1.

After the removal of repeated lines, the VLR was reduced to around 10,000 lines, and using the non-vulnerable lines, its size was further decreased to 3,200 and 350, respectively. Results of testing for both reduction methods can be seen in Table 3. The **L** prefix shows that the results belong to the test instance with the VLR given as a result of using a similarity value of 0.01. Similarly, the **H** prefix refers to a run using a VLR generated with a 0.1 similarity value.

The results of runs with the improved dictionary and original VLR mentioned in Sect. 3.2 are not represented separately, since their results were identical with the lower similarity value runs. Meaning that using the VLR consisting of 3,200 lines is nearly equivalent to the full 10,000 line version.

Therefore, we can answer our second research question.

RQ2: It is possible to significantly reduce the size of the VLR without losing the method's prediction performance. With a relatively high reduction ratio, the VLR is reduced to a third of its original size, yet the predictions based on it are practically equivalent with the original VLR.

The results produced by the high similarity value VLR, however, marks few lines, and even those are not usually correct. Applying that significant reduction already impacts the prediction power of the method.

3.4 Overview

We have attempted to recreate our previous success in JavaScript into a new environment, in order to both test our methods generalizability and to gain access to the more extensive datasets available for Java. The initial results were not ideal, however, we have fine-tuned our approach to get results very close to that we got for JavaScript.

We have created a method for Java that is capable of finding a large portion of the vulnerabilities in a given system. This shows that our method is generalizable, as it could be adjusted to a different language with minor modifications achieving comparable performance.

We have also found that the size of the VLR we use can be significantly reduced, while preserving the prediction performance.

We were able to show that while the base approach with word2vec similarity might perform slightly worse in this context, selecting the appropriate dictionary is crucial and boosts performance, similarly to the rules already established.

Considering the conclusions above, we can answer our third and possibly most important research question:

> **RQ3:** Although adopting the original algorithm without modifications performed slightly worse in the context of Java vulnerable line prediction, with small adjustments in the considered tokens and enhancing rules, we could achieve a comparable results to that observed in the context of JavaScript.

4 Related Works

While our approach to predicting vulnerabilities using machine learning is unique, there are already a number of related studies using other methods. We can group the works based on the granularity of their proposed vulnerability detection methods: file, class, and function-level. There are much fewer works targeting line-level prediction (mostly for bug prediction); to the best of our knowledge ours is the first line-level prediction model addressing vulnerability detection.

4.1 File-Level Predictions

Shin et al. [16] investigated three metrics - complexity, code churn, and developer activity - to see whether they are useful at detecting vulnerabilities. In their empirical case study, they looked at two widely used open-source projects: the Mozilla Firefox web browser and the Red Hat Enterprise Linux kernel. The results indicate that the metrics are discriminative and predictive of vulnerabilities, with the model using all three metrics predicted over 80% of vulnerabilities with a false positive rate of less than 25%.

In one of their other works, Shin et al. [17] found that faults (or defects) have some similarities to vulnerabilities that may allow developers to use traditional fault prediction models and metrics for vulnerability prediction. They again used the Mozilla Firefox web browser to conduct an empirical study, where 21% of files have faults, and 3% of files have vulnerabilities. Both of their models provided similar results: the fault prediction model had a recall of 83% and a precision of 11% at classification threshold 0.6, and the vulnerability prediction model had a recall of 83% and a precision of 12% at classification threshold 0.5. They concluded that, while both models behaved similarly, and traditional fault prediction metrics can substitute for vulnerability prediction models, they still require significant improvement.

Jimenez et al. [6] created VulData7, an extensible dataset and framework automatically collected from software archives. The dataset contains all reported vulnerabilities of four security-critical open-source systems: the Linux Kernel,

WireShark, OpenSSL and SystemD. The framework provides the vulnerability report data (description, CVE and CWE number, etc.), the vulnerable code instance and the corresponding patches, when available. Since this is a lot of data, additional processing is required before it can be used to predict vulnerabilities.

