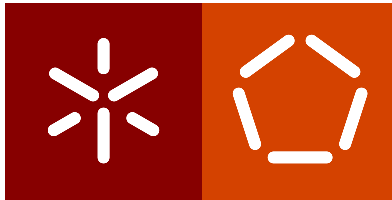
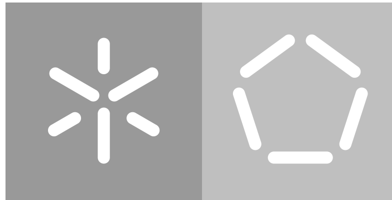


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Tiago João Fernandes Baptista

Fast Scan, an improved approach using machine learning for vulnerability identification

March 2022



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Tiago João Fernandes Baptista

Fast Scan, an improved approach using machine learning for vulnerability identification

Master dissertation
Integrated Master Degree in Informatics Engineering

Dissertation supervised by
Professor Pedro Rangel Henriques
Nuno Oliveira

March 2022

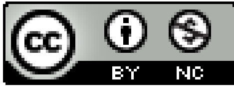
AUTHOR COPYRIGHTS AND TERMS OF USAGE BY THIRD PARTIES

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
CC BY-NC
https://creativecommons.org/licenses/by-nc/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.

Tiago Baptista

ACKNOWLEDGMENTS

In the course of my academic journey many people helped and took a big role in the path that lead me to this final destination, to all of them I express my gratitude.

First of all, I would like to express my acknowledgments to two special people who were important to me throughout this final journey in my final academic year.

To Dr. Pedro Manuel Rangel Santos Henriques for all the availability, enthusiasm and motivation since the day I first approached him. Also for his great vision over each subject that improved each stage of the Master Thesis.

To Nuno Oliveira for all the availability, guidance and close technical supervision of the entire process throughout the year. Also for the opening from his side and Checkmarx to allow access to datasets that were of the uttermost importance to the Master Thesis.

To Geoatributo and specially Dr. Ricardo Almendra which was supportive and flexible in my work schedule, only this way was possible to balance my academic and professional life.

To Regina Sousa for all the love, support and specially her capability to be patience and always be able to improve my mood even in the not so good moments and incentive to never give up on my objectives.

To all my friends, the ones that shared this academic journey with me and also the others, for the support and helping me to disconnect.

Finally, I would like to leave a special thanks to my parents for the love and allowing me to start this journey in the first place, for supporting me even throughout the moments when my academic journey seemed unlikely to reach a safe destination and also pushed me to improve.

ABSTRACT

This document presents a Master Thesis in the Integrated Master's in Informatics Engineering focused on the automatic identification of vulnerabilities, that was accomplished at Universidade do Minho in Braga, Portugal.

This thesis work aims at developing a machine learning based tool for automatic identification of vulnerabilities on programs (source, high level code), that uses an abstract syntax11tree representation. It is based on *FastScan*, using code2seq approach. Fastscan is a recently developed system aimed capable of detecting vulnerabilities in source code using machine learning techniques. Nevertheless, *FastScan* is not able of identifying the vulnerability type. In the presented work the main goal is to go further and develop a method to identify specific types of vulnerabilities. As will be shown, the goal will be achieved by changing the method of receiving and processing in a different way the input data and developing an architecture that brings together multiple models to predict different specific vulnerabilities. The best *f1* metric obtained is 93% resulting in a precision of 90% and accuracy of 85%, according to the performed tests and regarding a trained model to predict vulnerabilities of the injection type. These results were obtained with the contribution given by the optimization of the model's hyperparameters and also the use of the Search Cluster from University of Minho that greatly diminished the necessary time to perform training and testing. It is important to refer that overfitting was detected in the late stages of the tests, so this results do not represent the true value in real context. Also an interface is presented, it allows to better interact with the models and analyse the scan results.

Keywords: vulnerability, attention models, static analysis, security

RESUMO

Este documento apresenta uma dissertação do Mestrado Integrado em Engenharia Informática, que tem como foco a automação da deteção de vulnerabilidades e foi concluída na Universidade do Minho em Braga, Portugal.

O trabalho apresentado nesta tese pretende desenvolver uma ferramenta que utiliza machine learning e que seja capaz de identificar vulnerabilidades em código. Utilizando para isso a representação do mesmo numa abstract syntax tree. Tem como base *FastScan* que utiliza a abordagem do *code2seq*. *Fastscan* é um projeto recentemente desenvolvido que é capaz de detetar vulnerabilidades em código utilizando técnicas de machine learning, sendo que tem algumas lacunas como o facto de não ser capaz de identificar vulnerabilidades específicas. No trabalho apresentado o objetivo é ir mais além e desenvolver um método capaz de identificar qual o tipo específico de vulnerabilidade presente. Como será apresentado ao longo do documento, este objetivo será alcançado pela alteração do método de receção e processamento dos dados recebidos, assim como o desenvolvimento de uma arquitetura que junte os vários modelos de maneira a cooperarem e a ferramenta ser capaz de detetar e prever a presença de vulnerabilidades específicas. A melhor métrica de *f1* obtida foi de 93%, com precisão de 90% e accuracy de 85%, de acordo com os testes efetuados sobre um modelo treinado para prever a presença de vulnerabilidades do tipo de injection. Os resultados foram obtidos devido à otimização dos hiper-parâmetros dos modelos e o cluster Search da Universidade do Minho diminuiu consideravelmente o tempo necessário para efetuar o traino e testes dos modelos. É importante referir que foi detetado overfitting na fase final do desenvolvimento deste trabalho, sendo que os resultados apresentados não representam o valor real dos modelos em contexto real. Para além disso é apresentada uma interface que permite interagir e analisar os resultados de um scan feito pelos modelos.

Palavras-Chave: vulnerabilidade, modelos de atenção, análise estática, segurança

CONTENTS

1 INTRODUCTION 1
- 1.1 Contextualization and Motivation 1
- 1.2 Objectives 2
- 1.3 Research Hypothesis 3
- 1.4 Research Methodologies 3
  - 1.4.1 Methodology Approach 3
- 1.5 Frameworks, languages and libraries 4
- 1.6 Document Structure 5

2 BACKGROUND 7
- 2.1 Machine Learning and vulnerability identification 7
- 2.2 FastScan and code2seq 8
- 2.3 Machine learning performance metrics 9

3 STATE OF THE ART 11
- 3.1 Vulnerabilities 11
  - 3.1.1 Injection 13
  - 3.1.2 Broken Authentication 13
  - 3.1.3 *Cross-Site Scripting* (*XSS*) 15
- 3.2 Vulnerabilities identification 16
  - 3.2.1 Static analysis for security testing 16
  - 3.2.2 Dynamic analysis for security testing 17
  - 3.2.3 Interactive analysis for security testing 18
- 3.3 FastScan 19
- 3.4 Other approaches 23

4 PROPOSED APPROACH 24
- 4.1 System Architecture 24

5 DEVELOPMENT 27
- 5.1 Datasets 27
- 5.2 Hardware and technical details 28
- 5.3 First Phase 29
  - 5.3.1 Data filtering 29
  - 5.3.2 Preprocessing 29
  - 5.3.3 Hyperparameter optimization 30
  - 5.3.4 Training 31

5.4 Second Phase 32
6 results 34
6.1 First Phase 34
6.2 Second Phase 37
6.3 Webgoat Test 39
7 user interface 41
7.1 Architecture 41
7.1.1 Back-end 41
7.1.2 Front-end 43
7.1.3 Installation and general overview 45
7.2 Functionalities 46
7.2.1 Load and upload JAVA project files 47
7.2.2 Scan results 49
8 conclusion 51
8.1 Discussion 51
8.2 Possible Applications 52
8.3 Future Work 53

a representing code using vectors 58

LIST OF FIGURES

Figure 1 Schema about *code2seq*. 9
Figure 2 Schema explaining injection. 13
Figure 3 Schema explaining broken authentication. 14
Figure 4 Schema explaining *code2vec* approach. 20
Figure 5 Schema explaining *code2seq* approach. 22
Figure 6 Schema explaining the first phase. 25
Figure 7 Schema explaining the second phase. 26
Figure 8 CPU details. 28
Figure 9 Memory details. 28
Figure 10 Schema explaining second phase. 33
Figure 11 Injection model evaluation. 35
Figure 12 *XSS* model evaluation. 37
Figure 13 Home page design. 43
Figure 14 Preprocessing consult design. 43
Figure 15 Results list design. 44
Figure 16 Code result visualization design. 44
Figure 17 Schema explaining the interface web application. 46
Figure 18 Final Interface homepage. 47
Figure 19 Final Interface load project. 48
Figure 20 Final Interface scanning project. 48
Figure 21 Final Interface results table . 49
Figure 22 Final Interface code view. 50

ACRONYMS

A

AI Artificial Intelligence.

API Application Programming Interface.

AST Abstract Syntax Tree.

D

DAST Dynamic Application Security Testing.

DL Deep Learning.

DOM Document Object Model.

DSR Design science research.

H

HTTP Hypertext Transfer Protocol.

I

IAST Interactive Application Security Testing.

IDE Integrated Development Environment.

IS Information Systems.

J

JSON JavaScript Object Notation.

L

LDAP Lightweight Directory Access Protocol.

M

ML Machine Learning.

R

RNN Recurrent Neural Network.

S

SAST Static Application Security Testing.

SDLC Software Development Life Cycle.

SQL Structured Query Language.

SVM Suport Vector Machine.

U

UI User Interface.

X

XML Extensible Markup Language.

XSS Cross-Site Scripting.

XXE XML External Entities.

1

INTRODUCTION

This first chapter introduces the project, along with the motivations, objectives, methodology, research hypothesis and document structure. It is relevant to refer that this Master's Thesis was only possible due to the collaboration with the company Checkmarx, the supervisors and specially Samuel Ferreira a collaborator at Checkmarx [1].

1.1 CONTEXTUALIZATION AND MOTIVATION

Nowadays, information systems are a part of almost every aspect in life and furthermore, almost every company is dependent on the liability, safety and security of a software piece. So it is essential to have the capability to identify and correct pieces of code that contain known vulnerabilities in order to, at least, prevent the software from being compromised.

Cybernetic attacks are a constant and present a real threat to companies and people in general, since nowadays almost every device is used has an Internet connection. As a consequence, it is exposed to external threats that try to exploit vulnerabilities.

A vulnerability is a flaw or weakness in a system design or implementation (the way the algorithms are coded in the chosen programming languages) that could be exploited to violate the system security policy (Shirey, 2007b). There are many type of vulnerabilities and many approaches to try to detect such flaws and perform security tests. All the known approaches present pros and cons but none of them stands as a perfect solution. Static analysis is one of the approaches and it can be defined as the analysis of a software without its execution.*Static Application Security Testing* (*SAST*) is an application security testing methodology that allows detecting vulnerabilities at the early stages of software development. It is implemented by many companies, such as Checkmarx. These methodologies have many strengths such as the ability to find vulnerabilities without the need to compile or run code, offering support for different programming languages and being able to easily identify common vulnerabilities and errors like Structured Query Language (SQL) injections and buffer overflows. Despite this, there are still problems with the referred approach, mainly

1 https://www.checkmarx.com

in the production of a great number of false positives, in the lack of identification of the vulnerability type and even performance issues.

There are many tools that implement the concept of static analysis and try to apply it to vulnerabilities detection. Some tools rely on only lexical analysis like *FlawFinder*[2] (Mahmood and Mahmoud, 2018) but have the tendency to output many false positives because they don't take into account the code's semantic (Chess and McGraw, 2004). Other tools like *CxSAST* from Checkmarx overcome this lack by using the *AST* of the program being evaluated. In this context, another challenge is to create and apply the same tool across different languages, one that can clearly identify vulnerabilities with high accuracy and have good performance with big inputs.

To overcome the flaws identified in the *SAST* approach, decreasing the number of false positives and the processing time, a new approach came to the researchers mind: to integrate machine learning techniques.

The idea can be realized by altering and tuning open source projects,namely *code2vec* and *code2seq* in order to try to identify accurately and especially more efficiently than other tools like *CxSAST* (Ferreira, 2019). *code2vec* main idea is to represent a code snippet as a single fixed-length code vector, in order to predict semantic properties of the snippet (Alon et al., 2019) on the other hand *code2seq* represents a code snippet as a set of compositional paths and uses attention to select the relevant paths while decoding (Alon et al., 2018). The resultant approach named *FastScan* was not one hundred percent successful but opened the path to further investigation.

It is clear that a good analysis tool can help spot and eradicate vulnerabilities, furthermore, it is becoming a part of the development process. But, there is still room for improvement and all the research work done in this area can be of uttermost relevance for the industry.

1.2 OBJECTIVES

With all of the previous in consideration the main objectives expected from the presented Master's thesis are the following :

- To tune and improve the *FastScan* approach:
  - Find the best hyper parameters for each case study;
  - Use these hyper parameters to improve evaluation metrics for the models (precision, recall and f1);
- The development of a specific model for each type of vulnerability (mainly injection and also *Cross-Site Scripting* (*XSS*)), that is capable of identifying if a code snippet has a vulnerability or not (Boolean model) of a given type;

2 https://dwheeler.com/flawfinder/

- The design of a proper architecture to develop a general model capable of identifying the occurrence of vulnerabilities and their type, given a code snippet;

Since it has different objectives and it is an evolution of *FastScan*, from now on, when referring the Master's Thesis work, it will be called **new FastScan**.

1.3 RESEARCH HYPOTHESIS

With this work, it is intended to prove that it is possible to detect and identify vulnerabilities in source code of different languages, using machine learning techniques. It is an intention to demonstrate the high accuracy and performance when compared to more traditional *SAST* tools.

1.4 RESEARCH METHODOLOGIES

A research project needs an idea in order to start, this idea will be on the base for the formulation of the research question and will determine the choice of the research methodology (Burns and Groove, 2014).

A good literature search and review are important tasks in the process of research and can be decisive in the accomplishment of the research hypothesis and in the quality and usefulness of the final research result (Peters et al., 2012). It is obvious that if an author is not updated in all the breakthroughs in the researched area, it's work could be redundant or even fall short when compared to other approaches.

In the area of *Information Systems* (*IS*) the *Design science research* (*DSR*) is considered to be a good approach. *DSR* is a rigorous cyclical process methodology in scientific research normally applied to engineering and associated with the development of solutions regarding information technologies.(Hevner, 2007)

1.4.1 *Methodology Approach*

To accomplish this Master Thesis objectives, it is used this iterative methodology based on literature revision, solution proposal,implementation and testing.

