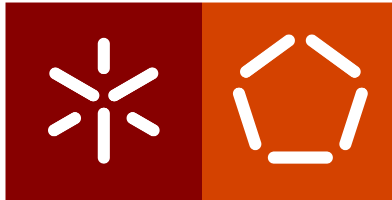


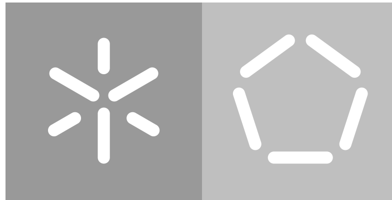
Universidade do Minho
Escola de Engenharia
Departamento de Informática

Samuel Gonçalves Ferreira

Vulnerabilities fast scan

Tackling SAST performance issues with Machine Learning

Academic Year 2018/2019



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Samuel Gonçalves Ferreira

Vulnerabilities fast scan

Tackling SAST performance issues with Machine Learning

Master dissertation
Master Degree in Informatics Engineering

Dissertation supervised by
Pedro Rangel Henriques
Daniela da Cruz

Academic Year 2018/2019

AUTHOR COPYRIGHTS AND TERMS OF USAGE BY THIRD PARTIES

This is an academic work which can be utilized by third parties given that the rules and good practices internationally accepted, regarding author copyrights and related copyrights. Therefore, the present work can be utilized according to the terms provided in the license bellow.

If the user needs permission to use the work in conditions not foreseen by the licensing indicated, the user should contact the author, through the RepositóriUM of University of Minho.

License provided to the users of this work

Attribution-NonCommercial-NoDerivatives
CC BY-NC-ND
https://creativecommons.org/licenses/by-nc-nd/4.0/

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.
I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

Signature: ____

Samuel Gonçalves Ferreira

ACKNOWLEDGEMENTS

Since I started my academic journey I met the most supportive and most helpful people I have ever met in my life. I would like to express my gratitude to everyone that was there in my best and worst moments throughout this journey and helped me become who am I today and be where I am today.

First of all, I would like to thank my supervisors, Daniela da Cruz and Pedro Rangel Henriques for all the guidance, advisory and for being able to deal with all the changes of plans throughout this work. What a great agile team!

I would also like to thank my fellow colleagues that were with me since my first day in college, for all the advice, availability, support and good times. Thank you, Rogério Moreira, Gustavo Andrez and Diogo Silva. After all the challenges and nights without sleeping, we finally made it and we never broke our bonds for any reason in the world.

Thank you, people from Checkmarx. Among all the amazing people that surround me every day at work, there are these special colleagues that really took some of their time to listen, think and help me whenever they could. Thank you for that, Ismael Vilas Boas, João Cruz and Adar Weidman, the Checkmarx wizards and thinkers. Now that my academic journey is almost over, I will be glad to say, as we say in the office ”Já estás Cândido!”.

Checkmarx itself, as a company, deserves all my gratitude for the amazing work environment, for the investment in training and for all the amazing social activities that I was included in, even though I was for a long time a part-time employee. All the flexibility, support and comprehension for my academic status show that this company really deserves my best effort in paying them back for everything they did for me.

To my non-academic buddies, who made it possible for me to have some distraction from all the work and academic duties, thank you Nelson Oliveira, Sandro Rodrigues, Miguel Costa, Fábio Lima (a.k.a. Castor), André Magalhães, Xavier Anjos, Filipe Costa, André Cardoso and Luís Pinho. I know that you guys will be always there when I need to get something off my chest, and that means a lot to me.

To Joana Cruz, for all the unconditional support and love. You literally made me a better person and changed the way I value relationships with everyone. Thank you for everything.

At last, I would like to dedicate the most loving and grateful ”Thank you” to my parents. They taught me the values of a humble, hard-working, responsible and perfectionist person. Although we all have our flaws and some times act in a regretful way, I promise to you I will make you both as proud as parents can be.

Thank you everyone

ABSTRACT

Nowadays, cybernetic attacks are a real threat that can compromise any individual, organization or company's integrity. Every day new cases are reported, that show the real damage cyber criminals can cause. Sensitive data exposure, identity theft, service malfunctioning or shutdown are just a few of the most common threats, which in many cases might impact companies either with financial loss or by damaging their reputation.

Population, in general, is becoming each time more aware of the risks of using electronic devices connected to the web and so are companies. With the rise of this awareness, over the last years, cyber security has become a major concern for Software companies.

This threat also led to the birth of the Software vulnerability detection market. Companies started commercializing Software and advisory to other companies, in order to keep them less exposed to cybernetic risks. There are many mechanisms and technologies used by these companies to identify vulnerabilities in applications. The most popular technology used to detect vulnerabilities is SAST (Static Application Security Testing) as it focus on the detection of vulnerabilities at the early stages of Software development. However, this requires the analysis of the source code, which in many cases, is huge and thus such analysis is too time consuming.

Being that the context and motivation for this dissertation, the goal is to investigate the possibility of performing source code analysis in a faster way, relying on machine learning approaches. Code embeddings, classification algorithms and clustering algorithms were the main approaches explored in this work.

Along the project, it was realized that some approaches performed better than others, in the task of detecting software vulnerabilities. Clustering algorithms, according to the performed experiments, are not suitable for the problem. Classification algorithms produced results that can be considered worthy of further investigation, but did not meet the established goals. After some failed attempts, this project demonstrated that it is possible to train a prediction model, based on code2seq approach, capable of detecting vulnerabilities in source code, with better performance and accuracy than classic SAST solutions (according to a specific set of experiments). Moreover, the used approach allows to easily extend the developed work to find vulnerabilities in any programming language.

RESUMO

Hoje em dia, os ataques cibernéticos são uma ameaça real que podem comprometer a integridade de qualquer indivíduo, organização ou empresa. Novos casos são reportados todos os dias, demonstrando os estragos que um atacante cibernético pode causar. Exposição de dados sensíveis, roubo de identidade, mau funcionamento ou corte de fornecimento de serviços são apenas algumas das ameaças mais comuns, que em muitos casos podem afetar as empresas com perdas financeiras ou prejudicando a sua reputação.

A Internet das Coisas cresce rapidamente todos os dias. Com a criação de dispositivos controlados por computador, desde relógios inteligentes até máquinas de lavar programáveis à distância, cada vez mais eletrodomésticos e outros acessórios eletrónicos estão diariamente ligados à Internet. Estes dispositivos tornam as nossas vidas mais fáceis e mais confortáveis, contudo podem expôr uma porta a redes corporativas para atacantes cibernéticos dotados e criminosos cibernéticos.

A população em geral está a ficar cada vez mais consciente dos riscos de usar dispositivos ligados à rede, e as empresas também. Com o crescimento desta consciência, ao longo dos últimos anos, a segurança cibernética tornou-se uma maior preocupação para as empresas de Software.

Esta ameaça também deu origem ao nascimento do mercado de deteção de vulnerabilidades em Software. As empresas começaram a comercializar Software e acomselhamento a outras empresas, com o objetivo de as manter menos expostas aos riscos cibernéticos. Há muitos mecasnismos e tecnologias usadas por estas empresas para detetar vulnerabilidades em aplicações. A tecnologia mais popular para identificar vulnerabilidades é o SAST (Static Application Security Testing), que se foca na deteção de vulnerabilidades logo nas primeiras fases do desenvolvimento de Software. Contudo, exige a análise de código-fonte, o que nos casos de programas de grandes dimensões pode ser muito demorada.

Sendo esta a motivação para esta dissertação, o objetivo é investigar a possibilidade de analisar código-fonte mais rapidamente, recorrendo a técnicas de *machine learning*.

Este trabalho demonstra que é possível treinar um modelo capaz de detetar vulnerabilidades em código-fonte, com bom desempenho e acerto aceitável. Para além disso, as tecnologias usadas permitem estender facilmente o trabalho desenvolvido para detetar vulnerabilidades em qualquer linguagem de programação.

CONTENTS

1 INTRODUCTION 1
- 1.1 Context and Motivation 1
- 1.2 Objectives 5
- 1.3 Research Hypothesis 5
- 1.4 Document Structure 5

2 STATE OF THE ART 6
- 2.1 Genesis project 6
- 2.2 Related Work 12

3 PROPOSAL 14
- 3.1 Proposed approaches 14
- 3.2 Technologies 14
  - 3.2.1 code2vec 15
  - 3.2.2 code2seq 16
  - 3.2.3 Scikit-Learn 17

4 DEVELOPMENT 18
- 4.1 Data sources 18
- 4.2 Approach 1 - code2vec embedding 19
- 4.3 Approach 2 - Using code vectors for clustering and classifiers 19
  - 4.3.1 Generating vectors 20
  - 4.3.2 Parsing 20
  - 4.3.3 Experimentation 23
- 4.4 Approach 3 - code2seq embedding 32
- 4.5 Supporting different languages 35

5 RESULTS 36

6 USE CASES 39

7 CONCLUSION AND FUTURE WORK 41
- 7.1 Conclusion 41
- 7.2 Future Work 42

LIST OF FIGURES

Figure 1 CxSAST Conceptual Engine pipeline 2
Figure 2 PoC data flow - High Level 6
Figure 3 PoC data flow - Overview 7
Figure 4 Genesis project data flow - 'Preprocessing' stage 8
Figure 5 PoC data flow - 'Clustering' stage 9
Figure 6 PoC data flow - 'Identifying vulnerable methods' stage 10
Figure 7 code2vec - Example predictions, from [1] 15
Figure 8 code2vec - AST path-context example, from [1] 16
Figure 9 code2seq - path-contexts example, from [2] 17
Figure 10 Approach 2 dataflow - overview 20
Figure 11 Approach 2 data structure - class diagram 22
Figure 12 Approach 3 data balancing 23
Figure 13 Clustering score functions in different datasets 27
Figure 14 Clustering score function comparison across different datasets 28
Figure 15 Training classifiers for detecting vulnerabilities - data flow overview 28
Figure 16 Classifiers results when trained with dataset *m* 29
Figure 17 Classifiers results when trained with dataset *l* 30
Figure 18 Classifiers results when trained with dataset *xl* 30
Figure 19 Classifiers results when trained with dataset *All* 31
Figure 20 Using CxSAST preprocessed projects to train models - from preprocessing to result analysis 33
Figure 21 Using original projects projects to train models - difference from last process 34
Figure 22 Accuracy comparison - models tested with CxSAST preprocessed files 37
Figure 23 Accuracy comparison - models tested with original source code files 37
Figure 24 Training times for the 5 different datasets 38

1

INTRODUCTION

In this introductory chapter there will be a detailed description of the context that motivated the theme of this dissertation, followed by the defined objectives, the research hypothesis and the document structure, that briefly explains what will be the content of every and each chapter in this document.

1.1 CONTEXT AND MOTIVATION

In the market of Software vulnerabilities detection, most of the commercialized products use static code analysis[3] as a mechanism to identify vulnerabilities in source-code. Checkmarx is a company that is currently among the leaders of this market, by providing one of the best Static Application Security Testing(SAST) solutions. However, even though these solutions serve its purpose - detecting software vulnerabilities - there are unavoidable computing costs in using Static Analysis techniques, and this limitation arose the curiosity of investigating other techniques to work around it. The following paragraphs will describe Checkmarx's SAST solution, to clarify the mentioned limitations.

Checkmarx, among other products, provides a static code analysis software, named CxSAST, whose purpose is to find security vulnerabilities in source code. Using static analysis to produce reliable and accurate results in vulnerabilities detection is not an easy task. It requires sophisticated algorithms and lots of processing. In the next paragraphs, a high-level description of the scanning pipeline that supports the referred static code analysis will be presented, allowing the reader to have some understanding of what this scanning process is about.

SCAN - What is called a scan is the process of reading a project's source code files and construct all the data structures needed to represent the source code, as well as other relevant data that is inferred from it.

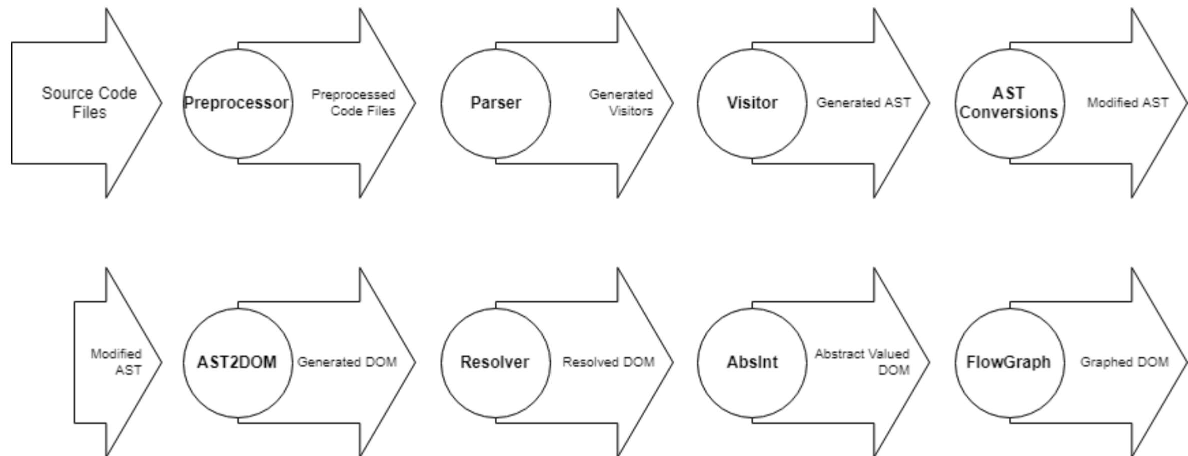


Figure 1: CxSAST Conceptual Engine pipeline

Figure 1 represents the stages of a project scan. This pipeline is not relevant to the thesis purpose, but it will be briefly described to let the reader have some understanding of how long and complex this scanning process can be, hence understand the motivation of this project.

PREPROCESSOR - This stage transforms text files into processed text files. In some cases this stage might be skipped because preprocessing is only needed under certain conditions, e.g. when a template engine[1] like Apache Velocity[2] is present in the project. Although it might be skipped in some cases, it can also take some time in other cases.

PARSER - The Lexical and Syntactic processors (generated automatically by ANTLR[3], giving the grammar of the Programming Language used in the source code under analysis) used in this stage will read the input program and generate tokens and derivation or parsing tree nodes for each file in the scanned project.

VISITOR - Each visitor contains the logic responsible for transforming parsing tree nodes in the corresponding Abstract Syntax Tree[4], taking the tokens' context into account.