In their work, Neuhaus et al. [11] introduced Vulture, a tool that automatically mines existing vulnerability databases and version archives, and maps past vulnerabilities to components by relying on the dependencies between them. In their approach, a component is a header-source pair for c++ and a .java file for Java. They used an SVM for classifying the dependencies and function calls between the different components. Their predictor correctly predicted about half of all vulnerable components, and about two thirds of all predictions were correct.

4.2 Class-Level Predictions

Siavvas et al. [18] conducted a study investigating the relationship between software metrics and vulnerability types. They studied 100 widely-used Java libraries and calculated a range of software metrics and quantified them through static analysis. They found that these metrics may not be sufficient indicators of specific vulnerability types but are capable of differentiating between security-specific and quality-specific weaknesses. They also found that there are certain metrics which could be used to search for security issues and that between a number of those issues there might exist some important interdependencies.

Basili et al. [2] used object-oriented design metrics described by Chidamber and Kemerer [3] to predict fault-prone classes. With these metrics they could make predictions in the early phases of the software life-cycle using a statistical approach. In contrast, our approach is mainly based on machine learning.

Palomba et al. [12] built a specialized bug-prediction model that they used on classes with code smells. They evaluated how much these code smells contributed towards bugs, and found that components affected by the smells were more bug-prone. To achieve this, they used several prediction models, and found that the best results were using the Simple Logistic model.

In their work, Sultana [19] proposed a vulnerability prediction model based on traceable patterns by examining Apache Tomcat, Apache CXF and three stand-alone Java web applications. Traceable patterns are similar to design patterns, but they can be automatically recognized and extracted from the source code. In their study, they compared the performance of these patterns and traditional software metrics, concluding that patterns have a lower false negative rate and higher recall in detecting vulnerable code than traditional metrics. Besides class-level predictions, the study also focuses on function-level predictions as well.

4.3 Function-Level Predictions

Giger et al. [5] performed experiments on 21 open-source Java systems with their model based on change- and source code metrics that are typically used in bug

prediction. Their models reached a precision of 84% and recall of 88%. They also found that change metrics significantly outperform source code metrics.

Ferenc et al. [4] compared 8 different machine learning algorithms to determine the best one for predicting vulnerabilities in JavaScript functions. Their data set consisted of static source code metrics, vulnerability data from NVD[5] and patches obtained from GitHub.

Pascarella et al. [13] replicated a previous research on function-level bug prediction done by Giger et al. [5] on different systems, then proposed a more realistic approach. They found that the performance is similar to that of the replicated research when using the strategy of said research. However, when suing their more realistic approach, they experienced a dramatic drop in performance, with results close to that of a random classifier.

In their work, Saccente et al. [15] created Project Achilles, a Java source code vulnerability detection tool. They used the National Institute of Standards and Technology's Juliet Java Suite, which is a set containing thousands of examples of defective Java methods for several vulnerabilities. They implemented an array of Long-Short Term Memory Recurrent Neural Networks, to detect the vulnerabilities. Their tool employs various data preparation methods and can automatically extract functions from the source code. The result of running the tool is an n-dimensional vulnerability prediction vector. They found that this tool can achieve an accuracy higher than 90% for most of the vulnerabilities.

Li et al. [7] created the tool VulDeePecker, a deep learning-based vulnerability detection system that is capable of automatically extract and define features. They achieve this by defining so-called code gadgets: semantically related code lines that are not necessarily consecutive. First they extract library function calls, then generate backward slices from them. These slices then get assembled into code gadgets and transformed into a vector representation. They train a BLSTM neural network on the vectors, and use them to predict vulnerabilities by transforming the target source code into vectors and classifying them. With their tool, they managed to detect 4 vulnerabilities that were not reported in the NVD, only "silently" patched by the vendors.