To carry out this approach, the working plan is composed of the following six steps that follow *DSR* phases (Peffers et al., 2007) :

- Identify the problem and the motivation to solve it:
  - Bibliographic study to deeply understand the state of the art in the areas of static analysis, current available *SAST* tools and vulnerabilities identification process;

- Bibliographic study to understand *AI* and *ML* concepts, namely how they are applied in the study area.

- Define objectives for the solution, referred previously.
- Design and Development :
  - Investigation and development of the boolean models capable to identify specific vulnerabilities;
  - Investigation and development of the generic model, capable of identifying if there is a vulnerability and its type.
- Demonstration :
  - Use the boolean models capable to identify specific vulnerabilities in a test dataset;
  - Use the generic model in a test dataset;
- Evaluation :
  - Results assessment and discussion in order to draw considerations and conclusions about the approach performance and usability in real cases;
- Communication ;

The Master Thesis report was written in parallel with all the steps described previously, since it started from the first step and it is improved with the iterations over the phases.

1.5 FRAMEWORKS, LANGUAGES AND LIBRARIES

In order to achieve the proposed objective different technologies were used, namely :

- Python : It is a programming language on which there can be applied different programming paradigms like object oriented, functional or even scripting. In this case it was used the object oriented approach using classes and other features. It was chosen because the base project for the developed work - code2seq uses it and it has much support and libraries to work with *ML*. More specifically, it was used in the models development and also to achieve the objective of optimize the training hyperparameters.
- Tensorflow : It is an open source framework for *ML*. It is used to train and improve models as well as verify predictions. "TensorFlow is an open source library for numerical computation and large-scale machine learning" [3]. It was used with python in the model train and test.

3 https://www.infoworld.com/article/3278008/what-is-tensorflow-the-machine-learning-library-explained.html

- Wandb : It is a python library, used to monitor, compare, optimize and visualize *ML* experiments. It can be integrated with many frameworks including tensorflow which was a great feature in the development of this Master's thesis. This library was of great importance in the optimization of the models hyperparameters for each dataset and also to visualize performance and time metrics from the experiments [4].
- Java : It is a object-oriented programming language that allows programs to run in any Java virtual machine regardless of the machine where it is running. It was used because the *FastScan* preprocessing was written in Java and there was the need to make modifications.
- Django : It is a *python* web framework that allows to develop full stack projects. It was chosen as the backend tool in order to more easily integrate the rest of the project that was mainly written in *python* and also because it is reliable, fast and scalable.
- ReactJS : It is a *JavaScript* framework designed to build web user interfaces. It was chosen because of previous knowledge and experience using it and also because it is widely use, proven and has good performance;
- code2seq: As previously mentioned, it was base for the machine learning component of the project.

1.6 DOCUMENT STRUCTURE

Chapter 1 is intended to present a brief description of the following chapters, in order to let the reader have some knowledge of the motivation behind the thesis and also it's objectives.

Chapter 2 presents important concepts and general information essential for a better understanding of the followings.

Chapter 3 starts with a detailed explanation of the context on the study area and also there is a section dedicated to related work, an important step to decide which direction the proposed work should go, by understanding the already existing approaches, techniques and results.

Chapter 4, focuses on explaining the defined strategy, used technologies and system architecture to prove the thesis hypothesis.

Chapter 5 details all the details about the used datasets, technical details as well as specifies how the proposed approach presented previously in Chapter 4 was accomplished.

Chapter 6 presents the results obtained namely details about the models performance.

Chapter 7 presents the architecture and details on the interface that allows to input data to the *New FastScan*.

4 https://wandb.ai/site

Chapter 8 presents the conclusions, reflections and future work of the Master Thesis.

2

BACKGROUND

This chapter covers some main concepts that are needed in order to better understand and follow the next chapters.

2.1 MACHINE LEARNING AND VULNERABILITY IDENTIFICATION

In this section it is presented the role of Machine learning in modern software security and more specifically in vulnerability identification.

Alan Turing question of "Can machines think?" in 1950 (Turing, 2009) set the path to the development and research in the *AI* field and in the attempt to respond the questions of what really defines a machine and the process of thinking. Having this in account, *Artificial Intelligence* (*AI*) can be defined as the science and engineering of making intelligent machines (McCarthy, 1998). But the definition of an intelligent machine is one of great debate, because of the underlying concept of a thinking machine and what really defines it. Instead of a conceptual definition we can refer to the way that *AI* is used, since it is a more relevant concept for this Master Thesis. *AI* is used as programs that are able to autonomously achieve and fulfill certain goals that would normally require human intelligence to perform.

Since this concept is so broad and it is used to group such diverse areas, it is necessary to acknowledge that there are many branches or subjects within this concept of *AI*, namely the machine learning area.

Machine Learning (*ML*) techniques attempt to build programs that improve autonomously through experience (Jordan and Mitchell, 2015). *ML* works by training, through multiple iterations, a system. The process basically consists in showing examples of a desired input and the expected output, instead of programming the desired responses for all possible inputs.

In a *ML* problem the objective is to improve the correct prediction rate, this can be achieved throw training experience. For example, learning to detect vulnerabilities in code, the task is to label methods as vulnerable or not vulnerable (a Boolean assignment). In order to improve the correct prediction it is important to give correctly labeled input and also non labeled input in order to diversify the training phase. So machine learning is still dependant

on human intervention because it requires the input of previously labeled datasets in order to train a model.

Artificial intelligence and more specifically machine learning and deep learning systems are being used to automate many different tasks and in different areas, with success. For example in image recognition, disease behaviour prediction, traffic prediction, virtual assistant, among others.

The area of software security is not an exception, as referred on the previous sections, current vulnerability identification tools and in general, all security tools have flaws and many rely on a great amount of manpower and are very time-consuming. In order to try to tackle such limitations and with the rise of many machine learning applications, there were many investigation focused on applying such techniques in the software security area Chan and Lippmann (2006).

2.2 FASTSCAN AND CODE2SEQ

code2seq creates a representation of source code using *Abstract Syntax Tree* (*AST*) and then uses it to infer properties.

An *AST* represents a source code snippet in a given language and grammar. The leaves of the tree are called terminals and the non-leafs are called non-terminals. It represents all the variables declarations, operator, conditions and assignments. In order to represent code in this structures it is necessary to create sequences of terminal and non terminal nodes and consider all possible paths between nodes.

This representation has some significant advantages over the use of tokens, namely when comparing two methods that have the same functionality but different implementations. Having the *AST* enables a better comparison since both functions paths will be similar as represented, in Figure 1 , functions will have different token representations but similar path representation only differing in the *Block* statement.

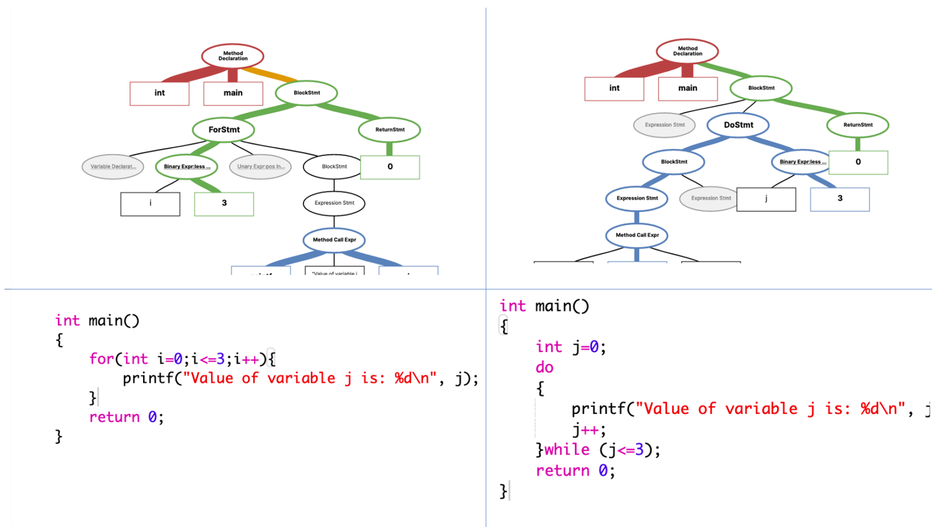


Figure 1: Schema about *code2seq*.

In a simplified overview *code2seq* uses encoder-decoder architecture that reads the *AST* paths instead of the tokens. In the encoding end, there is the creation of vectors for each path using a bi-directional LSTM and the extraction of tokens, where the *AST* terminal nodes are transformed into tokens and this tokens are split into subtokens (for example, an ArrayList is transformed into the tokens Array and List). In the end a decoder uses attention models to select relevant paths.

2.3 MACHINE LEARNING PERFORMANCE METRICS

In the following section it is presented the notions of accuracy, precision, recall and *f1*. These concepts are fundamental to understand some decisions and directions presented in the fallowing chapters, namely regarding hyperparameter optimization.

The metrics are :

- Accuracy is the most known performance measure and it is the result of a ratio between correctly predicted observation to the total observations. But it can be misleading if there are not the same number of false positives and false negatives, so it is important to explore other metrics;
- Precision refers to the ratio between correctly predicted positive observations and the total predicted positive observations. This metric answers the question of from the methods labeled as having a vulnerability, how much of them actually had one? A high precision means a low occurrence of false positives;

- Recall is often called sensitivity, it answers the question of, from the methods that really had a vulnerability how many of them were labeled?;
- The *f1* metric is the weighted average between precision and recall, this is the most balanced meausure of all the presented since takes false positives and false negatives into account. This metric is useful when there is uneven distribution in the dataset, in the case, where there is not the a close number between vulnerable and not vulnerable entries.

3

STATE OF THE ART

In this chapter, it will be presented the contextualization and state of the art.

The main spark that ignited the work in this thesis, and the work from specialists around the world, is software vulnerability. It is necessary to understand the concept of vulnerability and other main concepts related to them in order to understand all the methodologies presented next.

3.1 VULNERABILITIES

A vulnerability is as a flaw or weakness in a system design, implementation, or operation and management that could be exploited to violate the system's security policy (Shirey, 2007a). These flaws might be system design, development, or operation errors and it's exploit can lead to harmful outcomes. This perspective of harm or loss of information is normally called risk. Nowadays, these concepts have become common in any software development process because with the presence of electronic devices running software in almost every area of our society, there is the need to eliminate, or at least minimize, the occurrence of situations that can risk the security of data or operations.

There are different types and classifications of vulnerabilities, provided by different contributors. They expose discovered vulnerabilities, exploits, and solutions in online databases [1]. Even though there are many sources and knowledge about vulnerabilities, exploits still occur.

OWASP [2] foundation, that works towards a more secure software development process, has a list of the top tens security risks on software applications. Since one of the objectives of this master thesis work is to correctly identify types of vulnerabilities using machine learning models, first it is necessary to know them and understand how they use security breaches to harm systems. The top 10 stands as follows, according to *owasp* top 10, 2017 :

1. **Injection** : occurs when malicious data is sent via a query, the attacker makes the interpreter execute an unintended action;

1 https://cve.mitre.org/
2 https://owasp.org/www-project-top-ten/

2. **Broken Authentication** : occurs when there is poor management of the authentication process. This leads to the attacker be able to access the system with privileges that should not have;

3. **Sensitive Data Exposure** : occurs when there is exposure of sensitive data, for example, credit card information, that can lead to bank fraud;

4. ***XML External Entities* (*XXE*)** : occurs in applications that accept *XML* format to transmit data. The problem arises from the possibility to add parameters in the *XML*, that can have commands, like for example retrieve a certain password file;

5. **Broken Access Control** : occurs when there are flaws on the systems user permissions that allow an attacker to access a certain resource that was on a different access level than his;

6. **Security Misconfiguration** : occurs when there are errors on configurations such as *HTTP* headers;

7. ***Cross-Site Scripting* (*XSS*)** : occurs when an application allows an attacker to inject and execute scripts on the client side;

8. **Insecure Deserialization** : occurs when an attacker inserts non-validated input into an application and is able to execute malicious code. This atack is often an entry point to other attacks such as injections ;

9. **Using Components with Known Vulnerabilities** : this occurs when an application uses components with known vulnerabilities such as libraries or *Application Programming Interface* (*API*) ;

10. **Insufficient Logging & Monitoring** : occurs when there are low monitoring and procedures to allow to identify the presence of an attacker in the application.

Since this top ten is intended to identify the most serious security risks on web applications, and it is important to know them in-depth in order to know how to identify them, manually or automatically (on the presented work automatically), and even more realise what consequences can they bring to the system. To accomplish that, it is presented a more in-depth overview - Injection, Broken Authentication and *Cross-Site Scripting* (*XSS*). Also, as from September of 2021 a new top ten was in peer review [3] and reveals a new organization with the fall of Injection into the third place and also *XSS* was merged into injection. Also new vulnerabilities appear as a proof that security related tools must always be on a constant improve because new vulnerabilities and attacks are constantly being developed.

3 https://github.com/OWASP/Top10/

3.1.1 *Injection*

Starting with injection, an injection attack refers to an attack where untrusted data is supplied as input to a program. This input is then processed and changes the application's expected behaviour. Normally, this vulnerability is related to insufficient user input validation. Since this is a well known and one of the oldest exploits, that has some automatic tools in order to exploit without having much knowledge, makes this one of the most common and dangerous vulnerability. There are many types of injections, namely Code injection, Email Header Injection, Host Header Injection, *Lightweight Directory Access Protocol* (*LDAP*) injection, *SQL* injection among others.

Next is shown an example for SQL injection, where an attacker sends `'OR 1=1--` instead of a valid id, as shown in Figure 2. Without an input validation, this query will return all data contained in the table in this case the table `accounts`.

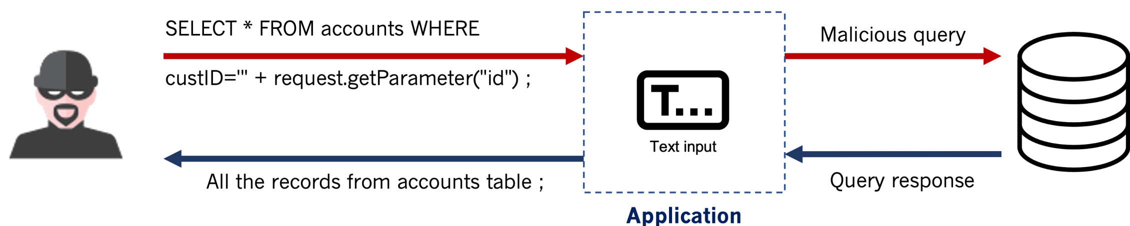


Figure 2: Schema explaining injection.

3.1.2 *Broken Authentication*

Broken Authentication attack refers to an attack that overrides authentications methods that are used by an application.