ASTCONVERSIONS - For each programming language, there is a set of AST Conversions. Each AST Conversion is defined by a predicate (a condition) and an action (a function that takes a AST node and returns a transformed AST node). For each AST node built by the visitors, every conversion's predicate is tested against it and eventually it's action executed. In large projects there are millions/billions of AST nodes so this process can take several minutes, or even more than an hour.

1 See https://en.wikipedia.org/wiki/Web_template_system
2 See https://velocity.apache.org/
3 See https://www.antlr.org/about.html

AST2DOM - This stage is responsible for mapping AST nodes to a language-agnostic code representation - the DOM[4] - that will be essential for all the following steps.

RESOLVER - The Resolver stage performs traversals on the DOM to fill some information that couldn't be inferred until this point. For example, for each variable and method reference, the resolver will try to find its definition and complement its DOM node with the ID of the found definition (if found).

ABSINT - AbsInt is short for Abstract Interpretation[5]. Once again, the DOM is traversed, this time to infer about the possible values that each variable can have, the possible values that each method invocation can return, variables' types, and other useful information.

FLOWGRAPH - On this stage, a last DOM traversal is made in order to build a structure capable of representing both a Control Flow Graph[6] and a Data Flow Graph[7].

One aspect that should be noticed is that each stage depends on the full completion of the previous stage in order to perform correctly. Another thing to notice is that on most cases, when traversing the AST or the DOM (which includes 5 of the 8 stages) there are dependencies between different nodes, which means that these traversals are very hardly parallelizable (there have been some efforts on this direction that resulted in some improvement, but it is far from what would be ideal for scanning large projects).

QUERIES - After the scanning process is complete, the outcome produced is (just) a populated data structure. The way CxSAST allows to identify vulnerabilities is by querying this data structure (the DOM). Checkmarx has its own Query Language (CxQL) that allows to retrieve data from the DOM, apply filters and find flows between DOM nodes. For each vulnerability there is a specific query, that will gather the needed information from the DOM and produce a set of results. These results provide to the users a meaningful visual representation, so they can understand where are the flaws in their source code.

RESULT - A result is defined as an ordered set of DOM nodes. It can also be called a flow. For example, in a SQL Injection result, the flow usually starts pointing at a method invocation that reads text from the user (input). The second node of the flow might be the variable where the user input is stored and the third node a reference to that variable that is being concatenated to a SQL statement.

PERFORMANCE ISSUES - As previously described, to perform a full scan on a project there are several stages and lots of processing involved. One particular stage that has major

4 See https://en.wikipedia.org/wiki/Document_Object_Model

impact on the scan time, is the creation of Flow Graphs. Checkmarx deals with customers that need to scan projects with millions of lines of code, which are likely to be converted to billions of DOM nodes. Traversing all these nodes to infer flows, definitions and other data takes a long time, even with some parallelization techniques (where possible, since most of the used algorithms are far from parallelizable). Just to point another topic (among others) that can cause some increased processing time, there is the matter of supporting frameworks. Both of the topics mentioned will be better described in the following paragraphs to let the reader have a better notion of why there isn't a way of performing such tasks in a shortly manner.

SUPPORTING FRAMEWORKS - Even with a good stable support for several languages, every day new frameworks are born and as expected, Checkmarx's customers use them. By using frameworks that change a language's native behaviour, when one scans a project with CxSAST, it is likely to fail at constructing a DOM that correctly represents the source code. This is a bad thing, because when there isn't a correct DOM, there might be lots of bad results (False Positives) or complete absence of results. For this reason, there is the need to add mechanisms able to detect when certain frameworks are present in a project, and trigger additional processes that will change the way the code is interpreted. Depending on the framework and on the project size, this added processing can be costly.
For example, let's assume we want to provide support for scanning a project built with ReactJS[5]. It is a framework based on Javascript, but it has its specific functions, flows and syntax (JSX[6]). Let's assume that we already have support for Javascript language. In order to support ReactJS many changes would be needed in different stages of the pipeline. The parser would need changes in order to correctly build AST for JSX elements. AST Conversions would be necessary, for example, to be able to identify, inside a 'render' method, that when 'this.state' or 'this.props' is referenced, these references have to be linked to the class members, which is a specific linkage from ReactJS and would not have the same meaning in pure Javascript. It would probably need some actions at the Resolver stage to link all the references correctly.

At this point, it is expected that the reader fully understands the performance limitations of the Checkmarx product, as well as all the products that follow a similar architecture. It is also expected that the reader knows some of the vocabulary mentioned (i.e. what a scan is; what a flow/result is), so that the terminology used in the next chapters can be fully understood.

5 See https://en.wikipedia.org/wiki/React_(JavaScript_library)
6 See https://en.wikipedia.org/wiki/React_(JavaScript_library)#JSX

1.2 Objectives

The main goal of this work is to develop a mechanism capable of detecting vulnerabilities in source code of different programming languages, relying on machine learning techniques. This mechanism should be better than other known techniques in terms of performance and should not neglect the accuracy of the results. There is no specific constraint in terms of how to accomplish it, being the goal to research available techniques, experiment different approaches and report the results and conclusions of each of them.

1.3 Research Hypothesis

With this work, it is intended to prove that it is possible to detect vulnerabilities in source code of different languages, using machine learning techniques such as code embedding [8], clustering and classification algorithms, in a way that outperforms other techniques in terms of performance and doesn't compromise accuracy.

1.4 Document Structure

In this section it is intended to present a brief description of the following chapters, in order to let the reader have some knowledge of their contents. The second chapter, "State of the Art", starts with a detailed explanation of the context that motivated the theme of this dissertation. Also in the second chapter, there is a section dedicated to related work that was helpful to understand the already existing approaches, techniques and results. The third chapter, "Proposal", focuses on explaining the defined strategy, used technologies and system architecture to prove the thesis hypothesis. The fourth chapter can be seen as a technical report of all the experimented approaches, where all the decisions made throughout the development phase will be explained, as well as the development procedures. In the fifth chapter all the obtained results from the different approaches will be summarized, and some comments will be expressed about them. Chapter 6 will address the possible use cases for the developed work, by detailing how it can be used in different applications. The seventh and last chapter will contain the conclusions from all the research made, as well as proposals for future work on top this research.

2

STATE OF THE ART

This chapter is divided in two sections. The first one will describe, in a detailed manner, a previous project developed by Checkmarx in order to evaluate the feasibility of using clustering and classifiers to try to predict if a method (i.e. a function written in source code) is vulnerable or not. The second section presents some related researches that helped in making decisions related to this project, as well as their contributions and their limitations.

2.1 GENESIS PROJECT

GENESIS PROJECT Some Checkmarx employees have started a project on this matter which revealed some interesting results. The strategy was to analyze several C# projects at a method level, by representing methods in a way that can be used for clustering, clustering them and try to find a way of identifying vulnerable methods.
To explain how this experiment was made, the following paragraphs will contain some illustrative images that will help the reader's comprehension. The process will be initially described at a high level and then each component will be described in detail.

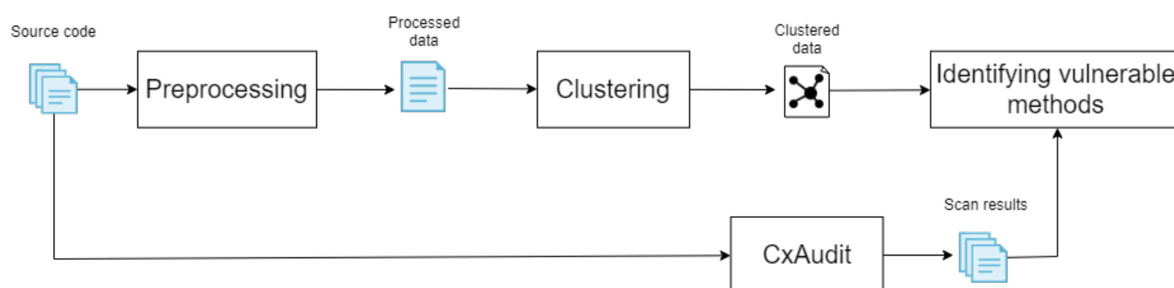


Figure 2: PoC data flow - High Level

COMPONENTS Figure 2 illustrates essentially the conceptual high level steps of the process. The input for this pipeline is a folder with source code files, which are preprocessed and then the processed data is used for clustering. The last stage takes the clustered data, as well as the results from a CxSAST scan, and inspects it to try to find a way of recognizing

vulnerable methods.

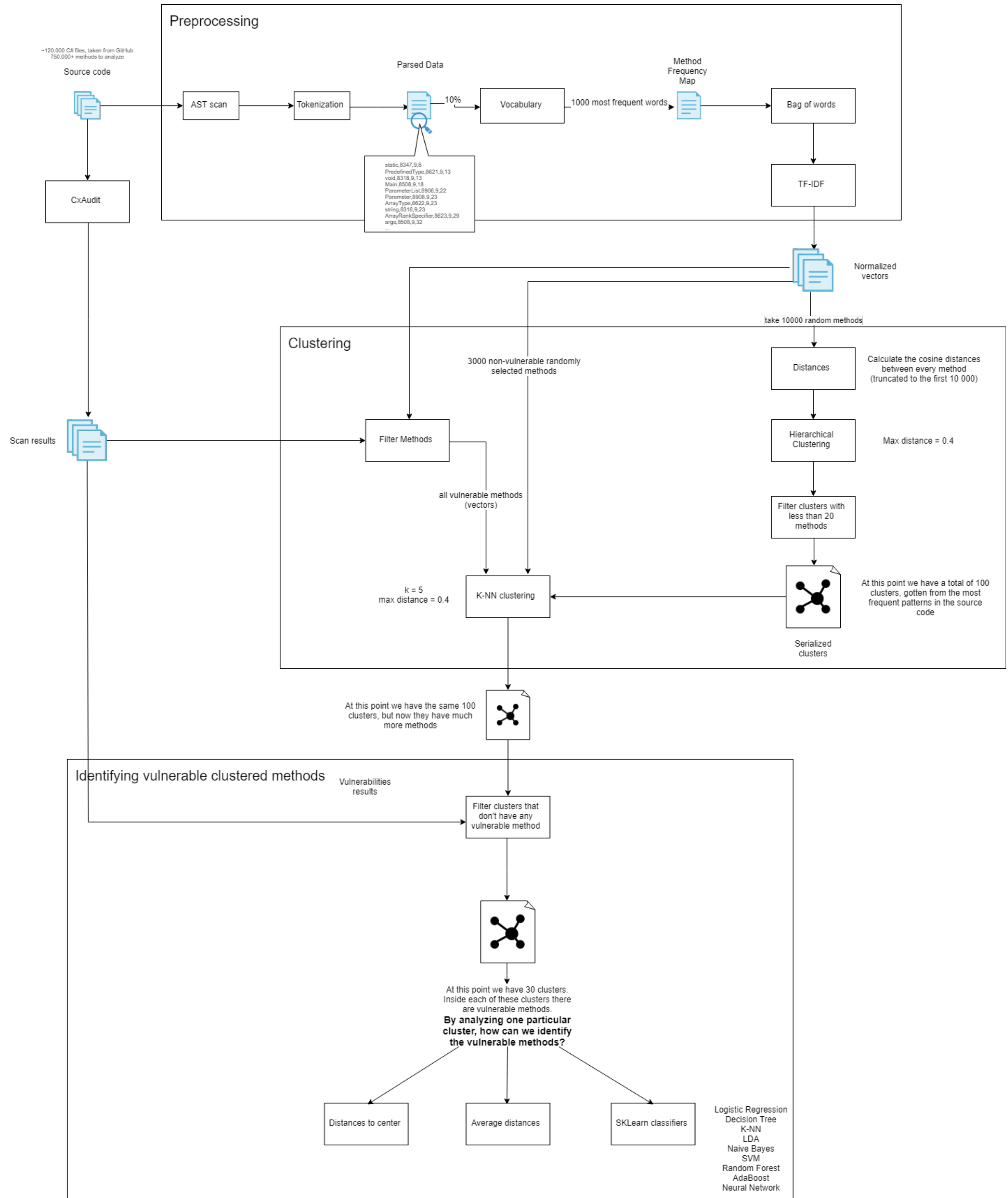


Figure 3: PoC data flow - Overview

OVERVIEW Figure 3 represents the data flow from the source code files to the identification of vulnerable methods. It has some annotations indicating particular aspects of the case study. This diagram will be dismembered and explained in the following paragraphs.

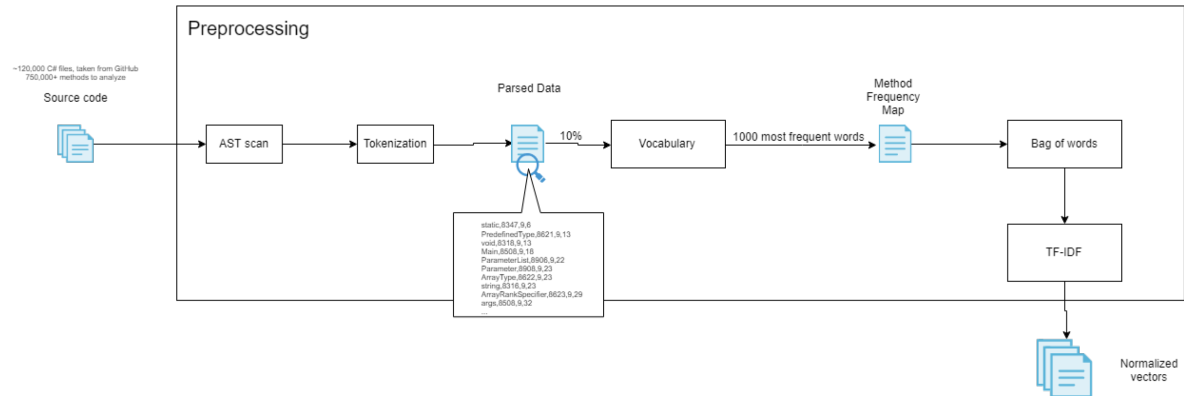


Figure 4: Genesis project data flow - 'Preprocessing' stage

PREPROCESSING Figure 4 represents the preprocessing stage. In a few words, this stage is responsible for processing a set of code files and transforming it into data that can be clustered. This clustering is intended to be made at a method level, i.e. we want to group methods that are similar to each other in the same cluster, for that we need to represent each method as a vector. The first step on this stage is to perform a AST scan (this step uses code from Checkmarx's solution) and generate the 'parsed data'(generated by the tokenizer). This parsed data is a list of all the tokens found in the scanned project, accompanied by some meta-data. Each line of the parsed data contains the text of the token, the ID of the method where the token is found (each and every method on the scanned files is assigned a unique ID), the line number where the token is found and the column number.
From all the parsed tokens only 10% (picked randomly) are used for the next steps because the authors thought that it was representative enough for this study and it saves processing time. With this subset of tokens, a vocabulary is created. This vocabulary is a dictionary that for each token stores the frequency of that token among all of the tokens ('parsed data'). From the vocabulary, only the 1000 most frequent tokens are used in the next steps, for the same reason as the usage of 10% of the tokens.
The next step is to start building vectors. For each parsed method (as described before, every method has it's own identifier), a 1000 dimension vector is created with the occurrences of each token inside that method. This is a technique commonly used in Natural Language Processing called Bag of Words[9]. The last step of this stage is to use Term Frequency–Inverse Document Frequency[9] to normalize the built vectors. The result of this operation will still be a list of vectors with dimension 1000, but each value of the vector will be a relative frequency (across different files) instead of an absolute frequency. This means that if a given

token has lots of occurrences inside a certain file, but zero occurrences on all other files, will have a lower relative frequency than a token that has fewer occurrences in global but appears in lots of files.
This list of vectors is the final output of the preprocessing stage - it is all that is needed to start performing clustering.