4.4 Line-Level Predictions

Wattanakriengkrai et al. [20] found that, on average, only 1%–3% of lines are defective in a file. In their work they propose a framework called Line-DP to identify defective lines using a model-agnostic approach. In other words, they used a state-of-the-art explainable machine learning method, called LIME to identify so-called risky tokens and to provide information about why their model made a prediction. First, their framework builds a file-level model using code token features, then it searches for the risky tokens (code tokens that lead the file-level defect model to predict that the file will be defective). Any lines that contain risky tokens will be flagged as a defect. The authors created a case study of 32 releases of nine Java open-source systems. Their approach achieved a recall

[5] https://nvd.nist.gov.

of 61% and a false alarm rate of 47%, while needing around 10 s of processing time. These results are statistically better than the six baselines they compared their model to. Although their approach is similar to ours, they apply it to predict defects, while we specifically targeting vulnerability prediction.

5 Threats to Validity

In this section, we list the major threats to our work. For evaluating our method, we split the data randomly allowing commits of the same project to be present in both the VLR and testing set. The likelihood of this skewing our results is small, since the lexing of the source code will abstract away any project specific patterns, like identifier names or comments.

To derive a single vector for a whole line, we took the average of the word2vec vectors of the tokens in that line. This is a simple, yet reductive way of representing lines as it does not take into account the order of the tokens within the line. In our case, this does not pose a major threat as in programming languages the order of the tokens is relatively strict.

We build a repository of non-vulnerable lines to help us reduce the size of the VLR. However, there is a slight but non negligible chance of a non-vulnerable line being flawed, which only gets uncovered later in the project development (i.e. the fix for a vulnerability contains another vulnerability). Nonetheless, the probability of this should be minimal, therefore we do not expect any major effect of this threat on the final results.

6 Conclusions

In this paper, we presented an incremental work on our previous explainable method for line-level detection of vulnerabilities in JavaScript programs. Our current goal was to adapt the method to an entirely new context, namely to detect vulnerabilities in Java programs. Our replicated study in this new context addresses two issues of equal importance: i) to study and prove the generalizability of the method, and ii) to achieve a practically applicable tool for fine-grained vulnerability detection leveraging the rich data sources available in Java.

We found that adapting our previous method for Java line-based vulnerability detection as-is is feasible but leads to somewhat degraded performance. The major cause of this was the lack of expressiveness of our vocabulary used for the tokenization and word2vec embedding of the Java source code. Therefore, we extended the original vocabulary to better fit the new context and were able to improve the performance significantly. We faced another issue concerning practical applicability, the large size of the VLR slowed down the detection process. We proposed an enhancement of the algorithm, which reduces the size of the VLR, while keeping its prediction performance. We also revisited and fine-tuned the rules we developed for reducing false positive predictions. The explanation mechanism of the algorithm has not been changed, we can provide

the most similar vulnerable line from the VLR to serve as a prototype-based explanation for a decision.

We ran an empirical evaluation using the 205 Java projects and 1282 vulnerability fixing commits contained in the project KB dataset. We used the dataset to build the VLR (using 90% of the records) and to test our method as well (on the remaining 10%). We were able to reduce the large size of our Java VLR (from 10,000 to 3,200) to maintain practical applicability of the method without loosing significant prediction power. In the two best setups, our line-level prediction model was able to identify 61% and 41% of the vulnerable code lines by flagging only 43% and 22% of the program code lines, respectively. We consider our experiment to be an overall success since we were able to – with only minor modifications – port our method from a vastly different environment to Java.

In the future, we plan to adapt the method to other languages as well and further study the effect of token vocabulary on the performance. Improvement and extension of the applied rules are also amongst our future plans.