This vulnerability has a high risk of exploitability because it is fairly easy to create an attack vector due to the many repositories with password combinations and penetration tools such as Burp Suit[4] even provided in some Linux distributions, namely Kali Linux[5]. Since authentication is a part of almost every application there is a possibility that a flaw in the system programming and design can occur and lead to a successful attack by half of an attacker using manual, automated scan or a penetrating tool as referred previously. Furthermore, this type of vulnerability can have a huge impact on the system, since it is only necessary to break the authentication system for one specific user (such as a root or administrator user). Once an attacker has access to this user, all the other security mechanisms become useless because they can be bypassed by a successful broken

4 https://tools.kali.org/web-applications/burpsuite
5 https://www.kali.org/

authentication exploit. There are many techniques that can be used to perform such attacks, namely :

- **Credentials stuffing :** refers to an attack where a brute force method is used based on a list of common passwords. Then via try error method, an attacker, attempts to successfully login in a account ;
- **Unhashed Passwords :** refers to an attack where there are passwords being sent via a network in plain text. Then an attacker can scan a network and extract passwords from the intercepted requests ;
- **Misconfigured Session Timeouts :** refers to an attack where cookies or other session storage are mishandled. Then an attacker can use, for example, a cookie, execute some brute force process and get access to the system.

Next is presented a more in-depth example of Unhashed Passwords exploit. An attacker (located on the same network than the victim) scans the network using a tool such as *wireshark*, finding a vulnerable request using clear text submission of passwords - like seen in Figure 3. He then uses the username and password to impersonate someone else and access the system without having the privileges to do so. In order to prevent this vulnerability, the system must not use clear text password, passwords should be sent hashed, but it could not be enough since there are reports of phishing sites intercepting passwords before the hashing process, but in this case, the protection must be done by the user by validating if they are in fact using the original application/page and not a forged one (Ross et al., 2005).

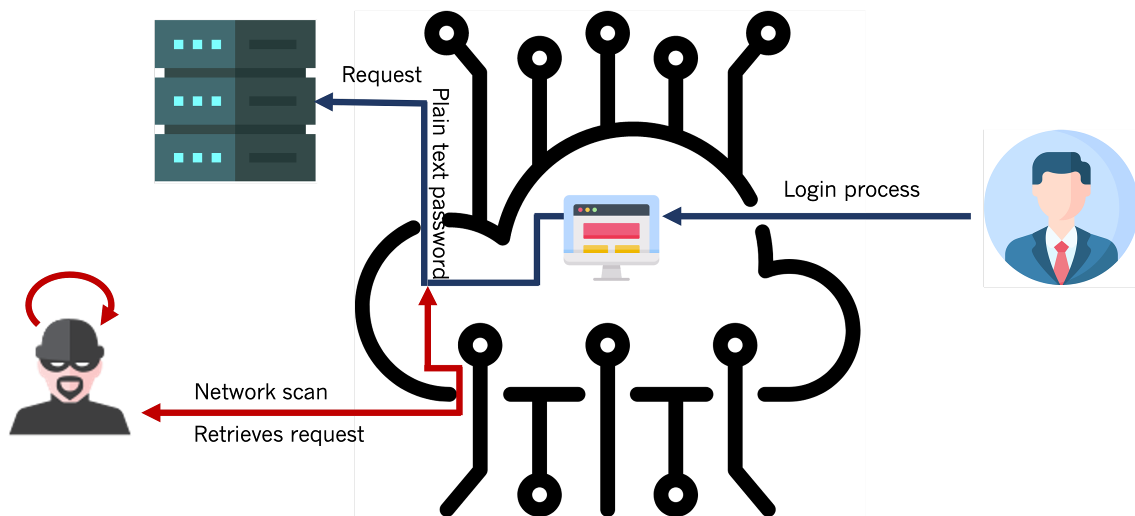


Figure 3: Schema explaining broken authentication.

This vulnerability can be mitigated by the use of two-factor authentication, weak password checkers, monitoring failed logins (in order to prevent brute force attacks), among others.

3.1.3 Cross-Site Scripting (XSS)

XSS is a type of attack against web applications, it occurs by injecting code into the output of an application that is then sent to a client/user. Normally is used to get access to sensitive data by an attacker (Vogt et al., 2007). The attacker usually injects a malicious script into a trusted website to access user data, take control of the browser and even take control of the application.

There are different types of *XSS*, namely :

- **Reflected *XSS* attack :** Reflected XSS attacks are the simplest *XSS* attacks, in these, the atacker passes a malicious script using a query, normally located within a link/url. This link is sent to users through spam emails, social media, sms and other comunication means. This attacks can be prevented in the client side.
- **Persistent *XSS* attack :** Also called stored *XSS* attacks, in this an attacker has to identify a vulnerability in the application that allows to inject a script, for example a input text box with no validation. This script is normally sent in a way that allows the attacker to store it in some persistent way, namely in a database. This attacks can be prevented in the client side but mainly there must be protection in the back-end.
- ***DOM*-based *XSS* attack :** the *Document Object Model* (*DOM*) definition is the data representation of the objects that comprise the structure and content of a document on the web[6]. In DOM-based XSS attacks data is written into the *DOM*. Attackers use this method to add a malicious script to a webpage. This attack is not possible to be prevented in the back-end because it only takes place in the client side.

Next it is presented a more detailed view over a persistent *XSS* attack [7]. Imagine there is an input text box with no validation in a web application blog where a user can make a post. An attacker create a post with the following content:

```
<script>
    window.location='http://atacker_domain.pt/?cookie='+document.cookie
</script>
```

The back-end receives this information that was supposed to be a post and then stores it persistently in its database. Now, when another user opens this post, the script will run in its browser, sending the current blog cookie in a post request to a domain controlled by the attacker. Now the attacker will have the user cookie information, this allows the attacker to impersonate a certain user and have access to information that is normally protected.

Web applications can be targeted with *XSS* attack regardless of the technology/language used to build their back-end. In this Master Thesis the analysis is done only over Java code.

6 https://developer.mozilla.org/en-US/docs/Web/API/Document$_{O}$*bject*$_{M}$*odel/Introduction*
7 https://www.stackhawk.com/blog/what-is-cross-site-scripting-xss/

3.2 VULNERABILITIES IDENTIFICATION

In many of these affected software development projects, there is no application of formal and systematic methods to execute source code auditing and testing. Auditing can be defined as the process of analyzing application code (in source or binary form) to uncover vulnerabilities that attackers might exploit (Dowd et al., 2006). This process is important because allows scanning the source code covering all the paths that normal testing might not cover in order to try to detect possible vulnerabilities.

Having in mind the big issue in software development presented before - vulnerabilities and how to detect them, there are many solutions that try to mitigate it.

Starting with the concept of defensive programming, that refers to the practice of coding having in mind possible security problems that can occur (Chess and West, 2007). This is clearly a good practice but one that does not solve the vulnerability related problems. Even good and experienced programmers cannot know how to prevent all the exploits created for certain *API*, library or other. Also, languages by construction were not built thinking on the way an attacker would take advantage of certain nuances. Taking as an example, the buffer overflow exploits, one the most common problem reported in vulnerabilities databases. This can occur due to mishandling of inputs but in its core is allowed by the construction of the language. Has it is easily confirmed by C or C++ that do not provide any built-in protection against accessing or overwriting data in memory (Cowan et al., 1998).

Since good intentions and practices are not enough to solve this problem, there was the need to apply and develop other more robust techniques that will be briefly presented next.

3.2.1 *Static analysis for security testing*

Static analysis methods are a resource for identifying logical, safety and also security vulnerabilities in software systems. They are normally automatic analysis tools and intend to avoid manual code inspections in order to save time and avoid the investment of resources in manual tasks that could be fruitless (Pistoia et al., 2007).

The first static analysis tools were difficult to use and limited in the accuracy and dependant of the used language. Currently, these tools evolved and are able to discover complex bugs/errors and can parse almost every coding language. Even more, these tools have integrated complex techniques such as code metrics, path analysis, semantic analysis, type checking, between others. Semantic analysis (that is an important topic in this thesis), allows the discovery of the basic structure and relation of the functions within the application (Lopes et al., 2009). This context information can be very important to understand and identify errors/bugs/vulnerabilities within code that require the knowledge of certain code paths. This helps to predict the behaviour of the program.

There are many tools in the market that provide *Static Application Security Testing* (*SAST*), one in special the *CxSAST* developed by *Checkmarx*. The purpose of this tool is to detect vulnerabilities in source code, and it follows a specifically created pipeline (Ferreira, 2019). This tool does not need to build or compile a software project's source code and it builds a logical representation of the code's elements and flows. Then this representation is used to execute queries for known security vulnerabilities for each programming language [8].

There are clear advantages in using these tools, namely :

- To reduce the development costs by finding issues sooner in the development cycle;
- To discover code level problems;
- To keep code complexity low;
- To detect problems that manual verification can not, for example, buffer overflows.

But there are some evident problems such as they struggle to identify functional problems. Even though static analysis tools do not identify all the code problems they allow testers to worry about the functional problems having the guarantee that the dynamic tools verify the rest.

3.2.2 *Dynamic analysis for security testing*

In order to find vulnerabilities, testing seems to be the easiest path to follow, and that is what *DAST* , that stands for *Dynamic Application Security Testing*, is based on. In *DAST*, a program is executed in real-time in order to test and evaluate it.

There are many tools in the market, for example, IBM Security AppScan Standard [9] that scans applications, identifies vulnerabilities, and generates reports with fix recommendations. So, their main objective is to find errors and vulnerabilities by repeatedly run applications so that all the program scenarios are covered.

In *DAST* the tested software is seen as a black box, and the tools or person performing the tests only interact with the application or software as users that have no knowledge of it's internal operations.

There are some advantages associated with dynamic analysis tools namely:

- To identify run-time vulnerabilities;
- To fix false positives given by *SAST*;
- To verify results from *SAST*;

8 https://checkmarx.atlassian.net/wiki/spaces/KC/pages/59211846/CxSAST+Overview
9 https://www.ibm.com/developerworks/library/se-scan/index.html

- Low false positives;
- The ease to reproduce and consequently detect the program path that is causing a vulnerability.

There are some identified disadvantages, such as the high consumption of time, because it is required the execution of a big quantity of tests in order to cover different scenarios and the possible inefficacy because testing might not cover all the possible cases of the software execution. Then there is the increase in production costs because the high demand for testing requires more man power to deal with it. And finally, these factors culminate in the increase of the software's final cost.

Regardless of these considerations, the success of this approach relies on a good choice of what to measure and what type of tests to run (Ernst, 2003). This approach mitigates security problems and it may be enough for some software pieces but this analysis may fail to cover all the program paths because it relies on the given inputs and scenarios that the tool/person feeds to the program or software.

3.2.3 *Interactive analysis for security testing*

Contrarily to *DAST* that looks at software pieces from the outside as a black box, *IAST* which stands for *Interactive Application Security Testing* aims to find vulnerabilities by analysing software from within such as *SAST*. But contrarily to *SAST* that works without running the software and similarly to *DAST*, *IAST* executes the analysis with the software running.

IAST uses different instruments to monitor a running application in order to gather information about the internal processes. These tools try to mitigate *SAST*'s and *DAST*'s limitations, namely, identify the specific place where a bug/vulnerability is located and lower the costs by integrating the monitoring at the beginning of the development pipeline and not only when the software development is terminated. By running from the software's inside has leverage over *DAST* and allows testers to define what paths or functionalities to cover, this can lead to misshandled bugs/vulnerabilities but if well thought can lead to gains in time and work efficiency contrarily to *SAST* that has full coverage over the software.

There are some tools in the market such as *Contrast Community Edition (CE)* [10], that have some identified advantages [11] such as :

- **Low false positives :** these tools can provide detailed information (code line location or function's names) that help testers to confirm the results;
- **Instant Feedback :** since the tool is scanning while the application is running, allows testers to have instant access to the results of the tests;

10 https://www.contrastsecurity.com/contrast-community-edition
11 https://resources.whitesourcesoftware.com/blog-whitesource/iast-interactive-application-security-testing

- **Highly Scalable:** easy deployment and the use and implementation is independent of the project's size;

- **Moves the testing process to the beginning of *Software Development Life Cycle* (*SDLC*):** problems get caught earlier in the development which leads to fewer costs and the guarantee of correctness in the entire software.

3.3 FASTSCAN

FastScan lays down the foundation of this thesis. As referred in Chapter 1, *code2seq* and *code2vec* were used as the base for the work.With this in mind, there were three different approaches.

It will be mentioned one (*code2vec* approach) and there will be more focus on the other two approaches because they were considered by the author as the ones with more contributing and potential for future work - the **code vectors for clustering and classifiers** and ***code2seq* embedding**. All the approaches were compared using three projects SecurityShеperd [12],WAVSEP [13] and WebGoat [14].

Starting with *code2vec embedding*, the main objective was to train a model capable to predict if a method is vulnerable or not. There were made some modifications in *code2vec* to make a custom labeling, the results were neither accurate nor promising (Ferreira, 2019).

In the second approach, there was a need to modify the default output of the code vectors from *code2vec*. By default, it outputs the method name and it's a vector but this is not enough for the proposed solution because in the original solution there was no problem in having different vectors for the same method name, but for the case study it is important to know specifically to which method the vector belongs in order to correctly identify where the vulnerability is located. So the author added a unique identifier, as well as a second output file with other important metadata information - method's filename, start line position in the file, start column position in the file, end line and end column in the file.

So the next phase is the *parser*, it analyses the files generated by *code2vec* and *XML* vulnerability report. In this process there are some important operations:

- Get information about vulnerabilities for each method: ID, name, vector, file name, and line-span ;

- Get CxSAST *XML* report information : Name, query path, severity and path nodes ;

- Merge of both information, leading to a data structure that for each method has the associated vulnerabilities ;

12 https://github.com/OWASP/SecurityShepherd
13 https://github.com/sectooladdict/wavsep
14 https://github.com/WebGoat/WebGoat

All these operations culminate on the information being stored in a data structure and written into a file that was used for experimentation. The previously explained steps are simplified and represented in Figure 4.