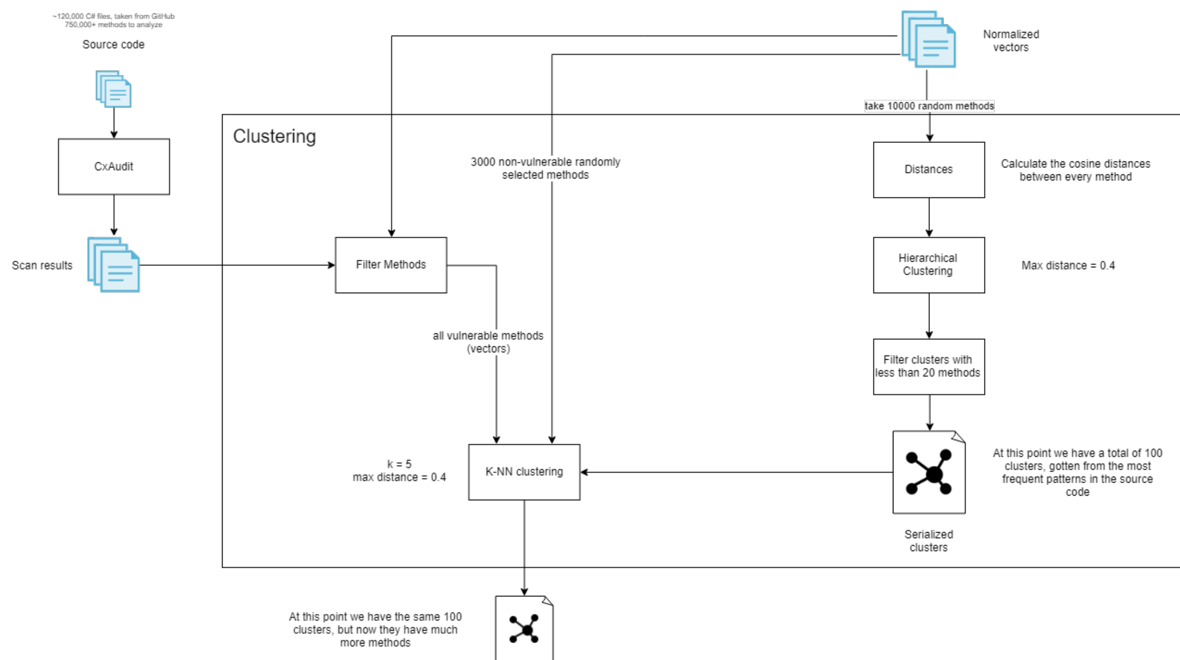


Figure 5: PoC data flow - 'Clustering' stage

CLUSTERING Figure 5 represents the clustering stage. The first step of this stage is calculating the Cosine distances[10] between each pair of methods. This step requires some memory usage because the number of distance calculations for a set of N methods is equal to ${}^{N}C_{2}$. For example, if there are 1000 methods to be clustered, ${}^{1000}C_{2} = 499500$ distances need to be calculated and stored in memory. For that reason, during the experiment, only 10000 randomly selected methods are used in the next steps. Having a set of methods and the distance between each pair of methods, Hierarchical Clustering[11, 12] is applied and then the hierarchy is flattened to single clusters. The collaborators of the research decided to preserve only the clusters that had 20 or more methods, because they thought that these small clusters where not significant (this decision is still to be studied and tested). At this point, in the experiment made, there were 100 clusters. Without adding any new clusters, K-Nearest Neighbors algorithm[1] is used to insert the methods that were discarded on the previous steps into the existing 100 clusters. For this insertion, all the vulnerable methods were picked as well as 3000 non vulnerable methods. If a given method is not close enough

1 Have a look at https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm

to a cluster it will be discarded from the next stages. This strategy of clustering a set of the data and then appending the remaining data to the existing clusters is way of having better performance and being able to perform with low computing resources, but might compromise the obtained results. This is an aspect to be studied further on this thesis.
The output of the clustering stage is a set of 100 clusters, each one having vulnerable and non vulnerable methods, to be analyzed on the next stage.

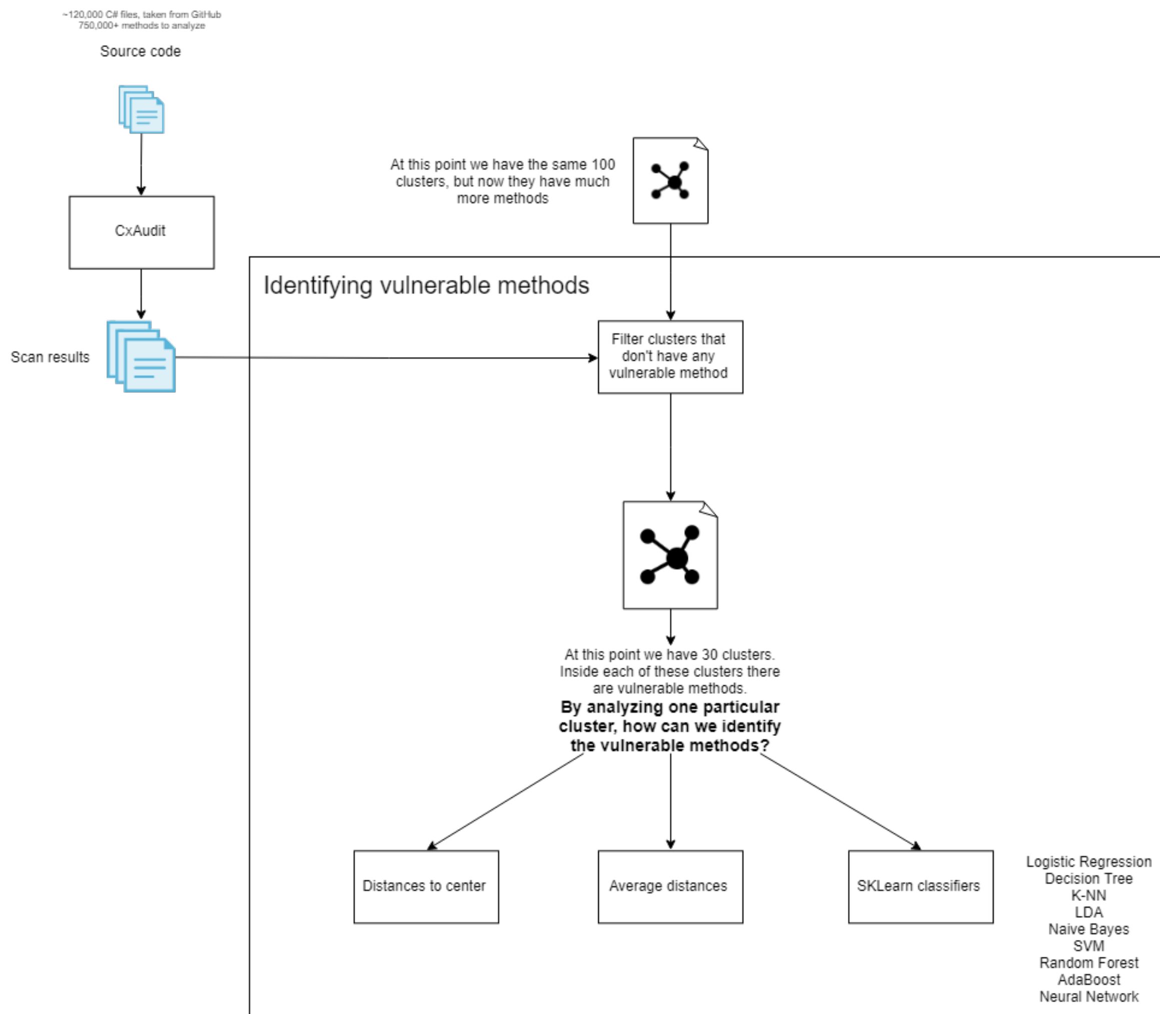


Figure 6: PoC data flow - 'Identifying vulnerable methods' stage

IDENTIFYING VULNERABLE METHODS Figure 6 represents the stage where the clustered data is analyzed. The goal of the experiment at this stage is to find out if, by analyzing only the clustered data, there is a way of differentiating the vulnerable methods from the non-vulnerable(safe). The first step was to discard from the clustered data all the clusters that did not contain any vulnerable methods (based on the results from the CxSAST scan). After this filtering, 70 clusters were discarded and 30 clusters were left for analysis. At

this point there are 30 clusters, each one containing safe and vulnerable methods. Three approaches were tested to try to find a way of differentiating the vulnerable methods from the safe ones.

Distances to center For each vulnerable method in each cluster, the distance to the center of the cluster is calculated. Then, the same average is calculated for the safe methods. The results were not very clear, there wasn't a clear difference between vulnerable and safe methods.

Average of distances For each vulnerable method in each cluster, the average of the distances between that method and every other method in the cluster was calculated:

$$\frac{\sum_i^M \sum_j^V distance(i,j)}{count(M)/count(V)},$$

being M a set of all methods in a cluster, V a set of vulnerable methods in a cluster, $distance(x,y)$ a function that calculates the distance between 2 methods and *count* a function that returns the number of elements in a set of methods. Similarly, the same average was calculated for the safe methods

$$\frac{\sum_i^M \sum_j^S distance(i,j)}{count(M)/count(S)},$$

where S is a set of all the safe methods in a cluster.
The results of these calculations were compared and in average, the vulnerable methods were closer to the other methods than the non-vulnerable.

Supervised learning With the SKLearn[2] classifiers, although it looked like there were very good results (above 95% Accuracy) and very good recall, after analyzing the results, it was concluded that in most cases there was a single vulnerable method (or a very small number of such) in each cluster. This means that if one would classify all methods in a cluster as "non vulnerable", the one method that is vulnerable would be wrongly classified, but it would have a small impact on the accuracy calculation (mathematically).

RESULTS The final conclusion of the experiment is that there are some indicators that it might be possible to identify vulnerable methods inside clusters, but due to the small amount of vulnerable methods on the used data set, the results aren't reliable and much more data (and more balanced data) is needed to get some reliability on the results.

2 See https://scikit-learn.org

2.2 RELATED WORK

There is a wide variety of approaches in the vulnerability detection field, as well as in program analysis. The most common approaches for vulnerability detection are Static Analysis and Dynamic Analysis. Both of them are highly dependant on human input and have scalability issues. Many commercial tools like Checkmarx[3], Fortify[4] and Coverity [5], and open-source tools like FlawFinder[6], RATS[7] and ITS4[13] are based on these techniques and therefore require human experts to write rules capable of identifying vulnerabilities.

Another well studied approach is finding vulnerabilities based on code similarity, or code clone detection. This approach can be explained in three steps. The first step is to split a program into code fragments [14, 15, 16, 17, 18]. The second step is to represent each code fragment in an abstract representation, including tokens [17, 14, 18], trees [19, 15], and graphs [16, 15]. The third step is to compute the similarity between code fragments via their abstract representations obtained in the second step. This approach has proven to produce good results but has some limitations. Some code clones can not be identified because of statements rearrangement and/or insertions/deletions, which might introduce the need of human validation. Also, it is not capable of detecting vulnerabilities that are not caused by code clones.

There are some projects that study the use of machine learning to find software vulnerabilities. To use machine learning for vulnerability detection, source code needs to be represented as vectors [1]. There are two approaches for this task. One is to convert tokens extracted from programs, such as variable names, function names, data types and keywords, to vectors [20]; the other one is to convert nodes of Abstract Syntax Trees from programs, such as identifier declarations, function definitions, function invocations and control flow nodes, to vectors [21, 22]. These techniques were the base for some projects [23, 24], [25], that showed that machine learning techniques can outperform other approaches in terms of performance and accuracy in some cases.

Also related with this project, it is worthy of mention the use of deep learning for software defect prediction [22, 26]. However, software defects are not the same as vulnerabilities (i.e., techniques for defect detection cannot be used for vulnerability detection) [27], and the file-level representation of programs in [22] isn't fine-grained enough to accurately point the locations of vulnerabilities. Also, the defect prediction technique developed in [26] deals with code changes rather than source code programs as a whole. Somewhat related work is the use of deep learning for purposes like API learning [28], code cloning detection [29],

3 https://www.checkmarx.com, accessed on May 2020
4 https://www.microfocus.com/en-us/solutions/application-security/, accessed on May 2020
5 https://scan.coverity.com/, accessed on May 2020
6 http://www.dwheeler.com/flawfinder, accessed on May 2020
7 https://code.google.com/archive/p/rough-auditing-tool-for-security/, accessed on May 2020

software language modeling [20], and malicious file paths, URLs and registry keys detection [30].

There is a particular project that was definitely helpful for this work. Inspired on Seq2Seq [31] models, used for natural language processing, namely for text translation, Code2Seq [2] uses a structured representation of code (AST) to generate sequences of text strings. Its developers have proven its great capabilities by publishing a model capable of predicting meaningful names for code snippets, as well as another model capable of predicting meaningful code summarization. Beyond the great results, Code2Seq seems to be the first and only mechanism that can be easily used for different programming languages, without needing any changes in the main logic of its flow.