References

1. Allamanis, M., Sutton, C.: Mining source code repositories at massive scale using language modeling. In: The 10th Working Conference on Mining Software Repositories, pp. 207–216. IEEE (2013)
2. Basili, V.R., Briand, L.C., Melo, W.L.: A validation of object-oriented design metrics as quality indicators. IEEE Trans. Softw. Eng. **22**(10), 751–761 (1996). https://doi.org/10.1109/32.544352
3. Chidamber, S.R., Kemerer, C.F.: A metrics suite for object oriented design. IEEE Trans. Softw. Eng. **20**(6), 476–493 (1994)
4. Ferenc, R., Hegedűs, P., Gyimesi, P., Antal, G., Bán, D., Gyimóthy, T.: Challenging machine learning algorithms in predicting vulnerable JavaScript functions. In: Proceedings of the 7th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering, pp. 8–14. IEEE Press (2019)
5. Giger, E., D'Ambros, M., Pinzger, M., Gall, H.C.: Method-level bug prediction. In: Proceedings of the 2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, pp. 171–180. IEEE (2012)
6. Jimenez, M., Le Traon, Y., Papadakis, M.: Enabling the continuous analysis of security vulnerabilities with VulData7. In: IEEE International Working Conference on Source Code Analysis and Manipulation, pp. 56–61 (2018)
7. Li, Z., et al.: VulDeePecker: a deep learning-based system for vulnerability detection. In: Proceedings 2018 Network and Distributed System Security Symposium (2018)
8. Mikolov, T., Chen, K., Corrado, G., Dean, J.: Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781 (2013)
9. MITRE Corporation: CVE - Common Vulnerabilities and Exposures (2020). https://cve.mitre.org/. Accessed 29 Apr 2020
10. Mosolygó, B., Vándor, N., Antal, G., Hegedűs, P., Ferenc, R.: Towards a prototype based explainable JavaScript vulnerability prediction model. In: 1st International Conference on Code Quality, ICCQ 2021, pp. 15–25 (2021)
11. Neuhaus, S., Zimmermann, T., Holler, C., Zeller, A.: Predicting vulnerable software components. In: Proceedings of the ACM Conference on Computer and Communications Security, pp. 529–540 (January 2007)

12. Palomba, F., Zanoni, M., Fontana, F.A., De Lucia, A., Oliveto, R.: Smells like teen spirit: improving bug prediction performance using the intensity of code smells. In: 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 244–255 (2016)
13. Pascarella, L., Palomba, F., Bacchelli, A.: Re-evaluating method-level bug prediction. In: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 592–601 (2018)
14. Ponta, S.E., Plate, H., Sabetta, A., Bezzi, M., Dangremont, C.: A manually-curated dataset of fixes to vulnerabilities of open-source software. In: Proceedings of the 16th International Conference on Mining Software Repositories (May 2019)
15. Saccente, N., Dehlinger, J., Deng, L., Chakraborty, S., Xiong, Y.: Project Achilles: a prototype tool for static method-level vulnerability detection of java source code using a recurrent neural network. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW), pp. 114–121 (2019)
16. Shin, Y., Meneely, A., Williams, L., Osborne, J.A.: Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. IEEE Trans. Softw. Eng. **37**(6), 772–787 (2011)
17. Shin, Y., Williams, L.A.: Can traditional fault prediction models be used for vulnerability prediction? Empir. Softw. Eng. **18**, 25–59 (2011)
18. Siavvas, M., Kehagias, D., Tzovaras, D.: A preliminary study on the relationship among software metrics and specific vulnerability types. In: 2017 International Conference on Computational Science and Computational Intelligence - Symposium on Software Engineering (CSCI-ISSE) (December 2017)
19. Sultana, K.Z.: Towards a software vulnerability prediction model using traceable code patterns and software metrics. In: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 1022–1025 (2017)
20. Wattanakriengkrai, S., Thongtanunam, P., Tantithamthavorn, C., Hata, H., Matsumoto, K.: Predicting defective lines using a model-agnostic technique. IEEE Trans. Softw. Eng. **48**(5), 1480–1496 (2022)