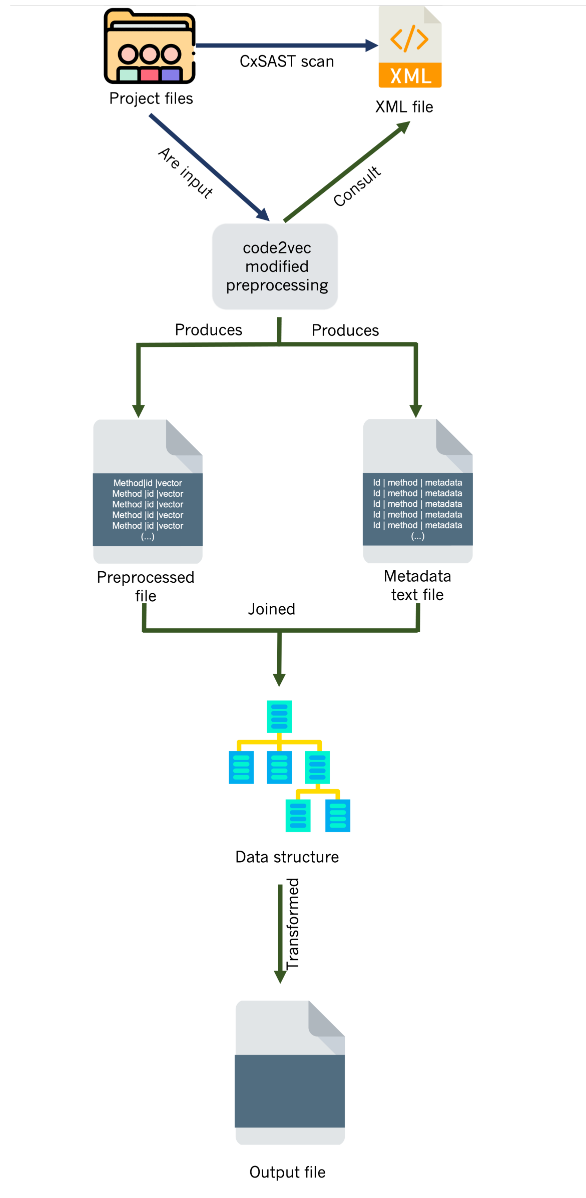


Figure 4: Schema explaining *code2vec* approach.

Regarding experimentation, there were two approaches. The first was to use code vectors to create clusters, there were two different types of clustering used, the hierarchical clustering and K-Means clustering. This approach failed with the author concluding that the goal of achieving a clear distinction between vulnerable and safe clusters is far from feasible. The second was using machine learning classifiers. The input was the produced file of Figure 4,

it was applied data balancing (for more details consult 4.3.3 from (Ferreira, 2019)) and then 60% of the methods were used in the training phase and 40% for accuracy testing. Then there were produced twelve models, using twelve different classification algorithms. Next are highlighted the ones the author classified has appropriated and that performed well :

- Gaussian Process: better accuracy for every dataset but resource-consuming and high test time ;
- Neural Net : High accuracy and low test time ;

The conclusion was that the Gaussian Process algorithm would be the best fit for use cases where the accuracy is absolutely crucial, but performance-wise, Neural Net would be the best choice. Both these approaches had low accuracy rates, sitting at only 70 %.

Ferreira (2019)'s third and final attempt was ***code2seq* embedding**. This approach consisted of altering *code2seq* instead of *code2vec* in order to train a model capable of predicting if a method is vulnerable or not.

Within this approach, there were two different paths, one using *CxSAST* preprocessed projects and another using the original projects.

In both cases (preprocessed and original projects) the modified code2seq preprocessor has the vulnerability information and also it's filename and line and column location and outputs text files. Then these files go through a data balancing process (for more details consult 4.3.3 from (Ferreira, 2019)) and are then used by *code2seq* to train and test different models. The previously explained steps are simplified and represented in Figure 5.

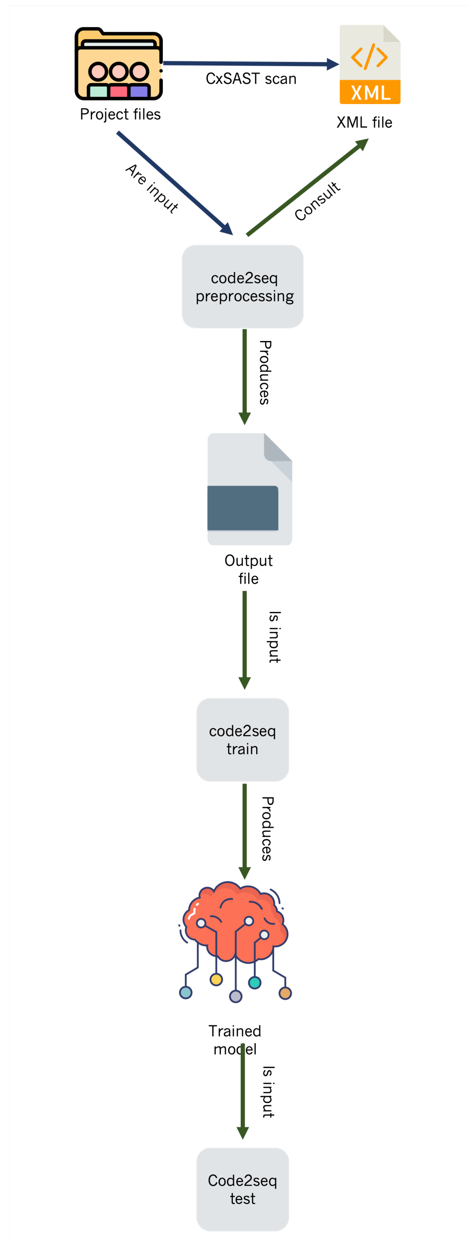


Figure 5: Schema explaining *code2seq* approach.

The process with the best results was the one using the original projects and not the preprocessed output by *CxSAST*. Using the input and sink nodes as the method for classification, the approach using *code2seq* (approach 3) had better accuracy results than the other approaches and even better than *CxSAST*.

3.4 OTHER APPROACHES

There are many different machine learning techniques applied to vulnerability detection.

In Hovsepyan et al. 2012, each "word" in the source code is seen as a feature using the bag-of-words technique to extract features and train an *SVM* model.

In Mou et al. 2014, the potential of deep learning was explored for program analysis by encoding the nodes of the abstract syntax tree representations of source code and training a tree-based convolutional neural network (CNN) for classification problems (Tanwar et al., 2020).

In (Li et al., 2018), there was created the concept of code gadget - a number of (not necessarily consecutive) lines of code that are semantically related to each other. It used code gadgets to represent programs and then transform then into vectors. The performed work was successful and more accurate when compared to other tools and even found 4 vulnerabilities in 3 software products that were not public but in fact, we're patched in more recent version.

In the work done in Briem et al. (2019), they have considered code embeddings which represents the semantic structure of a code block alone for bug prediction. They use code embeddings obtained by code2vec as a base to run a binary classification for off-by-one errors using data created by mutation of comparator operators.

In (Coimbra et al., 2021), *code2vec* model was trained to identify methods as vulnerable or not vulnerable with good accuracy and it was proven that the approach had potential and could be a path for a vulnerability detection tool.

All these approaches have points in common namely the code representation of code using vectors, the use of *code2vec* and the promising results. All of this information was taken into consideration when thinking about the approach in Chapter 4.

4 PROPOSED APPROACH

In this chapter, it is presented the system architecture and high-level representations, which constitute the approach to reach the objectives in Chapter 1.

4.1 SYSTEM ARCHITECTURE

In order to fulfil the objectives proposed in Chapter 1, it will be necessary to develop two different solutions.

To accomplish the development of a specific model for each type of vulnerability, that is capable to identify if code snippet has a vulnerability or not (Boolean model), it is proposed to follow the architecture and flow presented in Figure 6. This approach is a refinement of Ferreira's work with *code2seq*.

It is essential to use *CxSAST* in order to obtain the input for the preprocessing - since it converts several languages into a generic internal representation, allowing to create a language independent tool (as long as there is a preprocessing tool that transforms the code into the *AST* representation). Since the focus is on creating a model capable of identifying certain vulnerabilities then it is necessary to filter *CxSAST* output in order to correctly train the model. Then use *code2seq* normal pipeline in order to obtain a model able to identify a specific vulnerability. The goal is to have at least a model for each of *owasp* top 2 vulnerabilities - Injection and *XSS*.

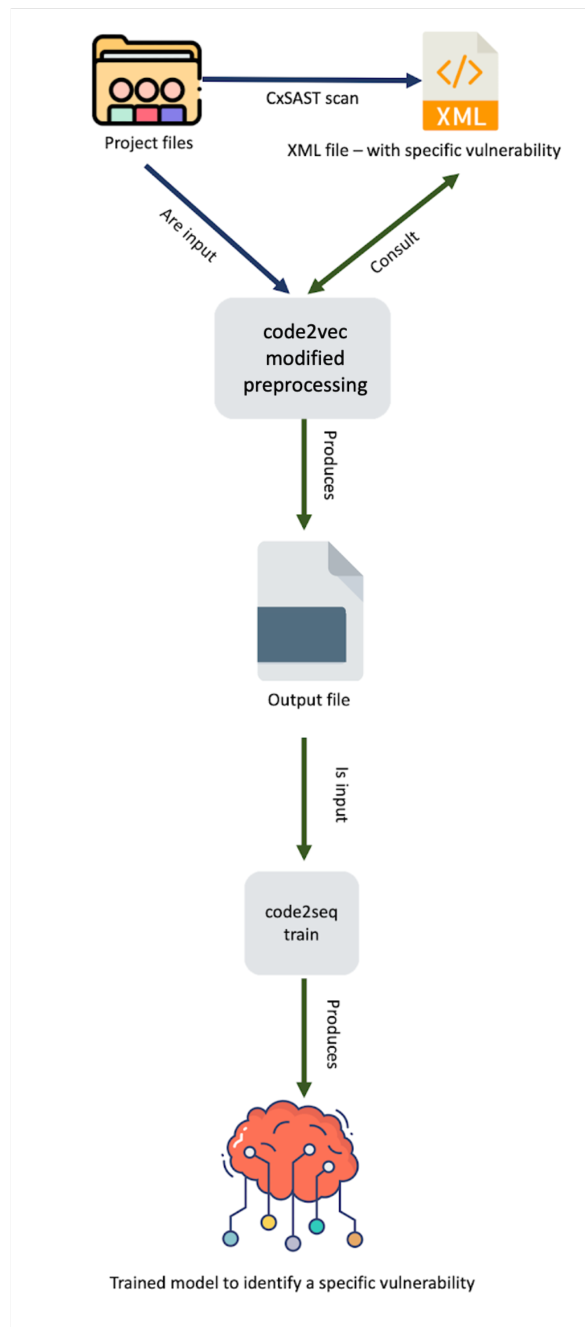


Figure 6: Schema explaining the first phase.

Having completed the first phase, it is necessary to develop a general model capable of identifying if there are vulnerabilities and it's type, given a code snippet. In order to complete this objective it is proposed to follow the architecture and flow presented in Figure 7. It is proposed to use the previously trained models and combine them. This combination

might be done using ensemble technique or only by running *code2seq* testing phase in parallel.

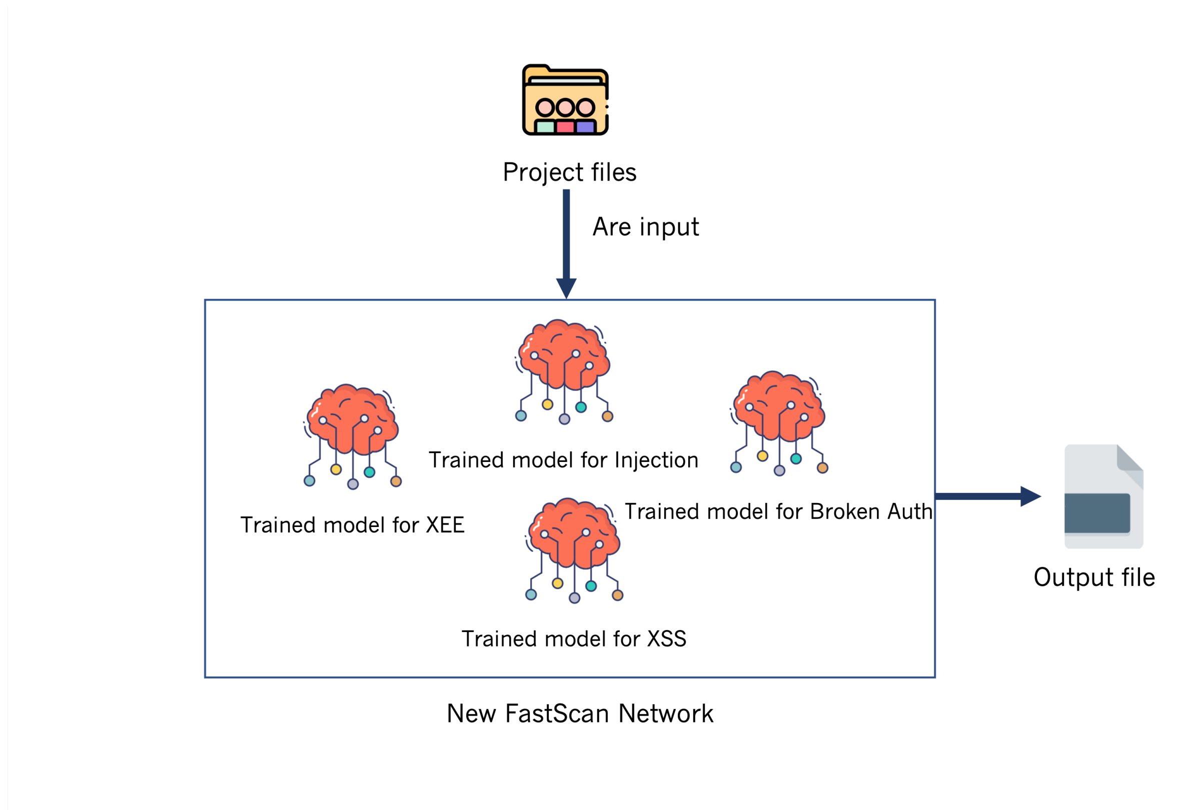


Figure 7: Schema explaining the second phase.

The developed pipeline will be deployed in University of Minho's cluster[1]. This cluster presented in Chapter 5 provides powerful hardware resources which maximize the development, test and consequently the results.

1 http://www4.di.uminho.pt/search/pt/equipamento.htm

5

DEVELOPMENT

FastScan, described in Section 3.3, had promising results and showed the potential on using machine learning for detecting vulnerabilities in source code having as a base the open source project code2seq. Nevertheless, it left open for investigation some issues, namely the ones that are a part of this thesis objectives, namely to tune code2seq parameters and to develop a way to identify specific types of vulnerabilities in source code.

This chapter will describe the different phases that were explored during this Master Thesis work. The first section will describe the datasets used in the different phases, the hardware and technical details on which the experiments were made and the following sections will detail the work done in each phase. Explaining the steps in each one, the challenges, experiments and the obtained results for further analysis in the next chapter.

5.1 DATASETS

During the development of this project, two datasets were used, they are composed by the aggregation of different projects and were grouped this way because of the time it were received in the development.