3

PROPOSAL

After explaining the context, motivation, objectives and state of the art on the subject that is to be researched, a proposal will be described in this chapter. This chapter will contain a brief description of the proposed approaches to achieve the objectives, as well as the technologies that will be useful for its accomplishment.

Since there isn't a single approach proposal, each of the architectures will be described with detail on the "Development" chapter.

3.1 PROPOSED APPROACHES

Taking into account the lessons learned while studying the related work and the genesis project, there are two main conceptual approaches that seem to be research worthy.

1. **Continuing the genesis project** - the developed pipeline seems promising, but it lacks quality and context awareness during the creation of vectors. The main focus in this approach should be to find a way of generating vectors that contain semantic information about the methods. Assuming that good quality vectors can be generated, experimentation should be made with the proposed techniques in the genesis project: clustering and classifiers.

2. **Training a model for vulnerability prediction** - take benefit of the infrastructure of projects that allow to train models for prediction at a method-level [1], [2] and train a model capable of predicting, with acceptable accuracy, if a given method is vulnerable or not.

3.2 TECHNOLOGIES

As this is a machine learning project, the used programming language will be python. Python is highly popular among data scientist for its simplicity, versatility and for all the high quality data-related libraries available. In the following subsections the proposed technologies are briefly described, as well as their value to the proposed approaches.

3.2.1 *code2vec*

Inspired in word2vec [32] models, commonly used in natural language processing, code2vec is a neural model for representing snippets of code as continuous distributed vectors (”code embeddings”). The main idea is to represent a code snippet as a single fixed-length *code vector*, which can be used to predict semantic properties of the snippet. To this end, code is first decomposed to a collection of paths in its AST. Then, the network learns the atomic representation of each path while simultaneously learning how to aggregate a set of them. Unlike other approaches for embedding code, code2vec does not treat code as a sequence of tokens. It uses ASTs to leverage a structured representation of code and therefore take benefit of it to obtain a more accurate representation of code as vectors. It’s authors demonstrated it’s efficiency with the challenging problem of predicting and generating meaningful names for methods, accomplishing “*an improvement of more than 75%, making it the first to successfully predict method names based on a large, cross-project corpus*”[1]. Given a code snippet, it’s AST is generated and every path between every pair of terminal nodes is stored. Each of these paths if called a *path-context*. Given a bag of *path-contexts*, attention models are used to determine which *path-contexts* better represent the snippet. The 4 most relevant paths in each snippet are then used to represent the code snippet itself.

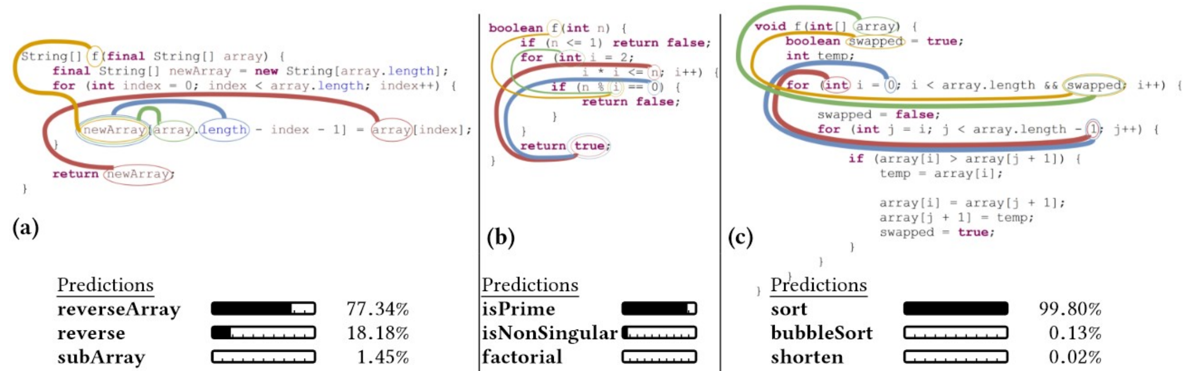


Figure 7: code2vec - Example predictions, from [1]

Figure 7 shows example predictions from code2vec model, with the top-4 paths that were given the most attention for each code snippet. The width of each path is proportional to the attention it was given by the model.

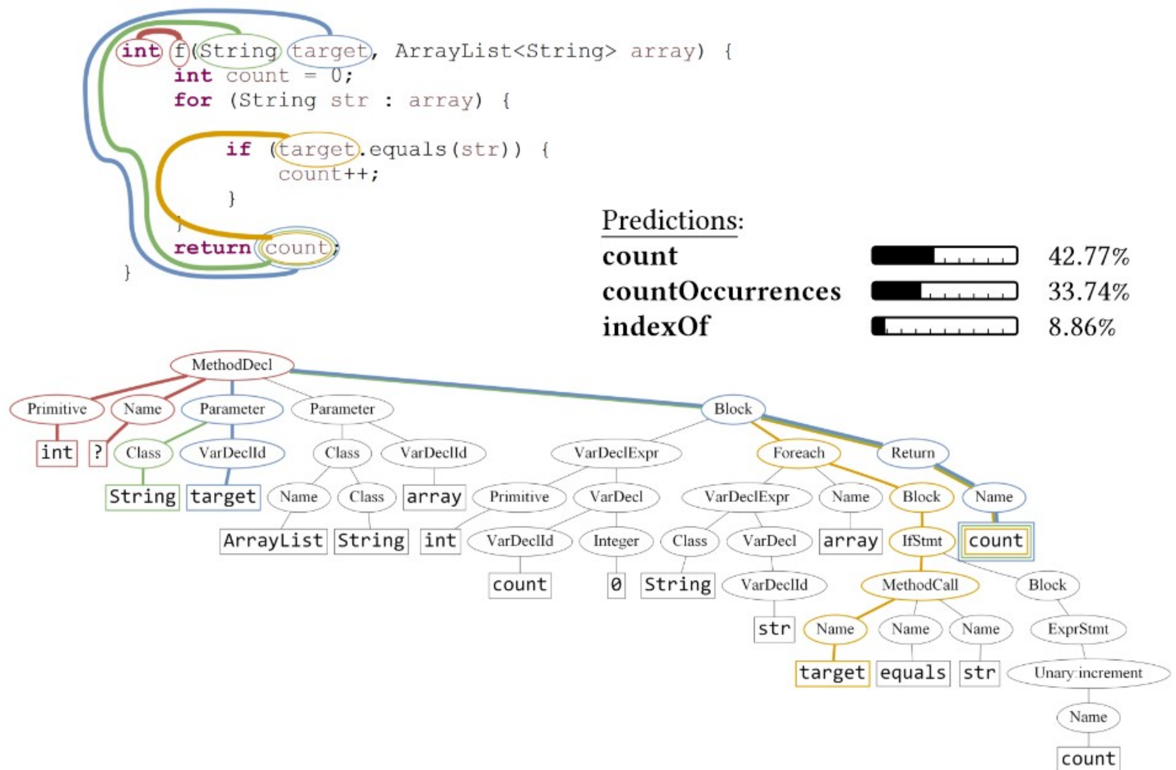


Figure 8: code2vec - AST path-context example, from [1]

Figure 8 represents an example for a method name prediction, portrayed on the AST. The top-4 path-contexts were given a similar attention, which is higher than the rest of the path-contexts.

The main contribution from code2vec to this project is the ability to represent code snippets as vectors, in a way that captures the code's semantic.

3.2.2 *code2seq*

Inspired in seq2seq[31] technique, commonly used in translation problems (e.g. Google's translation service uses seq2seq), code2seq is an approach to generate natural language sequences from source code snippets that leverages the syntactic structure of programming languages to better encode source code. This model represents a code snippet as the set of compositional paths in its AST and uses attention models [1] to select the relevant paths while decoding.

Similarly to code2vec, code2seq leverages code structured representation (AST) to create *path-contexts*. Instead of generating vectors with it, Encoder-Decoder Model [33] is used to

1 See https://deepai.org/machine-learning-glossary-and-terms/attention-models

generate sequences of words.

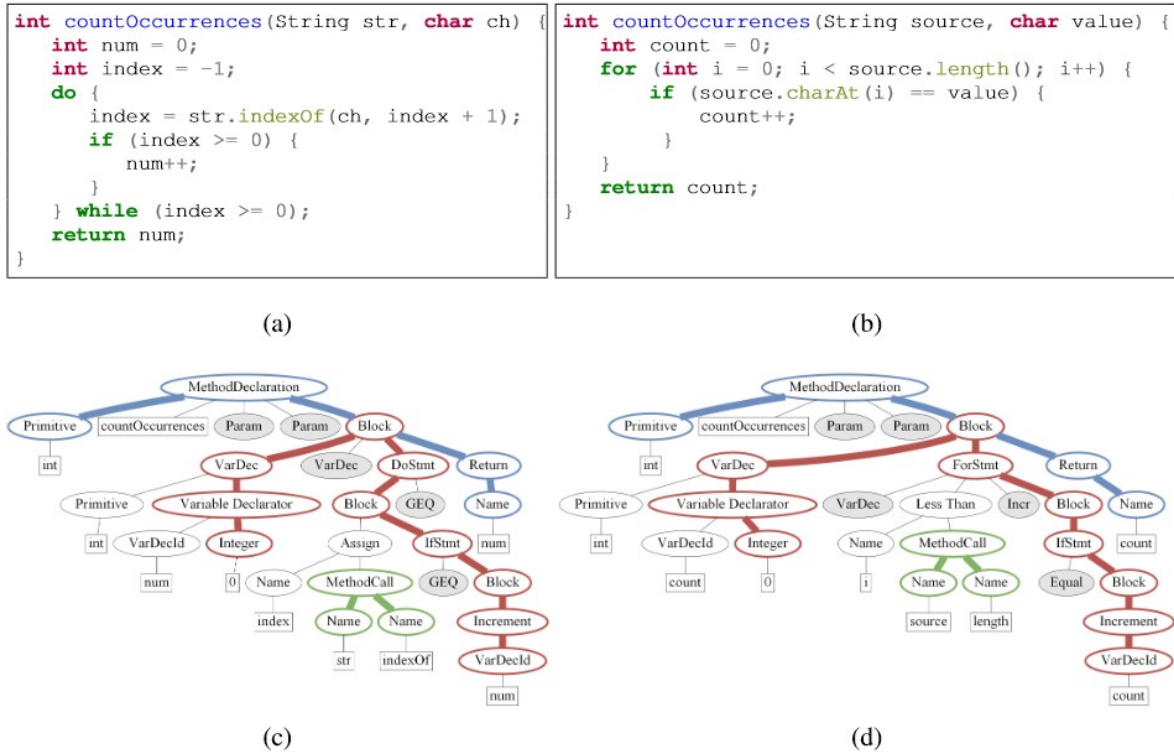


Figure 9: code2seq - path-contexts example, from [2]

Figure 9 represents an example of two Java methods that have exactly the same functionality. Although these methods have different sequential(token-based) representations, repeating paths, which might differ in only a single node (a ForStmtnode instead of a Do-while node), will be revealed if syntactic patterns are considered. The main contribution of code2seq for this project is the ability to train a model that predicts if a given code snippet is vulnerable or not.

3.2.3 *Scikit-Learn*

Scikit-Learn is an open-source Python library, used by a vast portion of data scientists around the world. It provides tools for several machine learning related problems, such as regression, preprocessing and plotting. The tools used for this thesis are related to classification and clustering.

4

DEVELOPMENT

The approach that led to this thesis, described in the ”Genesis Project” section (2.1), has one very obvious downside: the way vectors are generated lacks context information. Two tokens with the same written form can't be differentiated from each other. For example, the word `int` can appear on a parameter declaration (in the parameters of a method), but it can also appear in a variable declaration (in a method's body).

To address this issue, the first approach was to explore how to use word2vec to develop a better mechanism to generate vectors from methods. Before any work was made on this technology, research led to the discovery of code2vec, which seemed to be exactly what was needed from the beginning.

Conceptually, the same way word2vec allows to embed a word into a vector, code2vec allows to embed a method into a vector. There is, though, a difference between these embeddings: code2vec can be trained for different purposes. With code2vec, a model can be trained to generate a vector that represent a method's name, but it can also be trained to represent a method's complexity, for example. Taking that into account, this chapter will describe the different approaches that were explored during this work. The first section will describe the data used for all the experiments, and the following sections will report the processes of the approaches themselves, explaining all the decisions made, the challenges found, and some of the results obtained.

4.1 DATA SOURCES

The data used for all the following approaches was provided by Checkmarx. It consists of 2 distinct data types: a set of open-source Java projects, some of them developed specifically to be used for cybersecurity purposes and awareness; and a set of XML files that contain results for vulnerabilities. There are as many XML files as Java projects, i.e. each XML file contains the vulnerabilities of one Java project. The XML files were results obtained from a CxSAST scan, having all the results validated by humans. False positive results were discarded for all purposes in this thesis work. This dataset has a total of 43 Java projects, with a total of 752942 methods and 381010 vulnerabilities results.

Along the following sections there will be references to datasets called *m*, *l*, *xl* and *All*. These datasets are the result of the aggregating all methods of every project and then splitting them sets of different sizes. All the sets are disjoint, except the *All* dataset, which contains every method in all projects.

The following table shows the number of methods in each of these datasets.

Dataset	Number of methods
m	131696
l	182989
xl	300744
All	752942

4.2 APPROACH 1 - CODE2VEC EMBEDDING

The first experiment made with code2vec was to train a model that would predict if a method is vulnerable or not. This was made by labeling each method in the training dataset with a boolean value (vulnerable or not-vulnerable). This approach implied some modifications in code2vec's source code, in order to make this labeling in a customized way, by merging each method in the Java projects with the vulnerabilities results whose path nodes are contained in it.

The results obtained were not very good. Although during the training phase the reported accuracy was around 91%, some small code samples (these samples contained vulnerabilities that would be trivially detected by a SAST solution) were tested manually and the output was wrong around 90% of the times. This approach was soonly interrupted, not only because of the bad results but also because research led to a even better technology for this purpose: code2seq[2].

There was one other approach where code2vec was used and it will be described before describing the experiment made with code2seq.

Later on this document (4.3.3) there is a possible explanation to why this approach produced these results.

4.3 APPROACH 2 - USING CODE VECTORS FOR CLUSTERING AND CLASSIFIERS

The authors of code2vec have a model publicly available, that was trained to embed methods names. This approach relied on this model, as it was trained with approximately 12 million methods and was used on the code2vec demonstration project that had really good results. This approach essentially followed the pipeline of the genesis project, but all the code was redesigned to a object-oriented paradigm and the method embedding task was

delegated to code2vec.