The first dataset, **dt01** is composed of **43** different projects. It contains the original source code of these projects and the *CxSAST XML* report for each. All results in the report are true positives, validated by humans. In total there are **2444** different entries with vulnerabilities in all the *CxSAST* files. An entry, is considered the attack vector for a specific vulnerability, each entry is a set of nodes with functions that are contained within a vulnerability. This dataset was used in the training of the injection model with a total 418703 functions.

The second dataset **dt02**, is composed of **36** different projects. There is the original source code and for each project there is an *json* file provided by *Checkmarx* and it is the output of their static analysis tool *CxSAST*. Also, such as **dt01**, in **dt02** all the results are true positives, validated by humans. In total there are **3436** different entries with vulnerabilities of the type *XSS* in all the *dt02*.

5.2 HARDWARE AND TECHNICAL DETAILS

In the previous work by (Ferreira, 2019), the low computational power was identified has a major barrier and setback to the development. It became clear that the use of a regular personal computer was not enough to achieve the desired results in the experiments of this Master Thesis.

This barrier was overcame with the use of the University of Minho cluster [1]. More specifically using a node with the following specifications:

- **CPU as seen in Figure 8:**
  - Intel® Xeon® Processor E5-2695 v2 ; [2]
  - twelve cores ;
  - twenty four threads.
- **RAM :** 64 GB as seen in Figure 9 ;
- **GPU :** Two NVIDIA® TESLA® K20M . [3]

```
vendor_id       : GenuineIntel
cpu family      : 6
model           : 62
model name      : Intel(R) Xeon(R) CPU E5-2695 v2 @ 2.40GHz
stepping        : 4
microcode       : 0x428
cpu MHz         : 1228.500
cache size      : 30720 KB
```

Figure 8: CPU details.

```
[a75328@compute-662-1 ~]$ cat /proc/meminfo
MemTotal:       65769424 kB
```

Figure 9: Memory details.

There were problems installing all the needed dependencies to run *new FastScan* natively, because of the security constraints in the cluster system. Namely, there were no root permissions, so the installation of dependencies such as *JAVA*, *python* and it's libraries were a great technical obstacle. To overcome this, it was used docker. Docker presented a good

1 http://search6.di.uminho.pt
2 https://ark.intel.com/content/www/us/en/ark/products/75281/intel-xeon-processor-e5-2695-v2-30m-cache-2-40-ghz.html
3 https://www.techpowerup.com/gpu-specs/tesla-k20m.c2029

solution because it is consistent and makes all this installation process easy by using a base image with *JAVA* and adding all the dependencies needed without further concerns.

It is important to notice that this solution was followed after the confirmation that there was no significant overhead or performance loss introduced by the use of docker and that it was a better solution than a virtual machine (Felter et al., 2015).

5.3 FIRST PHASE

In this section is explained the first phase referred previously in Chapter 4. This phase works *dt01* out to output a trained model for a specific and predetermined vulnerability .

5.3.1 *Data filtering*

The objective of new *Fastscan's* first phase is to create a model capable of detecting a specific vulnerability, therefore it is required to filter the original dataset input data by vulnerability type. So that it is possible to train a model to predict only a specific type of vulnerability.

Before knowing how it was achieved, it is important to understand the two different components from the dataset:

- Source code files: These are the original files from several different projects;
- Result files: These are the files generated by the *CxSAST* from *Checkmarx*, in *XML* or *JSON* format. There is one file for each scanned project, on which are registered the vulnerabilities detected by the tool.

This filtering process was developed using *python* and aims to filter the *XML* or *JSON* files of each project, to keep only the ones referring to the vulnerability for which it is desired to train a model. It was developed a python program that is able to read and parse the *CxSAST* result files and filter the vulnerabilities entries by the vulnerability name. It is not a literal search, one can send arguments to the program with the range of words/strings to be searched in the vulnerability name field of the vulnerabilities entries. In the end, the output is a set of files only with the entries for a specific vulnerability and ready to be ingested in the next step, the preprocessing.

5.3.2 *Preprocessing*

This step was not changed from the *FastScan*, but it is different from the original *code2seq*. The preprocessing (which is performed on the train and test data) returns a file with the dataset labeled and the *AST* that represents the input dataset. The source code files and

XML files are parsed , in the preprocessing, which is built in *Java*. There is the creation of the *AST* from the input of the source code files and this was not modified from the original *code2seq*. But there is another step required to obtain the label for each method, the parsing of the *XML* or *JSON* files.

It is also performed the parsing from the *XML* or *JSON* file. In this step it is relevant to refer that the vulnerabilities metadata registered in the file is stored. This metadata is constituted by the name of the vulnerability, the start and end line and column in the file and also the filename and path within the project.

In the parsing stage of the source code used to train and test the models, each method in the source code files is analysed and transformed into its *AST* representation, then it is verified if the method is marked as vulnerable in the *XML* or *JSON* files, depending of the dataset.

This is achieved by comparing the file name and path within the project with the information that is stored in the result of the *XML*/*JSON*. Then it is registered the presence or not of vulnerabilities, with a *boolean*.

This combination of the parsing of the *XML*/*JSON* (with the *CxSAST* report) and the original source code leads to the output of the preprocessing, that is constituted by a text file that has a line for each method, on which the first element is the label (boolean) indicating the presence of vulnerabilities followed by the *AST* paths.

On the original *code2seq* preprocessing, in the output file the first entry was the function's name - the first entry is the label that will be used in the training phase. But *New Fastscan* preprocessing was modified in order to make the first entry of each line a boolean that indicates the presence or not of a vulnerability in the method. This change was enough to modify the prediction of the model obtained in the next training phase, since the training is now gonna be done with the boolean has the prediction label instead of the previous string that was the method name.

5.3.3 *Hyperparameter optimization*

This step was important to tackle a problem detected in *FastScan*, without the optimization of the hyperparameters the final model performance might not be the best possible Ferreira (2019).

As explained previously in Section 2.3, there are different metrics and *f1* was presented as the best metric to take into account for this work.

In order to obtain the best *f1* metric, it was used *wandb*[4] - It is a python library, used to monitor, compare, optimize and visualize machine learning experiments. It can be integrated with many frameworks including *tensorflow* (the python library used in the training) and it

4 https://wandb.ai/site

was of great importance in the development of this work. It implements different search methods in order to obtain the best parameters to increase a specific evaluation metric. This library allowed to solve the hyperparameter tuning which is a very complicated problem that normally requires experience in the field and complicated algorithms Snoek et al. (2012).

This library allowed the optimization of the models hyperparameters for a specific dataset by using *sweeps* that allow to find a set of hyperparameters with optimal performance and apply different methods to do so, such as grid, random and Bayesian [5]. It also allows to visualize performance and time metrics from the experiments, that can be consulted in section 6.1.

The chosen method was the Bayesian method because it guaranteed the best possible combination without compromising time and performance. There were other methods such as random search where the parameters are being chosen randomly from a specific range - this method could be even faster but it would not guarantee the best hyperparameters since it does not cover all the combinations and it does not have a method to improve its results. Other method could be the grid search, in this method all the possible hyperparameter combinations are tested, in this case it would be of great time and processing cost given the number of hyperparameters.

The library applies the Bayesian method by building a probabilistic model that maps the value of each hyperparameters with the values to optimize - that could be accuracy, f1 and precision. Each hyperparameter value used in the algorithm iteration and the obtained results are taken into account into the next iterations. This way the search for the best hyperparameters is faster because it is being guided by the previous experiences.

5.3.4 *Training*

This step refers to the effective training and production of the models. It is important to get to know more about the models architecture, why it was used originally in *code2seq* and the advantages over other approaches.

First, a simple *Recurrent Neural Network* (*RNN*) cell is not enough to process sequential data (as is the data input as a *AST*), so when dealing with data represented in sequences it is necessary to connect multiple cells in sequence allowing each cell to send its output to the next, so each cell as in account the previous states.

Furthermore, the model follows the encoder-decoder architecture, on the encoding phase the model takes the input *AST* and transforms it in a compact representation of each *AST* path. Then the decoder receives the compact representation and generates the output, in the *NewFastScan* case a boolean (*codes2seq* case was a sequence). With the detail, that are used attention mechanism. Namely, for each step in the decoding phase the previous results

5 https://docs.wandb.ai/guides/sweeps

from other cells are taken into consideration, more specifically the context vector (calculated by joining the hidden state (which encodes not a prediction but some information about the sequence that can help in the prediction) and the output of the previous cells) and the decoding state.

This architecture was used because it is proven to perform when dealing with the input of large sequences, also to model long-range dependencies (as it may be the case of vulnerabilities that start and end in different function) and also because of the recommendation and results obtained in *FastScan* (Ferreira, 2019).

The main disadvantages might be the excess of complexity of the cells, the training requires high resources and data size and the prone to occur overfiting and the difficulty to solve such problem even using dropout algorithms.

All of the previous was not developed or changed, it is presented because it was studied to give context and allow a better understanding of the project. In practise, the training is performed the same way has in the original *code2seq*, relying on the developed approach that uses *tensorflow* in order to perform all the process.

5.4 SECOND PHASE

This section describes the developed work regarding the second phase approach proposed previously in section 4. After having two different models trained using the previously described approach for the first phase, the second phase was started.

The architecture from the proposed approach in section 4 was refined since the presented before was only a conceptual idea that needed more work and detail.

Firstly, there was the creation of a python controller that is responsible for receiving the project files transform them into the *AST* representation through preprocessing. Then launch the parallel scans for each of the previously trained model using the methods described in Section 6.

Then the prediction output was changed in order to not only output the presence of a certain type of vulnerability but also the function name, filename and file location (begin and end line). This was possible due to changes in the preprocessing, more than simply outputting the *AST* representation, the preprocessing also outputs meta information about each function that was transformed. This metadata is consulted and merged into the final predictions, allowing the *New Fastscan* to output a more complete and useful output.

This changes were important because many projects have functions with the same name, this way, after the scan one can precisely identify where vulnerabilities occur. The conceptual final schema is represented in Figure 10.

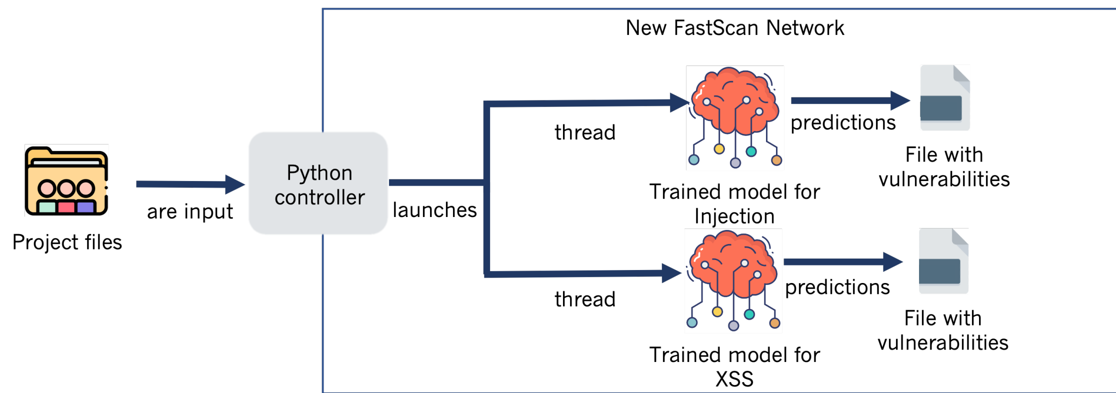


Figure 10: Schema explaining second phase.

6

RESULTS

6.1 FIRST PHASE

In this section will be presented the obtained results. The presented results were obtained using different datasets and the training was performed to target different types of vulnerabilities.

Injection results

For this experiment was used dt01 because it had more entries for the injection type and was the first dataset available to work on. Only the unknown part of the dataset was used in order to effectively test the models, the division was by number of projects, 75% for training (33 projects) and the rest for validation and testing (10 projects). First, in the data filtering step it was applied the filter of injection, in order to filter the results for this specific vulnerability so that it is possible to train the model only to identify injection vulnerabilities. Then, the preprocessing was applied to the source code and the filtered XML file obtaining the preprocessed input to train and test the final model.

The next phase was the hyperparameter optimization. Firstly it was attempted to run the script to obtain the best parameters that maximize precision. After analysing the results in table 1 it was observed that it was possible to obtain a really high precision but at the cost of very low recall.

A model with high precision but low recall returns a low count of results, but most of the predicted labels are correct when compared to the training labels, on the other hand a model with high recall but low precision returns a high count of results, but most of the predicted labels are incorrect when compared to the training labels. After this consideration it was clear that a vulnerability prediction system must have a balance between both because it is important to correctly identify a vulnerability but it is also important not to overlook one. This balance can be obtained by using the *f1* metric, *f1* is a weighted mean of the precision and recall and being so if in the optimization is focused in optimize the *f1* value then it is guaranteed the balance it is searched between precision and recall as seen in equation 1 .

$$f1 = 2 * precision * recall \,/\, precision + recall \quad (1)$$

After this, the model was trained using the best hyperparameters, as seen in Table 1.

Table 1: Best Hyperparameter for *dt01*

BATCH_SIZE	BEAM_WIDTH	BIRNN	CSV_BUFFER_SIZE
127	0	true	104857600
DATA_NUM_CONTEXTS	DECODER_SIZE	EMBEDDINGS_DROPOUT_KEEP_PROB	EMBEDDINGS_SIZE
0	302	0.4162172547338106	193
MAX_CONTEXTS	MAX_NAME_PARTS	MAX_PATH_LENGTH	MAX_TARGET_PARTS
234	8	10	6
NUM_DECODER_LAYERS	NUM_EPOCHS	PATIENCE	RANDOM_CONTEXTS
1	3000	4	true
READER_NUM_PARALLEL_BATCHES	RELEASE	RNN_DROPOUT_KEEP_PROB	RNN_SIZE
1	false	0.7488847205115016	256
SAVE_EVERY_EPOCHS	SHUFFLE_BUFFER_SIZE	SUBTOKENS_VOCAB_MAX_SIZE	TEST_BATCH_SIZE
1	10000	151701	256
TARGET_VOCAB_MAX_SIZE			
27000			

The final model for injection using the *dt01* dataset had 85% of accuracy, 90% of precision, 97% of recall leading to an *f1* of 93% and it was achieved in the first epoch as seen in figure 11, after this the model training leads to the increase of precision but with the lowering of recall values and therefore a lower *f1* value.

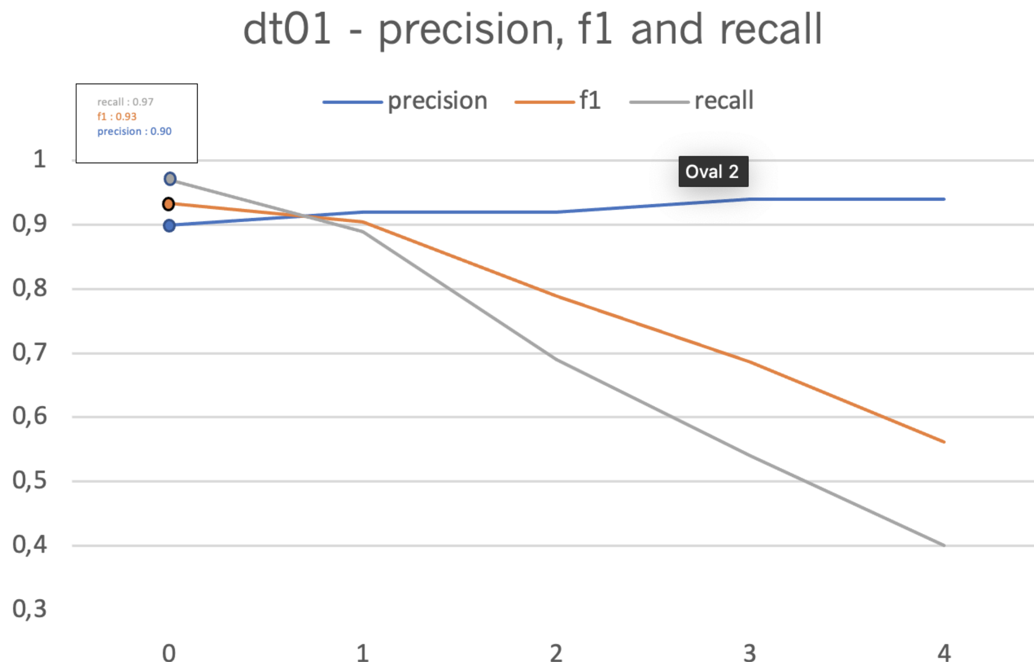


Figure 11: Injection model evaluation.

Regarding dt01 and this model trained for injection, it took 38 minutes to transform the datasets functions into their *AST* representation and 20 minutes to obtain the trained model as seen in Table 2.

dt01	
Number of functions	418703
Preprocessing time (min)	~38
Training time (min)	~20
Vulnerability type	Injection

Table 2: dt01 details.

XSS *results*

For the training of the model to predict *XSS* vulnerabilities was used the *dt02* because it had more entries for this type of vulnerability. Sames as in *dt01* only the unknown part of the dataset was used and the division was by number of projects, 75% for training (27 projects) and the rest for validation and testing (9 projects).

The followed process was similar to the described previously for injection. First in the data filtering step, it was applied the filter of *XSS*, in order to filter the results for this specific vulnerability so that it is possible to train the model only to identify *XSS* vulnerabilities.

Since the results provided by the *CxSAST* were in *JSON* format instead of *XML*, the preprocessing had to be modified to accept the new format. This was achieved by creating a new parser for the *JSON* in order to retrieve the desired information. Then the preprocessing was applied to the the source code and the filtered *JSON* file obtaining the preprocessed input to train and test the final model.

After this, the same process as described previously was done using *wandb* and the baeysian optimization in order to obtain the best hyperparameters, that are described in Table 3.

Table 3: Best Hyperparameter for *dt02*

BATCH_SIZE	BEAM_WIDTH	BIRNN	CSV_BUFFER_SIZE
101	0	TRUE	104857600
DATA_NUM_CONTEXTS	DECODER_SIZE	EMBEDDINGS_DROPOUT_KEEP_PROB	EMBEDDINGS_SIZE
0	334	0.7169372952597639	127
MAX_CONTEXTS	MAX_NAME_PARTS	MAX_PATH_LENGTH	MAX_TARGET_PARTS
113	9	11	6
NUM_DECODER_LAYERS	NUM_EPOCHS	PATIENCE	RANDOM_CONTEXTS
1	3000	4	true
READER_NUM_PARALLEL_BATCHES	RELEASE	RNN_DROPOUT_KEEP_PROB	RNN_SIZE
1	false	0.43619669604507155	256
SAVE_EVERY_EPOCHS	SHUFFLE_BUFFER_SIZE	SUBTOKENS_VOCAB_MAX_SIZE	TEST_BATCH_SIZE
1	10000	231267	256
TARGET_VOCAB_MAX_SIZE			
27000			

Then taking the best hyperparameters obtained by using the University of Minho Search cluster running successive attempts using the *wandb* bayesian implementation, the training was performed obtaining the following results performed in the dataset that contained 48303 functions. The results are presented in Figure 12, and the best score was obtained in the ninth epoch with 93% of f1.

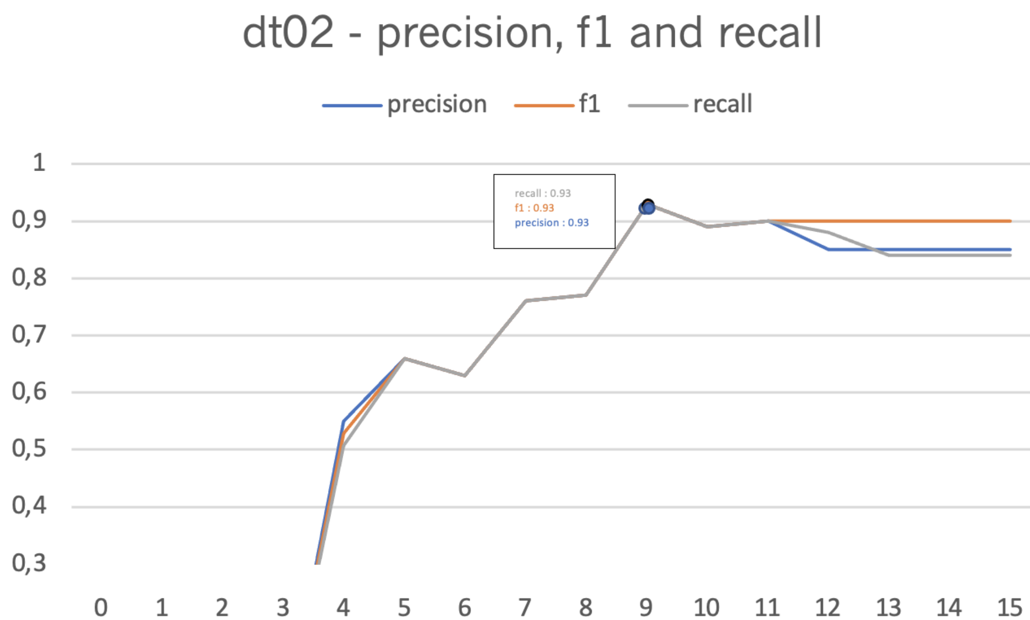


Figure 12: *XSS* model evaluation.

Regarding dt02 and this model trained for *XSS*, it took twelve minutes to transform the datasets functions into their *AST* representation and ten minutes to obtain the trained model as seen in Table 4.

dt02	
Number of functions	48303
Preprocessing time (min)	∼12
Training time (min)	∼10
Vulnerability type	*XSS*

Table 4: dt02 details.

6.2 SECOND PHASE

In the second phase there were made some tests both in the Injection and *XSS* models using previously unknown datasets for the models.

To conduct the tests, the input project files are given to the python controller which then preprocesses them and launches each model obtaining at the end a file with the results.

As dataset were used small examples with known vulnerabilities, provided by Checkmarx. During these tests appeared the suspicion that the models were not performed as observed in training, because the models were not able to detect the vulnerabilities or would rarely detect them.

So, these tests allowed to confirm that the developed architecture explained in Chapter 6.2 was indeed working but also to detect a problem/anomaly in the models.

In order to confirm or dismiss this suspicious, it was clear the need for a deeper, more precise testing . Namely, one that would allow to measure the results with the same measures as in the training phase.

To do so it was used the opensource project Lucee4 [1] and the *CxSAST* scan results that on determined version from 26204 functions had 6 injection vulnerabilities. This project was scanned by the injection model and detected three out of six vulnerabilities but had a really high rate of false positives as seen in Table 5.

This test was performed to evaluate the model's behaviour in a project with low vulnerability count. Given that, a low vulnerability count project requires a model with good performance to correctly predict the presence of vulnerabilities.

Lucee4 project	
Number of functions	26204
Number of injection vulnerable methods	6
True positives	3
False Positives	1326
True Negatives	24872
False Negatives	3

Table 5: Lucee4 project details.

As seen in Table 5, these were the obtained results. Taking these results and applying the previously referred metrics (accuracy, precision, f1 and recall) it was obtained :

- Accuracy = (TP + TN) / (TP + TN + FP + FN) = (24872 + 3)/26204 $\sim$ 0,949
- Recall = (TP) / (TP + FN) = (3)/6 = 0,5
- Precision = (TP) / (TP + FP) = (3)/6+1326 $\sim$ 0
- F1 = = 2TP / (2TP + FP + FN) = 6/6+1326 $\sim$ 0

These metrics show that the model is much better in the training and testing set than in an external validation dataset, this means that the model might memorized training

1 https://github.com/lucee/Lucee4

examples and adapted in order to respond effectively to the training data. This leads to a great performance over the training data and a poor response when facing new data, the model is not capable of generalize to the unseen example and so fails to make accurate predictions. This behaviour of fitting the training data too well is known as over-fitting, basically the neural network during the training period does not improve its ability to make a prediction anymore, instead it learns some random pattern contained in the set of training patterns allowing it to have a good performance within the training data (Jabbar and Khan, 2015).

In order to prevent and solve this behaviour there are some solutions, namely :

- **More data :** With millions of training examples, it is very unlikely that a model over-fits. It would have to memorize too many training examples and also it will not see the same example many times because there are many more than previously. This could be a viable solution since it is recognized that the amount of data might not be enough for the proposed task of predicting vulnerabilities;
- **Early stopping :** Stopping the training earlier could be a solution, but it is a difficult one to do, since it is not easy to decide when is the right time to stop the training only by observation of the learning curve. It could be a solution since the training was super optimized for the dataset by the hyperparameter phase and also it was not stopped, it only stopped after 5 epochs of not achieving any improvements;
- **Smaller model :** A smaller model with a less complex architecture and less layers can lead to a model that will not be as prone to memorize examples. It is a more complicated solution, since the model's architecture is based on code2seq Long Short-Term Memory. And to drop and develop a new architecture would be almost discard all of of the presented and developed work.

6.3 WEBGOAT TEST

In this section is presented the test performed in order to confirm if it was possible to get valid results from both *XSS* and Injection vulnerabilities. For that it was used the wegoat project dataset[2] since it has multiple examples of both vulnerabilities.

Firstly the *CxSAST* was used to obtain the results in order to make a comparison with the results obtained by the models, such results were not human validated.

Then the Webgoat project was inputed into the *NewFastScan*, that runned the pipeline, first preprocessing the inputed java code and then passing it into the Injection and *XSS* models to obtain their previsions.

2 https://github.com/WebGoat/WebGoat

NewFastScan marked 150 vulnerabilities and *CxSAST* found 70. *NewFastScan* marked more methods as vulnerable, so a more in depth analysis was performed to evaluate and compare the results between both engines.

Taking *CxSAST* results has the metric there were 130 false positives and 20 true positives (summary can be found in Table 6). All of the results were checked by using *python* scripts to compare both results file in order to confirm that the 20 true positives were in fact present on both result files. Then the code of all the false negatives was manually checked to verify if there was any entry that could be a vulnerability that *CxSAST* did not marked and *NewFastScan* did.

It was concluded that there was only one result that was consensual to be marked as a true positive from the 130, and it was of the type *XSS*. This was the only result found by *NewFastScan* that was not in the *CxSAST* results that was considered vulnerable.

WebGoat project	True Positives	False Positives
CxSAST	70	-
NewFastScan	20	130

Table 6: Webgoat results summary

This test was of the uttermost importance because it proved the initial concept that this approach could allow to identify different vulnerabilities in a single project and also showed that with an improved model there might be the identification of vulnerabilities missed by a *SAST* tool.

7

USER INTERFACE

This chapter intends to present the developed user interface and the approach the reasons, objectives and architecture behind it. Also it is intended to demonstrate the functionality and design.

The *User Interface* (*UI*) was developed with the main goal to create a way to visualize the results of scanning a project. It was intended to create a workflow that allowed users to upload *JAVA* project files and obtain and inspect the positive results.

7.1 ARCHITECTURE

The *UI* was built using a monolithic architecture since it was intended to be a proof of concept (Gos and Zabierowski, 2020). This architecture was chosen to focus on the functionalities rather than on scalability and high performance. Mainly it was intended to show the potentialities - the creation of a powerful tool based on the trained models as for example a web application or a open API that could be used by other services/clients. Also to allow a better analysis of the model's results, better than the manual analysis from a file containing predictions (which was the output of the second phase referred in section 6.2).

The chosen technology for the back-end development was *django*, it was chosen because it was considered the best fit regarding this case because all of the previous work was done in python, so using a python framework made sense since it allowed to better integrate all of the previous work namely regarding the preprocessing and also the predictions.

The chosen technology for the front-end was *reactJS* because it is a well proven, with much support and with good performance tool. Also because of the previous knowledge and experience in academic and professional level, which was a big factor since it allowed a faster development.

7.1.1 *Back-end*

Regarding the back-end there were two main tasks :

- To develop the *API* using *django*, that allows a client to send *JAVA* project files and get as response the results for each function in the project ;
- To adapt and integrate the prediction and preprocessing scripts developed in the second phase (Section 6.2) with the developed *API*.

Referring to the prediction script the task consisted in adapting it from the previous developed simple *python* script that received arguments from the terminal to a class that had a main method and constructor in order to be imported and declared in the *API*.

The following routes developed in the *API* were :

- **/results/files/:projectName :** this route receives post request with a body field named *project* which must contain the project files. This field must exist and contain a zip file in order to be a valid request. Once a valid request is received the *API* calls the preprocessing and then feeds the models by launching a thread for each model to make the predictions for each function (as explained in Section 6.2). Also there is the parameter *projectName* which is meant to indicate the project name the user wants to give to the uploaded files, this name is not used for storage purpose because of safety and to prevent duplicate locations. Before a scan is started it is created a directory (randomly generated) on which all of the information regarding an uploaded project is stored - preprocessing result and scan results, both in files. Next it is presented a sample result output, in order to show the structure of the *JSON* response :

```
[
  {
    "function_name": "on_update",
    "filename": "/api/project_files/01fe8784-0b73-4746-9e30-3330603c8756/servlet/LuceeServlet.java",
    "begin_line": "37",
    "end_line": "44",
    "type": "Injection",
    "prediction": "true"
  }
]
```

- **/results/code/ :** this route receives a post request with a body field named *path*, this variable must contain the relative path to a specific file. This route is important so that it is possible for the front-end to show where a vulnerability occurs in the source code. This route returns the content of the file that was sent in the body field.