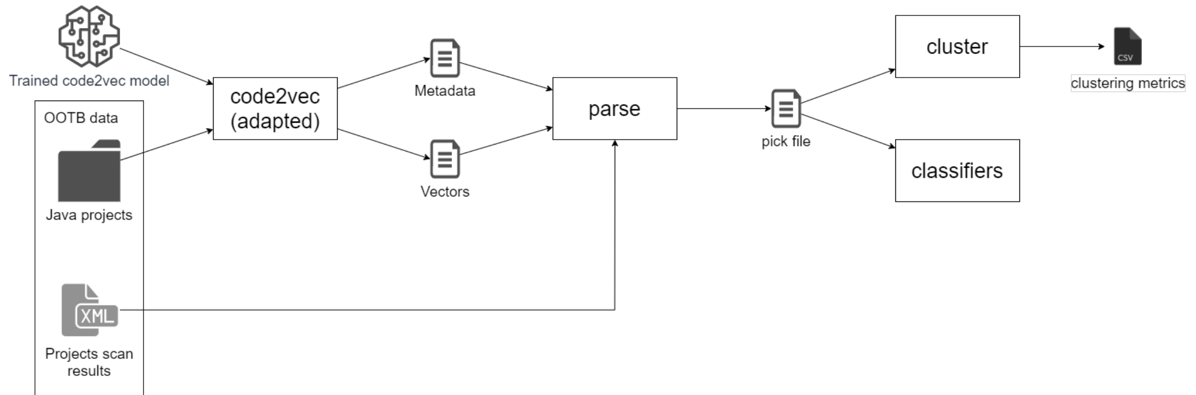


Figure 10: Approach 2 dataflow - overview

Figure 10 is a representation of the data flow for this approach that helps understanding what will be described in the following paragraphs.

4.3.1 *Generating vectors*

The first step was to modify code2vec source code to output metadata that is needed for the next steps. code2vec default output is a file where each method of the input is described with 2 components: the method label and the corresponding vector. This data is insufficient to be merged with the vulnerabilities results - the goal would be to have each method linked to it's vulnerabilities. To accomplish that, for each method, some metadata is needed, namely the file where it is present and its position in the file (start line, start column, end line and end column). The strategy was to assign a unique ID to each method during the preprocessing stage of code2vec. This ID was concatenated to every method name in the output file. Along with this output, the source code was modified to output a second file with the needed metadata.

4.3.2 *Parsing*

This stage has the task of parsing the files generated by code2vec, as well as the XML files with vulnerabilities reports, merging them and creating a data structure where every method's data is stored as conveniently as possible to allow different experiments with it. Initially, the methods are parsed from the code2vec output files, allowing the data structure to contain, for each method: it's ID, name, vector, file name, and line span (to be interpreted as a method's position in a file : line and column where the method starts and the line

and column where the method ends). Separately, the XML files are parsed into a list of vulnerabilities. Each vulnerability is described with the following properties:

- **Name** - Vulnerability name (SQL Injection, Remote Code Execution, etc...);
- **Query path** - a Checkmarx internal path to identify a query;
- **Severity** - the severity of the vulnerability, which can be High, Medium or Low;
- **Path Nodes** - a Checkmarx vulnerability result (flow), as described on 1.1, where each path node contains the file name, line number and column number.

After all methods and all vulnerabilities are parsed, all vulnerabilities are traversed to be merged with the methods. After this merge process it is possible to determine, for each method, the list of vulnerabilities that are contained in it.

METHOD CLASSIFICATION CRITERIA The concept of "vulnerable method" is not trivial nor well defined, i.e. given a vulnerability flow, a method can be classified as vulnerable (or not) according to different criteria:

- **All nodes contained in method** - A method is classified as vulnerable if all the nodes in the flow are contained in the method. This is a very strict (and probably the most accurate) classification, but represents a scenario that is not likely to happen, since vulnerabilities flows tend to reach several methods in a call stack;
- **Input and Sink nodes in method** - A method is classified as vulnerable when the Input Node and the Sink Node are contained in the method. This means that the nodes in between these two might be contained in other methods, that were called by the method were the Input and the Sink are contained. This scenario is more likely to happen than the previous, but it also introduces some uncertainty - it is possible that the most critical vulnerable segments of code are contained on the called methods instead of the method itself. Note that this approach will also classify a method as vulnerable if all the nodes of the flow are contained in the same method;
- **Input node in method** - A method is classified as vulnerable when there is a flow whose Input Node is contained in it. Semantically, this means that the method can be considered as a vulnerable entry-point. Although in some cases this actually represents a vulnerability in a method, it is a very heuristic approach and causes lots of false positives.
- **Sink node in method** - A method is classified as vulnerable when there is a flow whose Sink Node is contained in it. Similarly to the previous approach, although it can identify real vulnerable methods, would produce many false positives.

There is a particular case that would be correctly identified in any of the 4 cases described above: when a vulnerability result has only 1 node.

During this approach, only the "All nodes contained in method" criterion was used, as it is the most reliable criterion. "Input and Sink nodes in method" criterion was used later in the third approach (4.4).

After the tasks of parsing and merging are completed, the built data structure can be represented by the following class diagram.

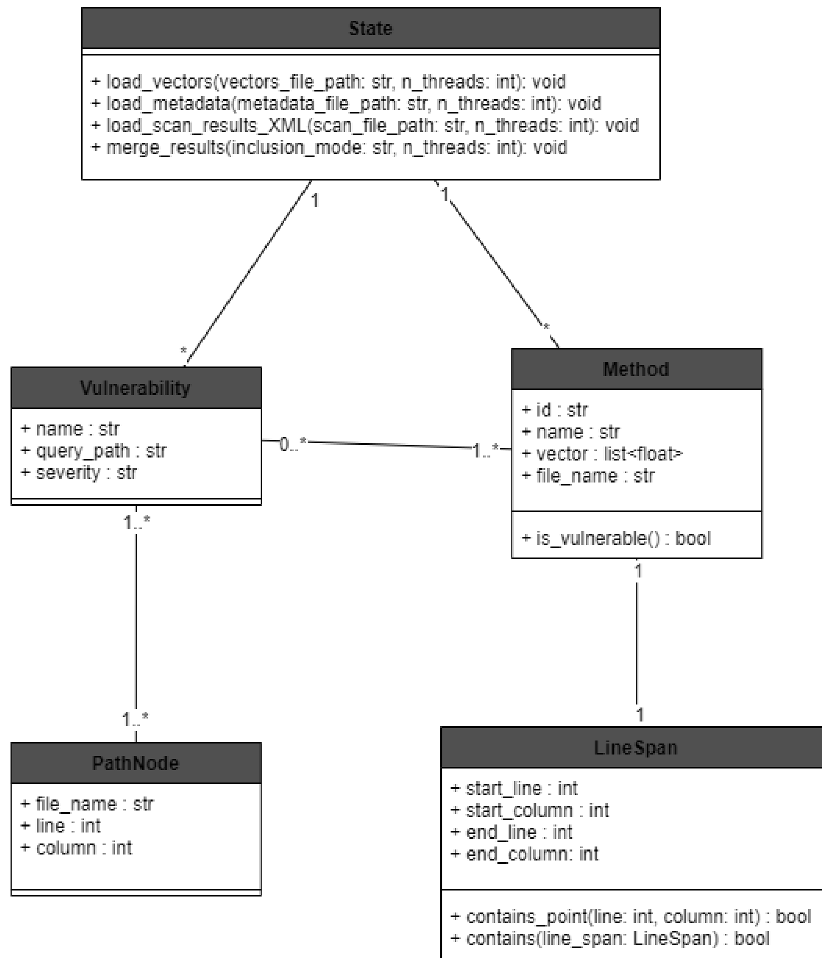


Figure 11: Approach 2 data structure - class diagram

Figure 11 displays the class diagram of the designed structure for experimentation. With this generic and versatile data structure, it is easy to traverse the methods list and, for each of the methods, output its identifier, code vector and vulnerabilities (if there are any).

4.3.3 *Experimentation*

After the parsing stage, the loaded data is saved to a file. Methods and vulnerabilities are now stored in a convenient structure, to be consumed by two different experiments that will be described in the following sections.

BALANCING DATA Due to lack of experience in machine learning and working with large datasets, only after some attempts to have good results, it was concluded that the dataset needed to be balanced. The conclusion was that around 95% of the methods in the dataset were considered not vulnerable, according to the "All nodes contained in method" criterion. This means that if a model is trained with such dataset, it will most likely predict every input method as not vulnerable. This aspect was probably what made the first approach have such bad results. The reason why that approach was not resumed was because, at the time this issue was noticed, there was already work made on approach 3, which is based on code2seq. According to the developers of code2vec and code2seq, code2seq is more reliable for the type of training that is intended to perform in approaches 1 and 3. Taking that into account, the conclusion was that approach 1 could have produced good results, but was abandoned and he effort was dedicated to approach 3. Something that is worth mentioning also is that this matter caused some setback on both approaches (2 and 3), which means that some weeks of training and result analysis were thrown away.

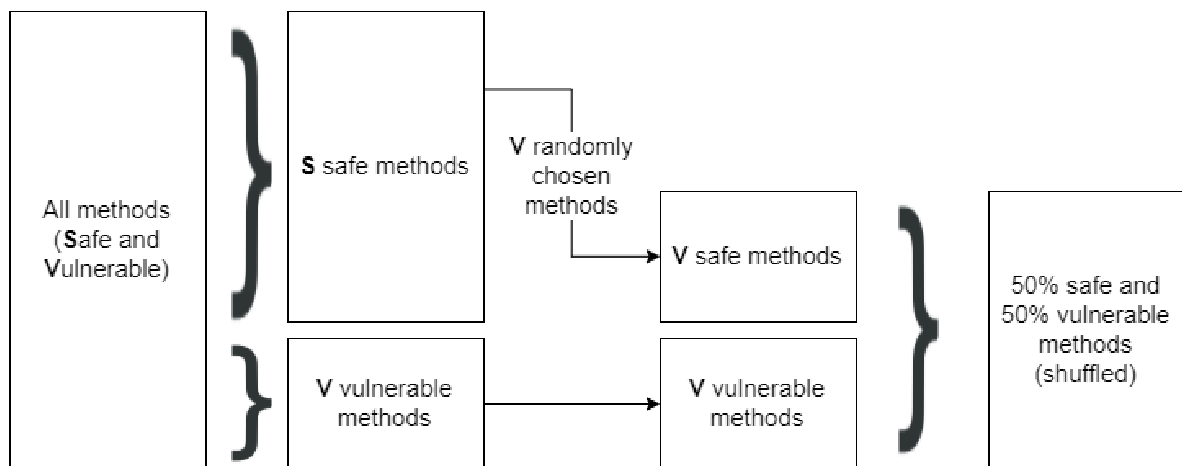


Figure 12: Approach 3 data balancing

Figure 12 represents visually the process of balancing the available data. The output of this process is a set of methods, being half of them vulnerable and the other half not vulnerable. This process causes a significant data loss because most of the safe methods end up to not be used. For that reason, this process was repeated several times, in order to

generate multiple datasets where the vulnerable methods are always the same, but the safe methods are chosen randomly.

Clustering

The first experiment made with the described data structure was to use the code vectors to create clusters and see if there was a clear distinction between vulnerable and safe methods. The ideal result of this experimentation would be to obtain a set of clusters only with vulnerable methods and a set of clusters only with safe methods. If this could be accomplished, in an ideal scenario, it would be pretty straight-forward to detect if a method was vulnerable or not by calculating the cluster where it would fit better. In order to try to accomplish this, two types of clustering were experimented: hierarchical clustering and K-Means clustering.

HIERARCHICAL CLUSTERING The goal with hierarchical clustering was to let SciPy's function fclusterdata create a hierarchy of clusters based on the distances(euclidean) between the methods and try to do it in a way that there was an evident distinction between the vulnerable and the safe clusters. To perform hierarchical clustering, one parameter has to be provided: the maximum distance between methods of the same cluster. The algorithm was executed several times, with several datasets and with different values for the maximum distance parameter. This technique was clearly not a fit for the desired results for 2 reasons:

- **Scalability** - As far as research could lead this investigation, there isn't any hierarchical clustering algorithm that can perform better than O(N^2). The machine used for the experiment has 32GB of RAM and it was only enough to process approximately half of the available data.
- **Results** - Several experiments were made with sub-sets of the available data but the results never got even near to the expected, i.e. there wasn't a clear distinction between vulnerable clusters and safe clusters.

K-MEANS CLUSTERING Just as with hierarchical clustering, the goal with this algorithm was to create a set of clusters where there was a clear distinction between vulnerable and safe methods, based on the distances between the vectors. Differently from hierarchical clustering, K-Means tries to fit the vectors to a predefined number of clusters, *nClusters*. Given that, the main goal was to find the optimal *nClusters* so that the formed clusters are as close as possible to the ideal distinction between vulnerable and safe methods.

In order to solve the problem of finding the best number of clusters, the decision was to express this as a maximization problem. When K-Means algorithm is applied to the code

vectors, the output contains clusters of several types. To start modelling the problem, these types of clusters were classified in the following categories:

1. **Clusters with size 1** - Clusters with only 1 method, being that method either safe or vulnerable. Clusters of this category should be minimized because they represent outliers, and therefore contribute negatively to the desired output;
2. **Mixed clusters** - Clusters with vulnerable and safe methods. These are also to be minimized, since the desired output would have a clear distinction between safe and vulnerable methods;
3. **Good clusters** - Clusters with more than 1 method, that contain only safe methods, or only vulnerable methods. This kind of clusters should be maximized;

Having these categories defined, a score function was defined in order to classify the quality of the outputs of the K-Means algorithm. Being x a set of clusters (an output from K-Means algorithm):

$$score1(x) = (1 - length1Freq(x)) \times (1 - mixedFreq(x)) \times goodFreq(x)$$

where

$$length1Freq(x) = clustersWithSize1(x)/totalClusters(x),$$
$$mixedFreq(x) = mixedClusters(x)/totalClusters(x),$$
$$goodFreq(x) = (allVulnerableClusters(x) + allSafeClusters(x))/totalClusters(x)$$

This score function was calculated for several values of *nClusters* and with different subsets of the available data. The results were quite inconclusive. Because of that, some modifications were made to the score function, with the objective of getting more significant conclusions. Three more score functions were defined:

$$score2(x) = (1 - 2 \times length1Freq(x)) \times (1 - 2 \times mixedFreq(x)) \times goodFreq(x)$$

,

$$score1StdDev(x) = score1(x) \times (1 - normalizedStdDev(x))$$

and

$$score2StdDev(x) = score2(x) \times (1 - normalizedStdDev(x))$$

where

$$normalizedStdDev(x) = clusterSizeStdDev(x)/biggestClusterSize(x)$$

In the functions $score1StdDev$ and $score2StdDev$, the standard deviation of the clusters size is intended to me minimized, so that sets of clusters with approximate sizes get better score than sets of clusters with high variation of cluster size.