7.1.2 *Front-end*

Regarding the front-end there were the following tasks :

- To create a design for the interface, this task resulted in the development of the following mockups :

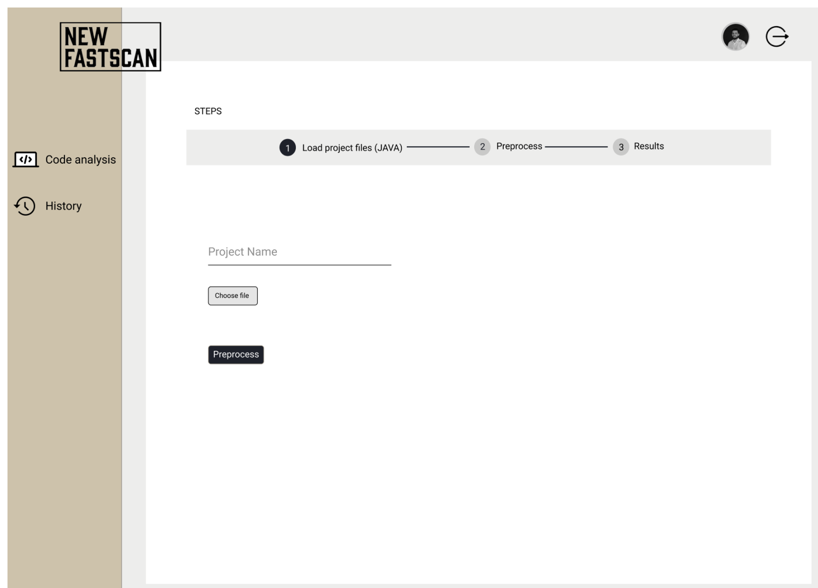


Figure 13: Home page design.

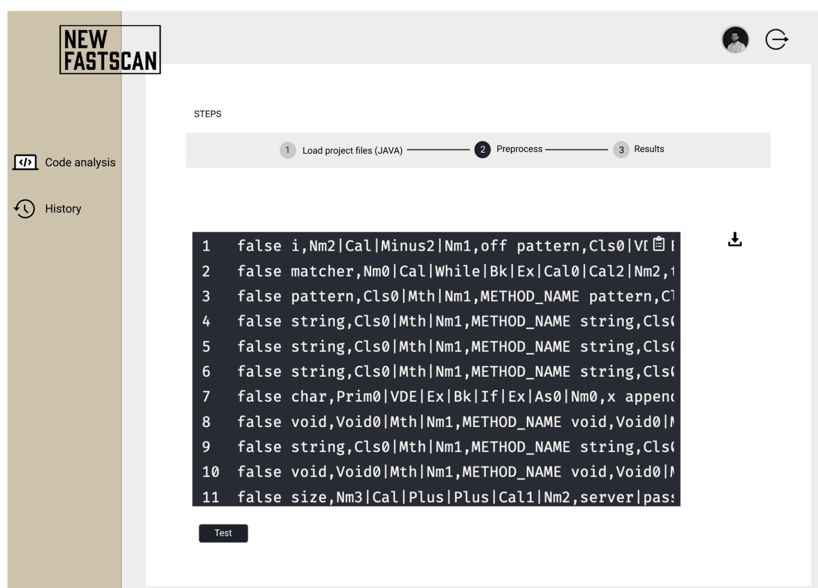


Figure 14: Preprocessing consult design.

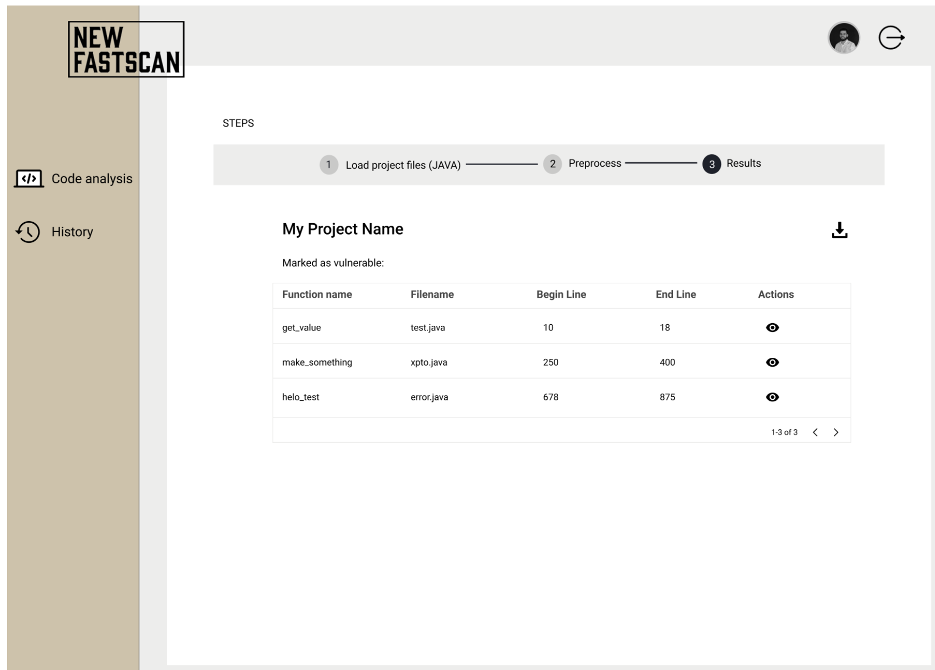


Figure 15: Results list design.

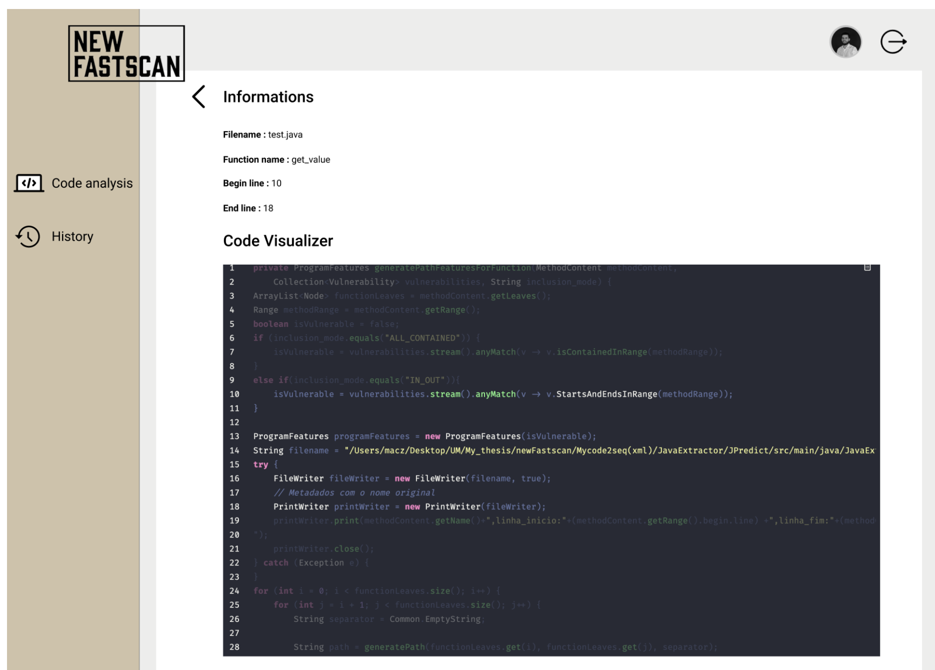


Figure 16: Code result visualization design.

The mockups were made in order to make the development process faster and consistent with the initial concept and idea. It was made a draft for the home page where it was idealized to be possible to load a project as seen in Figure 13. Then a page to visualize and download the preprocessing result (which was considered an over achievement and it was removed in the development phase, because most users will not be interested in such information) as seen in Figure 14. Followed by a page to visualize all the scanned functions from the project (Figure 15), its results and a way to visualize them inside the application (Figure 16).

This development was achieved by using *figma* [1], wich is a platform that allows web and mobile design and prototyping of applications.

- To transform the mockups into the interface in the *reactJS*. It was chosen the react hooks approach instead of the more classical components. Mainly because it is the most recent paradigm in *reactJS* that enable a functional programming approach that makes code easier to understand, test and because of previous experiences. In order to simplify the creation of the visual elements it was used the library *material-ui* [2], only applying different styling and minor modifications to apply the design from the mockups.

7.1.3 *Installation and general overview*

So that the application can be installed in a testing machine it is necessary to take into account the need for a web server, *nginx* is a good option since it is simple and efficient provider.

Also it is necessary to build the *reactJS* project (tranform the JSX code and assets into a static page using the automatic process built into reactJS) and to install and configure *uWSGI* with *nginx* in order to serve the *API* that was built in *django*.

Taking the previously explained into account it is possible to get a better perception and visualise the complete stack in Figure 17. Also it is possible to understand where each technology is located, their main task and also the data flow between them.

1 https://www.figma.com/
2 https://material-ui.com/pt/

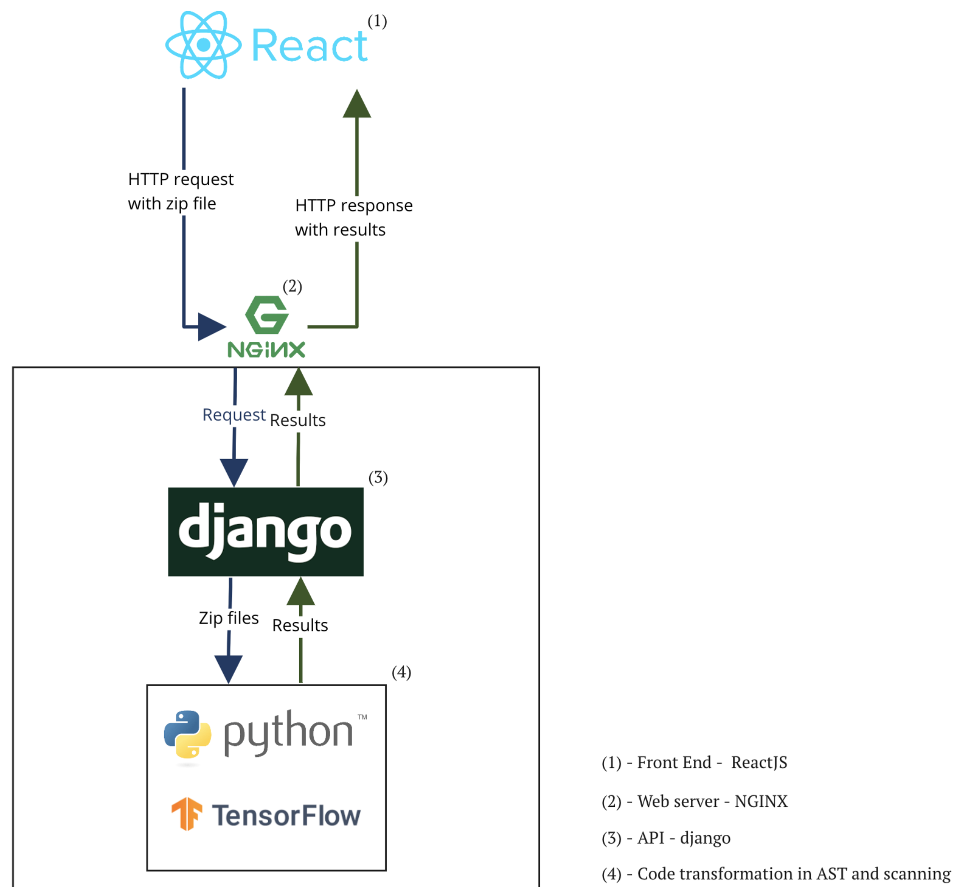


Figure 17: Schema explaining the interface web application.

7.2 FUNCTIONALITIES

In this section it will be presented the developed functionalities and the final visual result in order to prove that the mockups were indeed useful and took a key part in the fast and accomplished development.

The functionalities developed in this prototype interface were :

- To load and upload JAVA project files in zip format ;
- To consult the positive scan results for XSS and *injection*.
- To view the vulnerability code location in the files.

7.2.1 *Load and upload JAVA project files*

In the home page it is possible to introduce the desired name for the project, then to load the project files in zip format as seen in Figure 18. At this stage the "Analysis" and "Back" buttons are locked.

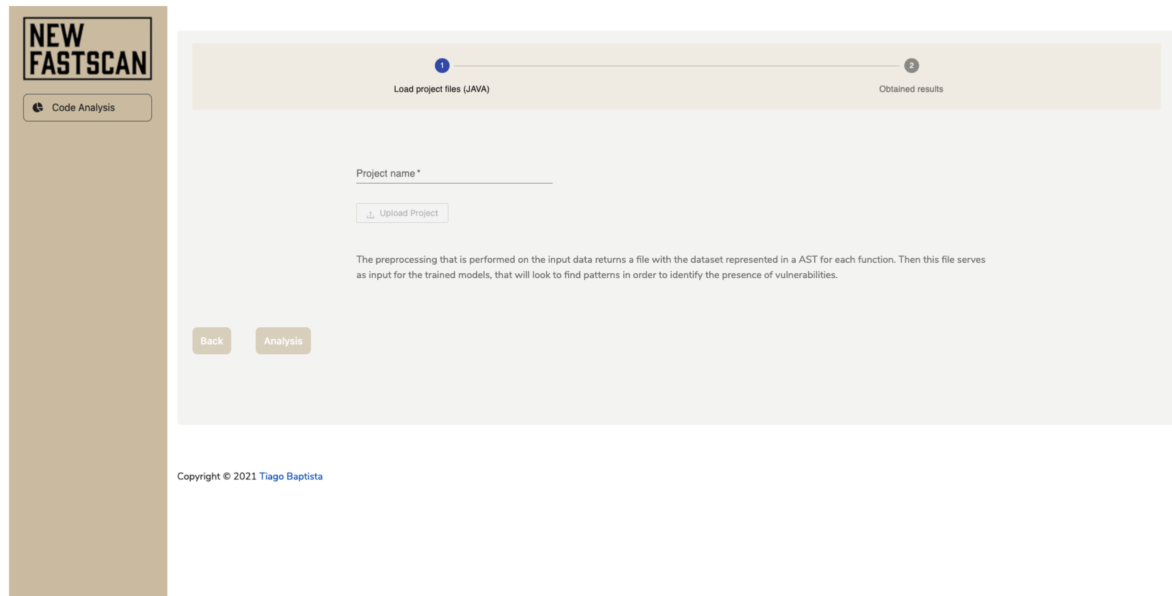


Figure 18: Final Interface homepage.

Once the project name and files are uploaded then the "Analysis" button is unlocked as seen in Figure 19.