The K-Means algorithm was applied to four different datasets (m, l, xl and All), with $nClusters \in [250, 500] : 10 \; divides \; nClusters$. This $nClusters$ range was chosen after some experimentation in the range between 2 and 1000. The intention of using different datasets was to try to identify common peaks in the score functions.

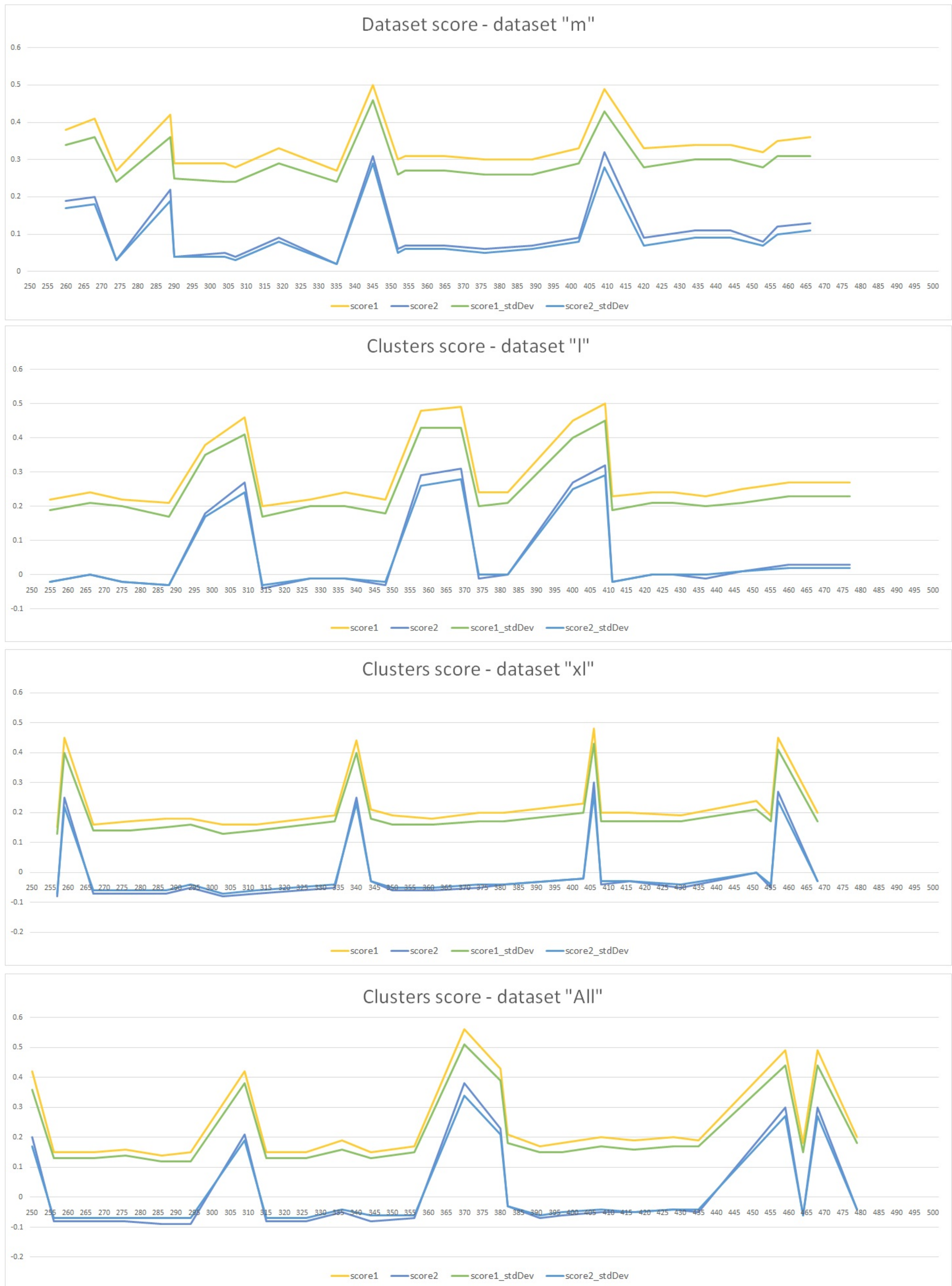


Figure 13: Clustering score functions in different datasets

Figure 13 shows the charts for the four score functions for each of the four datasets. The dots on the chart don't align perfectly because K-Means tries to fit the input to a given number of clusters, but there is no guaranty that the output will contain that exact number of clusters. The first conclusion that was taken from its observation was that the four score functions have very similar curves, and therefore are redundant. Regarding the matter of identifying common peaks across the different datasets, it is possible to conclude that there is some similarity between some of the curves. To better analyze these similarities, since it was concluded that the score functions are redundant, the results of the function *score*1 were merged in one chart, in figure 14.

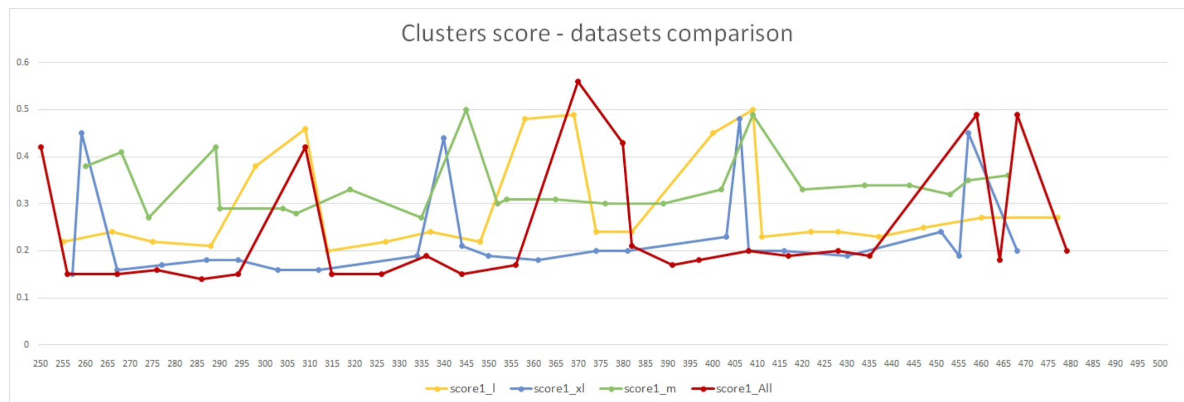


Figure 14: Clustering score function comparison across different datasets

By observation of figure 14 the most obvious common peaks are around the *nClusters* values 310, 370, 410 and 460. These are the four candidates to be the optimal value for *nClusters*. By analyzing these regions in every and each of the datasets, the conclusion was that, although they present the best scores, the number of 100% vulnerable clusters was always zero. This means that, in this experiment's conditions, the goal of achieving clear distinction between vulnerable and safe clusters is far from feasible. That's the reason why this technique was considered as inappropriate and it's study interrupted.

Classification

The second experiment made with the defined data structure was to use machine learning classifiers. The goal with this experiment would be to train classifications models capable of predicting if a given method is vulnerable or not.

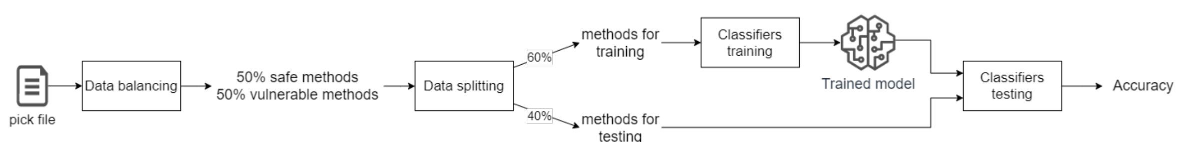


Figure 15: Training classifiers for detecting vulnerabilities - data flow overview

Figure 15 represents the data flow of this experiment. The input is a serialized file, with data stored as described in figure 11. Having a set of methods and vulnerabilities in that file, it is read and then the data is balanced as described in 4.3.3. After the data is balanced, 60% of the methods is randomly chosen to be used for the training phase, leaving the other 40% for accuracy testing. The training data is then used to train 12 models, using 12 different classification algorithms (note: all the classification algorithms used were imported from python Scikit-Learn library). All of the classification algorithms were used with the default values for its parameters. There wasn't any experimentation regarding the variation of these values, but there is interest in trying it in future work.

Each of the trained models is then tested against the testing portion of the data and the results are stored.

Dataset name	m
Total methods	131696
Vulnerable methods	6402
Used methods	12804

Classifier	Score	Train Time (m)	Test Time (m)
Gaussian Process	0.8729	24.83	0.24
Neural Net	0.8586	0.12	0
Linear SVM	0.8106	0.25	0.09
Logistic Regression (liblinear)	0.8071	0.02	0
Logistic Regression (lbfgs)	0.8071	0.01	0
Nearest Neighbors	0.8057	0	0.46
QDA	0.8005	0.01	0
AdaBoost	0.7784	0.028	0.01
Decision Tree	0.7399	0.02	0
Random Forest	0.7255	0	0
Naive Bayes	0.7216	0	0
RBF SVM	0.5738	0.5	0.25

Figure 16: Classifiers results when trained with dataset *m*

Dataset name	l
Total methods	182989
Vulnerable methods	8036
Used methods	16072

	Classifier	Score	Train Time (m)	Test Time (m)
	Gaussian Process	0.8385	47.99	0.39
	Neural Net	0.8297	0.3	0
▲	Logistic Regression (liblinear)	0.7836	0.02	0
▲	Logistic Regression (lbfgs)	0.7836	0.01	0
▼	Linear SVM	0.7796	0.4	0.15
▲	QDA	0.7662	0.01	0
▲	AdaBoost	0.7606	0.33	0.01
▼	Nearest Neighbors	0.7535	0	0.7
▲	Naive Bayes	0.7203	0	0
▼	Decision Tree	0.7163	0.03	0
▼	Random Forest	0.7098	0	0
	RBF SVM	0.5323	1.24	0.37

Figure 17: Classifiers results when trained with dataset *l*

Dataset name	xl
Total methods	300744
Vulnerable methods	14354
Used methods	28708

	Classifier	Score	Train Time (m)	Test Time (m)
	Gaussian Process	0.8533	206.57	1.09
	Neural Net	0.8396	0.29	0
▲	Logistic Regression (lbfgs)	0.7944	0.01	0
▼	Logistic Regression (liblinear)	0.7943	0.03	0
	Linear SVM	0.7925	1.74	0.46
▲	Nearest Neighbors	0.7815	0.01	2.05
	AdaBoost	0.7645	0.62	0.01
▼	QDA	0.7612	0.02	0
	Naive Bayes	0.7215	0	0
	Decision Tree	0.7144	0.05	0
	Random Forest	0.7105	0	0
	RBF SVM	0.5516	6.06	1.13

Figure 18: Classifiers results when trained with dataset *xl*

Dataset name	All
Total methods	752942
Vulnerable methods	27188
Used methods	54376

	Classifier	Score	Train Time (m)	Test Time (m)
	Gaussian Process	Not enough memory		
	Neural Net	0.8444	0.62	0
▲	Nearest Neighbors	0.7969	0.02	8.28
▲	Linear SVM	0.7916	11.43	1.87
▼	Logistic Regression (liblinear)	0.7893	0.05	0
▼	Logistic Regression (lbfgs)	0.7893	0.02	0
	AdaBoost	0.7613	1.28	0.02
	QDA	0.7465	0.03	0.01
	Naive Bayes	0.7184	0	0
	Decision Tree	0.7146	0.11	0
	Random Forest	0.7034	0	0
	RBF SVM	0.5585	24.98	4.32

Figure 19: Classifiers results when trained with dataset *All*

Figures 16, 17, 18 and 19 show the accuracy of each of the trained models, as well as the elapsed times in the training and testing phases. As the data balancing process introduces some data loss and randomization, these results were obtained from calculating the average of 5 different executions. The standard deviation between the executions was very small, meaning that the scores and times were almost the same even with different datasets (of the same size). In figures 17, 18 and 19, the green and red arrows on the left side of the tables can help in understanding how the different algorithms behave when inputs of different sizes are used as input. The precision of the time measurements is 2 decimal places, meaning that zero values represent less than 0.01 minutes, or 0.6 seconds. The reason why train time and test time were measured separately is the interest in understanding what would be the best algorithm for specific use cases, e.g. for a real-time vulnerability detection application, the test time should be as low as possible.

By analysing the score column, it is quite obvious what are the 2 best performant algorithms in terms of accuracy, regardless of the size of the dataset - Gaussian Process and Neural Net. Although the Gaussian Process algorithm has a better accuracy for every dataset, the test time is not as good as desired, when compared to other algorithms. Also, this algorithm is quite resource consuming and the *All* dataset could not be used for training in a machine with 32GB of RAM (19). The Neural Net algorithm seems to be the best choice for most use cases because of it's accuracy and test time. Regarding the train time: this data

is almost merely informative, since the train time should not be a concern in most use cases. It would only be important to take this metric into account if the model would be used in an application with continuous learning. The main goal of this work does not fit into that category, so for that matter the train time can be seen as irrelevant. In 6 there are some suggestions of applications where this metric would be relevant.

To summarize, the conclusion is that Gaussian Process algorithm would be the best fit for use cases where the accuracy is absolutely crucial, but taking into account that the main goal of this work is to improve performance, Neural Net would be the best choice. Some more results analysis of this approach will be available in 5.

4.4 APPROACH 3 - CODE2SEQ EMBEDDING

From the same developers as code2vec, there is a project that performs better on encoding methods - code2seq. This third approach is conceptually the same as the first one, but this time code2seq was used instead of code2vec. Just as in the first approach, the goal was to train a model capable of predicting if a method is vulnerable or not.

Trying accomplish this, code2seq preprocessing stage was modified in a way that each method from the input data is labeled with a boolean, indicating if the method is vulnerable or not.

After preprocessing all the methods from the dataset, a model was trained and the accuracy was around 95%. Unfortunately these results weren't actually reliable. After analysing the data from the used dataset, it was possible to conclude that only around 5% of the methods in it were vulnerable. This means that if the model classified every single method as safe, the accuracy would be 95%. It was understood that the data had to be balanced to produce more reliable results. The decision was to create a dataset with 50% safe methods and 50% vulnerable methods. This process was described previously in 4.3.3.

In this approach, the classification method used was the "Input and Sink nodes in method" (4.3.2). Some attempts were made with "All nodes contained in method" criterion, but it was before the data balancing issue was noticed. That experiment was abandoned, but it would be interesting to resume it in future work.

USING CXSAST PREPROCESSED PROJECTS TO TRAIN MODELS As described in 1.1, CxSAST performs preprocessing to some files, in order to obtain more and better results. The fact is that some of the results available on the dataset can only be identified with that preprocessing, since some files in the projects are not actually written in Java. This preprocessing stage transforms those files into a Java representation and that is what makes it possible to identify vulnerabilities on such files. Taking that into account, it would make

sense to train the models with those preprocessed files instead of training with the pure source code projects.

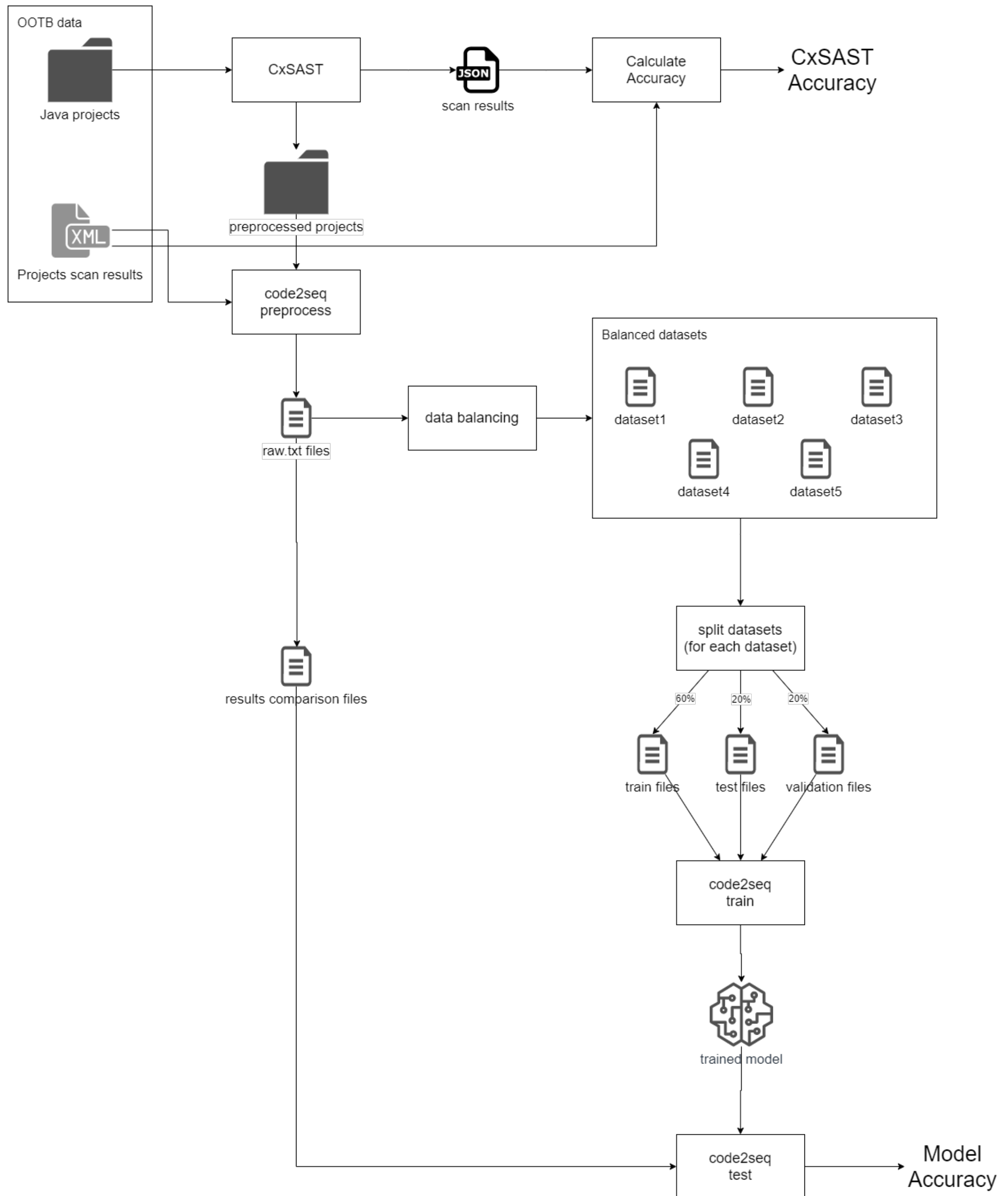


Figure 20: Using CxSAST preprocessed projects to train models - from preprocessing to result analysis

Figure 20 shows the process from the original Java projects to the accuracy testing. The output of the ”code2seq preprocess” stage is a file for each project (43 files) from which 3 projects were separated from the rest, not being used for the training stage, to be used for result comparison. The data balancing takes as input a set of files (40 files in this case), concatenates them all and them splits it into 2 files: one file with only vulnerable methods and the other with only safe methods. Having those 2 files, the data was balanced as described in 4.3.3, and 5 different datasets were generated, where all of them contained the same vulnerable methods, but randomly chosen safe methods. The decision of generating 5 datasets for training was due to the fact that there would be a major waste of available data if only 1 dataset was trained (taking into account that the data balancing truncates significantly the amount of used data). Each of the 5 datasets were used to train a model, meaning that this process resulted in 5 different trained models. These models' accuracy was then tested with the 3 projects that were separated from the rest. These 3 projects are not balanced, to simulate a real vulnerability analysis scenario. The results of this experiment will be detailed in 5.

USING ORIGINAL PROJECTS TO TRAIN MODELS The results obtained from the process described above were not as good as expected, so the process was repeated all over again, but this time the source code analysed by code2seq was not preprocessed by CxSAST.

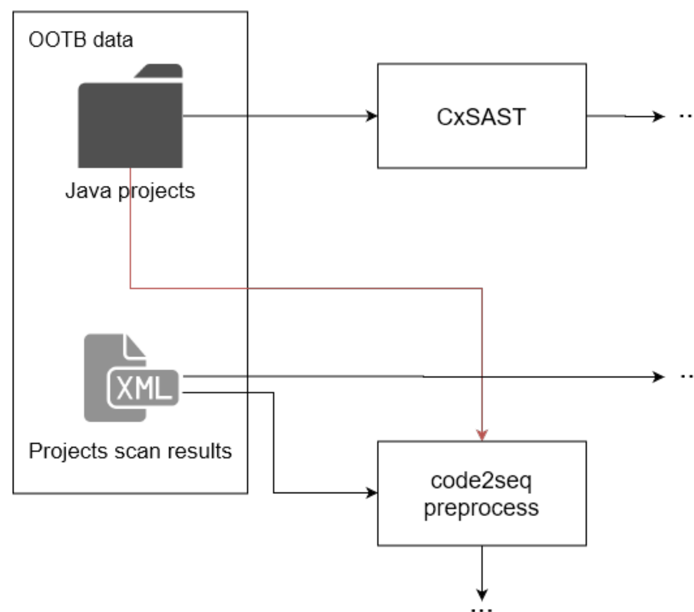


Figure 21: Using original projects projects to train models - difference from last process

Figure 21 shows the portion of the process represented in figure 20 that changed from one experiment to another. The results of this experiment were better than the previous and will be reported in 5.

SPLITTING VULNERABILITIES RESULTS Until this point, models were trained to predict any type of vulnerability. Training models for specific groups of vulnerabilities would be an interesting approach for investigation (proposed as future work) and, in case of success, would provide the ability to identify what type of vulnerability is found in a method. There are some vulnerabilities that can be detected with very similar patterns, e.g. when SQL Injection and Code Injection vulnerabilities are detected by a CxSAST scan their flows are likely to contain similarities. The results from the XML files were splitted into categories to allow this type of training, but no model was actually trained for this purpose, because of the setbacks related to data balancing and other challenges. At first sight this experiment seems like it could achieve very good results, and for that reason it will be proposed as future work.

4.5 SUPPORTING DIFFERENT LANGUAGES

All the based on Java projects. Contrarily to other related projects, that doesn't mean that the contribute of this work can't be easily extended to other programming languages. In fact, the documentation pages of code2vec [1] and code2seq [2] provide simple instructions and links to what they call "extractors". Extractor is the component that reads source code of a specific language and converts it to a normalized and language-agnostic format, that will be then consumed to generate vectors/sequences. The community around code2vec and code2seq has made publicly available extractors for the following languages: Java, C# , Python, C, C++ and TypeScript. Regarding the many other languages, that are supported by CxSAST but don't have an extractor publicly available, it would be easy to create extractors for them, since Checkmarx already has grammars capable of extracting the AST of such languages. Having the AST of a source code file, it is quite easy and well documented how to convert it to the format required for generating vectors/sequences with code2vec/code2seq.

It is also important to understand that, since there is a language-agnostic representation of the AST of a code snippet, it would possibly be feasible to train a single model capable of identifying vulnerabilities in projects of different programming languages. This subject arises interesting possibilities that will be suggested as future work 7.2.

1 https://github.com/tech-srl/code2vec#extending-to-other-languages
2 https://github.com/tech-srl/code2seq#extending-to-other-languages

5 RESULTS

To analyze the results of the trained models, in comparison to the CxSAST software, 3 projects were chosen specifically to perform accuracy and performance tests. These are open-source projects, developed to be intentionally vulnerable so they can be used for several purposes, like teaching, raising awareness to web application security and helping to assess accuracy of vulnerability scanners. These projects are called SecurityShepherd [1], WAVSEP [2] and WebGoat [3] and were scanned in 5 different scenarios:

- **CxSAST scan** - A regular CxSAST vulnerability scan, which outputs a JSON file. This file was then parsed and the results were compared to the results in the corresponding XML file to obtain the accuracy;
- **code2seq models trained with CxSAST preprocessed files** - Each of the 3 projects was preprocessed with the adapted version of code2seq, in order to get a labeling for each method (vulnerable or safe). For each of the 5 models trained with the CxSAST preprocessed files (4.4), it's accuracy was tested in predicting the labels for each of the 3 projects;
- **code2seq models trained with original files** - The test was performed in the same circumstances as the previous, but the 5 models were trained with the original source code projects instead of a CxSAST preprocessed version.
- **Classification models - tested with CxSAST preprocessed files** - The models trained in Approach 2 were tested using vectors generated from the CxSAST preprocessed files.
- **Classification models - tested with original files** - The models trained in Approach 2 were tested using vectors from the projects original files.

Note: in figures 22 and 23 the models from Approach 2 are the same, tested with different code vectors (CxSAST preprocessed vs original files). The models from Approach 3 are

1 https://github.com/OWASP/SecurityShepherd
2 https://github.com/sectooladdict/wavsep
3 https://github.com/WebGoat/WebGoat

different, since they were trained with different data, and were also tested with different data.

		Approach 2 - classifiers		Approach 3 - code2seq					
		Logistic Regression	Linear SVM	model1	model2	model3	model4	model5	CxSAST
SecurityShepherd	Accuracy	0.29	0.26	0.43	0.47	0.45	0.45	0.46	0.96
	Time (s)	192	192	78	77	79	76	77	950
WAVSEP	Accuracy	0.52	0.5	0.66	0.55	0.71	0.67	0.71	0.64
	Time (s)	51	51	19	16	17	17	17	8460
WebGoat	Accuracy	0.67	0.65	0.84	0.81	0.82	0.79	0.8	0.78
	Time (s)	108	108	40	39	39	40	39	5823
Average Accuracy		0.493	0.470	0.643	0.610	0.660	0.637	0.657	0.793
		7	8	4	6	2	5	3	1

	Safe methods	Vulnerable methods	Total methods	% vulnerable
SecurityShepherd	1364	0	1364	0
WavSep	58	0	58	0
WebGoat	788	115	903	0.13

Figure 22: Accuracy comparison - models tested with CxSAST preprocessed files

Figure 22 shows the accuracy and scan times for each of the trained models, plus the CxSAST scan. The table on the right side of the figure displays information about the 3 test projects' nature. According to the methods labeling, SecurityShepherd and WAVSEP have 0 vulnerable methods. This information is doubtful and the reason for those numbers is that CxSAST preprocessed files were used for the method labeling. Although, as explained in 4.4, it seems to be a good idea to use the preprocessed files to obtain more information, the preprocessed files have modifications in its content and it caused problems in the matching between vulnerabilities and methods. For example, if a method on the original source code starts on line 3 and ends on line 10, and the CxSAST preprocessing transforms it in a way that it will end on line 8 instead, it will impact the merging process between the vulnerabilities on XML file and the source code.

One curious aspect about these results is that Logistic Regression and Linear SVM are the algorithms from Approach 2 whose models had better accuracy. In section 4.3.3 it was concluded that Gaussian Process and Neural Net were the best candidates, as they performed better in the experiments made then. Further investigation would be needed to explain why these models performed differently on the two experiments.

By analyzing the scores and their averages, it is possible to conclude that the models trained with classifiers performed much worse than the models trained with code2seq. The average accuracy of the code2seq models is not bad but is not good enough to achieve the desired results.

		Approach 2 - classifiers		Approach 3 - code2seq					
		Logistic Regression	Linear SVM	model1	model2	model3	model4	model5	CxSAST
SecurityShepherd	Accuracy	0.62	0.61	0.74	0.77	0.8	0.77	0.75	0.96
	Time (s)	187	187	76	77	76	72	76	950
WAVSEP	Accuracy	0.78	0.78	0.83	0.83	0.83	0.83	0.79	0.64
	Time (s)	50	50	17	17	17	15	19	8460
WebGoat	Accuracy	0.7	0.69	0.82	0.79	0.81	0.82	0.79	0.78
	Time (s)	104	104	41	40	42	40	41	5823
Average Accuracy		0.700	0.693	0.797	0.797	0.813	0.807	0.777	0.793
		7	8	3	4	1	2	6	5

	Safe methods	Vulnerable methods	Total methods	% vulnerable
SecurityShepherd	693	540	1233	0.44
WavSep	51	7	58	0.12
WebGoat	788	115	903	0.13

Figure 23: Accuracy comparison - models tested with original source code files

Similarly to Figure 22, Figure 23 displays the accuracy and time measurements for the trained models, but this time the labeling was performed with the original source code projects. Just like in the previous Figure, it is possible to conclude that the code2seq models perform better than the classification models. In this experiment the obtained average accuracy is very good, even better that CxSAST in some cases. Regarding the scan times, it is very clear than the developed mechanisms outperforms CxSAST in performance by a large difference.

dataset	total time (m)	total time (h)	best epoch	total epochs	time(m)/ epoch
1	1133	18,88	22	26	43,58
2	1890	31,50	39	43	43,95
3	2012	33,53	42	46	43,74
4	2012	33,53	42	46	43,74
5	1133	18,88	22	26	43,58

Figure 24: Training times for the 5 different datasets

Figure 24 report the times elapsed for training the code2seq models evaluated in figure 23. The training was performed with a patience level of 4 epochs, i.e. the training of a model stops when there are 4 consecutive epochs without improvement. That is the reason why the ”total epochs” column values are equal to the ”best epoch” column values plus 4. The chosen number of epochs was 4 because it was possible to conclude that the added time of increasing the patience level wasn’t worth it, in terms of accuracy of the trained models. The training was performed in a machine with a Intel Core i7-7700 @ 3.6GHz CPU and 32GB of RAM. The average time for each training epoch is clearly fixed around 44 minutes, but this could be improved by using a machine with NVIDIA graphics. code2seq uses TensorFlow [4] algorithms which perform much faster if ran on a GPU, by using CUDA technology [5].

4 See https://www.tensorflow.org/
5 See https://developer.nvidia.com/cuda-zone

6

USE CASES

In this chapter, different possible uses of the developed work will be presented. Although this work was initially intended to be integrated in the CxSAST pipeline, there are many different possible applications for it:

- **Reduce CxSAST scan files** - One possible application of a well-trained model would be to integrate it's functionality in the CxSAST pipeline, to discard files that are definitely not vulnerable. The idea would be to integrate it in an early stage of the pipeline, in a way that it could be used as a filter for large projects. Taking into account that some projects from Checkmarx customers have millions of lines of code, it would make a significant impact on performance if it was possible to classify some files, or even entire components/packages, as safe and not scanning them at all. Although this is conceptually a relatively easy integration, and would result in significant performance improvement, some research should be made in order to find out a way to do this without discarding any files that might actually have vulnerabilities.
- **False Positive and False Negative detection** - A well-trained model could be used at the end of the CxSAST pipeline to confirm results from the scans, and therefore add some more credibility to it. For instance, if CxSAST reports a vulnerability in a method, and a trained model predicts the same method as safe, that result could be flagged as doubtful and be highlighted for human analysis. The same would happen the other way, i.e. when CxSAST scan doesn't find a vulnerability that the trained model finds. This application, contrarily to the previous, would increase the scan time instead of decreasing it, but on the other hand would increase the results credibility and possibly alert for issues that were not found by the CxSAST scan.
- **Stand-alone application** - A program could be developed to scan an entire folder and output the predicted vulnerabilities to a JSON file. With such program, there could be two possible ways of consuming it: integrating it in any build pipeline so that a scan is always performed before code goes to production; with a proper User Interface, a stand-alone vulnerability detection Software could be developed. As long as the UI

exhibits a clever way of showing results and performing scans. These applications would benefit both the accuracy and the high performance of the trained model.

- **GitHub plug-in** - Similarly to the previous use case, a GitHub plug-in could be developed in order to provide automatic scans every time code is submitted to a repository or a specific branch. Once again, there could be a UI to make the result analysis more appealing or just a report with the locations of the predicted vulnerabilities.

- **IDE plug-in** - Taking into account the high performance of the developed work, it would be interesting to benefit it for Just-in-Time vulnerability detection. A plug-in could be developed for IDEs like Visual Studio Code[1] or IntelliJ IDEA[2], in order to find vulnerabilities immediately as programmers write code. This would also allow the model to keep improving, if the plug-in users reported FPs and FNs, as well as the code that generated the bad results.

1 See https://code.visualstudio.com/
2 See https://www.jetbrains.com/idea/

7

CONCLUSION AND FUTURE WORK

After the literature survey and the development work and tests have been explained in detail along this Master's dissertation, this chapter is intended to express conclusive thoughts about the project outcomes, in two sections. The first section contains the overall conclusions of the research, the contributions and an analysis of the proposed objectives in comparison to the results. The second section contains proposals for future work that might lead to interesting improvements to the work described in this dissertation.

7.1 CONCLUSION

Taking into account all the experiments made and the results obtained, it is possible to state that there is potential on using machine learning for detecting vulnerabilities in source code. Some projects referenced in 2.2 already stated that, and this work not only reinforces that statement but also introduces new techniques that might contribute to the development of this field. Also, although the feature was not tested, this project is the first to propose an architecture capable of performing in different programming languages, with few effort.

The conclusion that can be taken from training code2vec models to detect vulnerabilities (approach 1) is that, although the experiment made was unsuccessful, it might be possible to have good results with it. The results were bad because of the data balancing issue (4.3.3), that was noticed in a late stage of the development. Its study was interrupted because at the time the issue was found, there already was work made on approach 3, which according to the developers of both technologies code2vec and code2seq, is prone to produce better results. Having that in mind, prioritization had to be made and the choice was to focus on approach 3, leaving approach 1 behind, even though it would possibly produce good results if some more effort was invested in it.

Concerning approach 2 (using code vectors for clustering and classification algorithms), the designed data structure was a great starting point. The way methods and vulnerabilities are structured and persisted allows several experiments and might be used in the future inside Checkmarx for other purposes. From this approach we can conclude that using clustering techniques to detect vulnerabilities is unfeasible, as far that this work could lead

to. On the other hand, the experiment made with classifiers revealed good results and showed that classification algorithms are a path that should be further explored in this field of vulnerability detection.

The third and last approach, where code2seq models were trained to classify methods as vulnerable or safe, has proven that Bidirectional Long Short-Term Memory (BiLSTM) neural networks, which is the technology behind code2seq project, can perform the task of detecting vulnerabilities in source code with good accuracy and great performance.

The main contributions of this dissertation are the architectures used in approaches 2 and 3, as well as the models that were trained during the experiments made.

The dataset used for the experiments during this dissertation can be considered small, since machine learning datasets usually have millions of entries for training models with good accuracy. In this case, big datasets are difficult to obtain because it is required that the vulnerability results are validated by humans, and the data balancing ends up shortening the dataset even more. Checkmarx possesses more analyzed results, but in other programming languages than Java. In the future, if the approach of training a single model from different programming languages is explored, it will be possible to gather datasets of a more desirable size.

It should be noticed that there were plenty of challenges and setbacks during this work. Also, the long training times made it impossible to achieve results early enough so that other experiments could be made. Regardless the challenges and setbacks, the results were very positive and many other experiments were thought of. This means that this work might lead to further investigations in this area and possibly result in great discoveries in the vulnerability detection field.

7.2 FUTURE WORK

During the research and development of this Master's project, some interesting ideas came up, but unfortunately the time span of this project wasn't wide enough, so it wasn't possible to address them. Taking that into account, those ideas were noted and will now be presented in this section as future work proposals.

- **Train specific models for specific vulnerabilities** - The models trained in this work are capable of identifying multiple types of vulnerabilities, but aren't able to specify which vulnerabilities are detected. An approach that would definitely solve this problem and even improve the accuracy would be to train models for specific vulnerabilities. However, this means that, if taken literally, there would be a model for every vulnerability (e.g. SQL Injection, Deserialization of Untrusted Data, Cross-site-scripting, etc.). This means that, to scan a method for all kinds of vulnerabilities, that method would have to be analyzed by every trained model. That approach would probably

produce the highest accuracy that this work can lead to, but on the other hand might be quite time-consuming. Experimentation should be made to validate that assumption. Another variant that would be somewhere in the middle of both extremes (i.e. a trained model for all vulnerabilities and a single model for every single vulnerability) would be to create groups of vulnerabilities whose CxSAST results path nodes contain similarities, and train a model for each of the defined groups. This approach would make it possible to specify a set of possible vulnerabilities when a method is classified as vulnerable, and would possibly keep a good performance.

- **Train a single model for multiple languages** - Taking into account that there is a language-agnostic representation of the AST during the preprocessing stage of code2seq, it would be interesting to evaluate the feasibility of training one single model to identify vulnerabilities in different languages. This approach could even be combined with the previous and, for instance, train a model capable of identifying SQL Injection in different languages.
- **Train a model to identify critical spots** - During the development phase, it was possible to conclude that there are some specific code fragments that are present across multiple vulnerability results, of different types. This is a high indicator that multiple flaws could be mitigated in one single region of code. The proposal here would be to train a model capable of identifying those regions. This approach could also be combined with the previous, if both of them produce good results.
- **Tuning code2seq parameters** - Code2seq has multiple hyper-parameters that impact the training phase. The impact of the variation of such parameters wasn't studied during this work. Better accuracy could possibly be achieved, by automating the process of experimenting different configurations during the training stage.
- **Training a model to identify vulnerabilities names** - The model trained in this work was trained to output a boolean value for every analyzed method (true - vulnerable; false - safe). As mentioned before, this means that it isn't able to specify which vulnerability is present in a vulnerable method. Since code2seq allows the target prediction to be any sequence of text strings, a model could be trained to predict vulnerability names instead of boolean values. It would even be possible to identify different vulnerabilities in the same method.

BIBLIOGRAPHY

[1] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *CoRR*, vol. abs/1803.09473, 2018.

[2] U. Alon, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," *CoRR*, vol. abs/1808.01400, 2018.

[3] B. Chess and J. West, *Secure Programming with Static Analysis*, ch. 2. Addison-Wesley Professional, first ed., 2007.

[4] R. Harper, *Practical Foundations for Programming Languages*, pp. 4–6. Cambridge University Press, 2016.

[5] J. Boulanger, *Static Analysis of Software: The Abstract Interpretation*. Wiley, 2011.

[6] F. E. Allen, "Control flow analysis," *SIGPLAN Not.*, vol. 5, pp. 1–19, July 1970.

[7] M. Wolf, *Computers as Components: Principles of Embedded Computing System Design*. The Morgan Kaufmann Series in Computer Architecture and Design, Elsevier Science, 2016.

[8] Z. Chen and M. Monperrus, "A literature study of embeddings on source code," *CoRR*, vol. abs/1904.03061, 2019.

[9] S. Raschka, *Python Machine Learning*, ch. 8. Packt Publishing, 2015.

[10] J. Han, M. Kamber, and J. Pei, *Data Mining Concepts and Techniques*, ch. 2.4.7. Elsevier, 3 ed., 2012.

[11] C. C. Aggarwal, *Data Mining: The Textbook*, ch. 1.4.2. Springer International Publishing Switzerland, 2015.

[12] C. C. Aggarwal, *Data Mining: The Textbook*, ch. 6.4. Springer International Publishing Switzerland, 2015.

[13] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw, "Its4: A static vulnerability scanner for c and c++ code.," in *ACSAC*, pp. 257–, IEEE Computer Society, 2000.

[14] S. Kim, S. Woo, H. Lee, and H. Oh, "Vuddy: A scalable approach for vulnerable code clone discovery," in *2017 IEEE Symposium on Security and Privacy (SP)*, (Los Alamitos, CA, USA), pp. 595–614, IEEE Computer Society, may 2017.

[15] N. H. Pham, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Detection of recurring software vulnerabilities," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, (New York, NY, USA), p. 447–456, Association for Computing Machinery, 2010.

[16] J. Li and M. Ernst, "Cbcd: Cloned buggy code detector," *Proceedings - International Conference on Software Engineering*, pp. 310–320, Jun 2012.

[17] J. Jang, A. Agrawal, and D. Brumley, "Redebug: Finding unpatched code clones in entire os distributions," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, (USA), p. 48–62, IEEE Computer Society, 2012.

[18] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: Scaling code clone detection to big-code," in *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, (New York, NY, USA), p. 1157–1168, Association for Computing Machinery, 2016.

[19] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, (USA), p. 96–105, IEEE Computer Society, 2007.

[20] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk, "Toward deep learning software repositories," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, p. 334–345, IEEE Press, 2015.

[21] L. Mou, G. Li, Y. Liu, H. Peng, Z. Jin, Y. Xu, and L. Zhang, "Building program vector representations for deep learning," *CoRR*, vol. abs/1409.3358, 2014.

[22] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, (New York, NY, USA), p. 297–308, Association for Computing Machinery, 2016.

[23] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," *Proceedings 2018 Network and Distributed System Security Symposium*, 2018.

[24] R. L. Russell, L. Y. Kim, L. H. Hamilton, T. Lazovich, J. A. Harer, O. Ozdemir, P. M. Ellingwood, and M. W. McConley, "Automated vulnerability detection in source code using deep representation learning," *CoRR*, vol. abs/1807.04320, 2018.

[25] J. A. Harer, L. Y. Kim, R. L. Russell, O. Ozdemir, L. R. Kosta, A. Rangamani, L. H. Hamilton, G. I. Centeno, J. R. Key, P. M. Ellingwood, M. W. McConley, J. M. Opper,

S. P. Chin, and T. Lazovich, "Automated software vulnerability detection with machine learning," *CoRR*, vol. abs/1803.04497, 2018.

[26] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep learning for just-in-time defect prediction," in *Proceedings of the 2015 IEEE International Conference on Software Quality, Reliability and Security*, QRS '15, (USA), p. 17–26, IEEE Computer Society, 2015.

[27] P. Morrison, K. Herzig, B. Murphy, and L. Williams, "Challenges with applying vulnerability prediction models," in *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security*, HotSoS '15, (New York, NY, USA), Association for Computing Machinery, 2015.

[28] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep API learning," *CoRR*, vol. abs/1605.08535, 2016.

[29] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, (New York, NY, USA), p. 87–98, Association for Computing Machinery, 2016.

[30] J. Saxe and K. Berlin, "expose: A character-level convolutional neural network with embeddings for detecting malicious urls, file paths and registry keys," *CoRR*, vol. abs/1702.08568, 2017.

[31] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *CoRR*, vol. abs/1409.3215, 2014.

[32] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings* (Y. Bengio and Y. LeCun, eds.), 2013.

[33] K. Cho, B. van Merrienboer, Çaglar Gülçehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *ArXiv*, vol. abs/1406.1078, 2014.