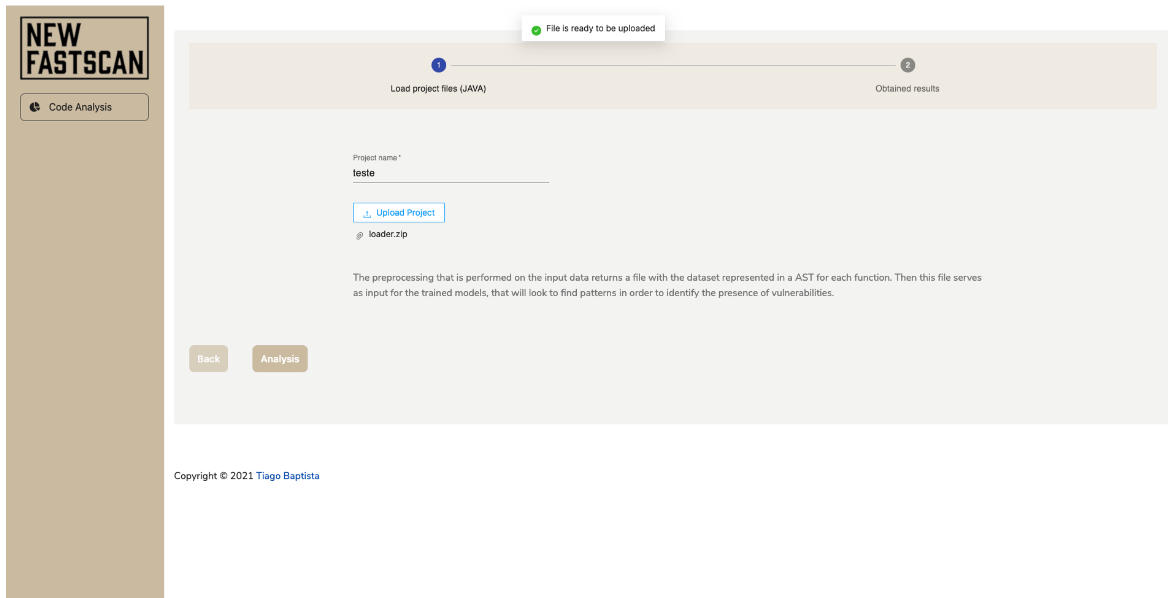


Figure 19: Final Interface load project.

Then it is possible to click "Analysis", when clicking this button the request is sent to the Back-end and the user must await until the scan is completed as seen in Figure 22.

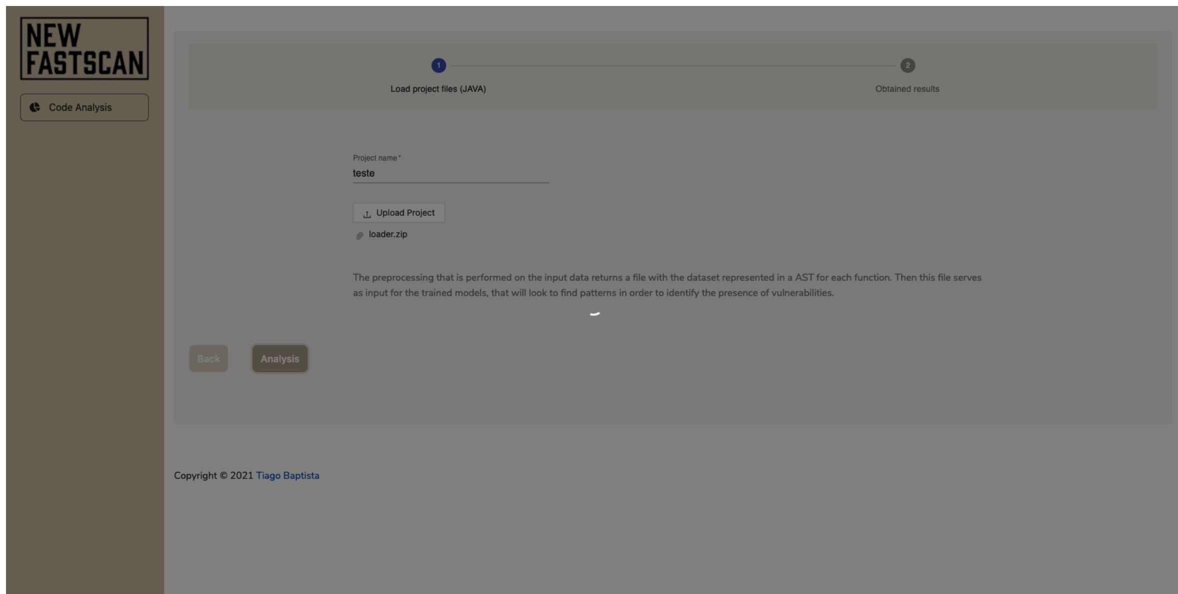


Figure 20: Final Interface scanning project.

7.2.2 *Scan results*

After following the steps presented in Section 7.2.1, it is possible to view the results in two ways. First in table (depicted in Figure 21) where it is possible to filter and order by all the fields, which are the function name, function location in the project files, the begin, end line and also the model's prediction.

Figure 21: Final Interface results table .

For each entry of the table it is possible to preview the file where the entry is located, with the highlight of the begin and end line as seen in Figure 22.

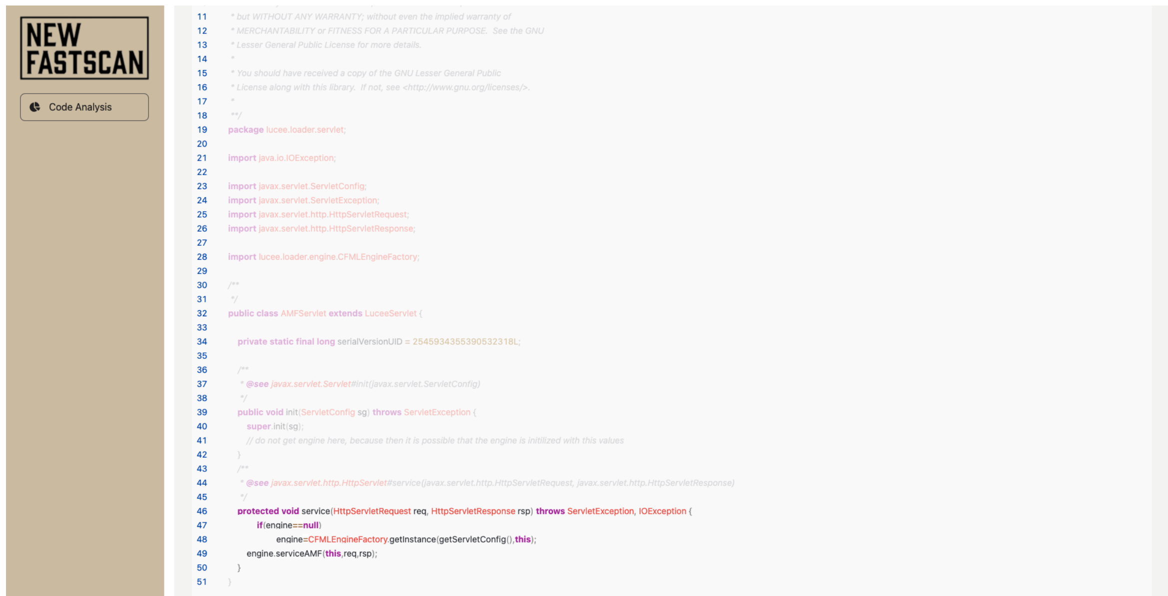


Figure 22: Final Interface code view.

8

CONCLUSION

This chapter is intended to close the master thesis, summarising the document, outcomes reached and a reflection over them.

The first chapter contains the motivation, objectives, research methodology and presents the document organization of the master thesis.

The second chapter contains some key background concepts that must be presented in order to follow the presented work.

The third chapter is a literature review on vulnerability detection. The outcomes of the reported stage provided the foundations for the proposed approach.

The fourth chapter presents and discusses the working proposal with a general system architecture presentation which was made on the early stages of the master thesis.

The fifth chapter explains the development and includes the presentation of the dataset used for training as well as describes the hardware details and stages of the different phases.

The sixth chapter analyzes the results of each phase and also the general test of both models.

The seventh chapter present in depth the work performed in the interface. Finally the eighth chapter presents the different use cases for the developed work as well as conclusions and future work.

8.1 DISCUSSION

Taking into account the results from the first phase, it becomes clear that the hyperparameter optimization has improved the results in the increase the precision and the other metrics. Also the train only for a specific vulnerability might as well have an influence since the train for a more strict purpose is more effective, namely in this case. While *Fastscan* attempts to predict the presence of many types of vulnerabilities, *new Fastscan* aims at creating models to predict a single type of vulnerability, gathering the parts into a global analyzer in a final system.

Furthermore it was confirmed that when code2seq models are trained to classify methods as vulnerable or not vulnerable, the Bidirectional Long Short-Term Memory (BiLSTM) neural networks (that is the technology behind code2seq project), can perform the task of detecting vulnerabilities in source code with good accuracy and performance.

Even with the use of powerful hardware is still a slow process to train, optimize the hyperparameter for the data and add new models to the *NewFastScan*. Also the lack of data for training might be a drawback. The same is applied to the prediction, on which the time performance is directly related to hardware computational power.

Regarding the second phase, the cross testing lead to the detection of overffiting mainly caused by the low volume of data. This is a hard problem to solve, since it is not easy to get a great volume of validated and relevant vulnerabilities entries but a solution will be explored in the next section.

Also, it is important to highlight the effort that was done in the adaptation and installation of algorithms and programs to run the system in parallel platform offered by the SEARCH cluster available in the Informatic Department of University of Minho to be possible to train, tune and test the models in acceptable time.

The original hypothesis to develop a specific model for each type of vulnerability and then join them into a system that allows to scan a project for these different vulnerabilities was achieved as explained in Section 6.2,.

The test performed in section 6.3 was an important step to prove that the concept initially presented, of being able to use this system to detect different vulnerabilities in a single project, is possible and was achieved even though the results accuracy still must be improved in order to make *NewFastScan* a viable alternative.

This thesis main contributions are the changes performed in the preprocessing and prevision stages, *XSS* and injection models as well as the architecture, developed in the second phase, that allows to use them together. Also the interface and the back-end that, relies on the work performed in the second phase, and allows a better interaction and result visualization of the models prediction. Also a paper was published with some of the intermediate results and conclusions from the presented Master Thesis (Baptista et al., 2021).

8.2 Possible Applications

In this section are approached the different applications envisioned for the developed work. Some of these applications for the *New FastScan* might be :

- **A stand-alone application :** After improving the proposed architecture in Chapter 7 in order to allow a better response in terms of volume and response time. It would be possible to make available a platform were users could scan and inspect their code in order to detect the presence of vulnerabilities of a certain type. This could be very

helpful when used by companies which do not usually use scanning tools or even as a first scan before applying a specific analysis tool or technique, in this way certain vulnerabilities might be detected and eliminated earlier;

- **Reduce the *SAST* input :** To integrate it in the earliest stage of a *SAST* pipeline. This would allow it to act as a filter, specially in large projects, turning millions of lines of code in less, only sending the positive results for further confirmation and analysis by the *SAST*. This would result in an improved performance of the *SAST*, since the smaller the input the faster it obtains the results.Applying this model as a scanner that verifies projects before it goes through a *SAST*, other tool or manual verification could represent a great improve in terms of spent time, processing and manual work since it could eliminate the projects without vulnerabilities from further scanning. Comparing to the traditional tools this approach requires less results verification and promises better accuracy;
- **Confirm the results from an analysis technique:** After an analysis is done, it could be usefull to run the same project in another scanning tool in order to confirm or even detect new vulnerabilities. So, the idea of integrating this tool with others in such way that there is the possibility to scan a project only looking for some specific functions or code portions. This would be something of great interest since it would be a "lightweight" step because the scan would be only done in a small set of code portions wich in normal cases would not be such a lengthy scan as scanning the entire project files ;
- **A plugin :** There are many different platforms were this could be implemented namely in *github* [1] *or in a* Integrated Development Environment (IDE). The idea would be to reuse and improve the developed *API* and create services as plugins that automatically scan the code in this platforms and alert the users for the possibility of the presence of vulnerabilities in their code. This could have a great impact since user would not have to upload their code to any tool and wait for results, it could be useful to act as a prevention by developers.

8.3 FUTURE WORK

During the duration of research and development of this Master Thesis other interesting ideas came up, that were not explored due to the context and the available time. Still it is important to refer and leave them registered for potential further investigation. Next, it presented some of those ideas and proposals :

1 https://github.com/

- To train more models in order to predict different vulnerabilities and then integrate them into *NewFastScan* ;
- To extend compatibility to other languages. For that it is necessary to retrain the models with examples from such languages and also create a preprocessing for such languages ;
- To improve the interface architecture in order to make the application usable in real scenarios, more efficient and secure ;
- To solve the overffiting detected on the late stages of development. For that different techniques might be applied such as early stopping, regularization or even change the model's architecture (this one might not be the best approach since the code2seq type model is proven to have good performance in this kind of prediction). Also more external data could help to solve the problem, for instance using different datasets [2];
- To create a mechanism, using the interface, where a user can review the predictions made by the model and then learn from those feedbacks in order to improve its predictions. This could also be used in order to correct the previously referred overffiting ;
- To evaluate the possible gains of using balanced datasets, since it is still a matter of debate when it comes to training models for vulnerability identification;
- To investigate if there is the possibility to reduce the computational cost with the mix of traditional and machine learning techniques.

2 https://tqrg.github.io/secbench/

BIBLIOGRAPHY

Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400*, 2018.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL): 1–29, 2019.

Tiago Baptista, Nuno Oliveira, and Pedro Rangel Henriques. Using Machine Learning for Vulnerability Detection and Classification. In Ricardo Queirós, Mário Pinto, Alberto Simões, Filipe Portela, and Maria João Pereira, editors, *10th Symposium on Languages, Applications and Technologies (SLATE 2021)*, volume 94 of *Open Access Series in Informatics (OASIcs)*, pages 14:1–14:14, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-202-0. doi: 10.4230/OASIcs.SLATE.2021.14. URL https://drops.dagstuhl.de/opus/volltexte/2021/14431.

Jón Arnar Briem, Jordi Smit, Hendrig Sellik, and Pavel Rapoport. Using distributed representation of code for bug detection. *arXiv preprint arXiv:1911.12863*, 2019.

E Burns and W Groove. Research method. *Ergonomics*, 32(3):237–248, 2014.

Philip K Chan and Richard P Lippmann. Machine learning for computer security. *Journal of Machine Learning Research*, 7(Dec):2669–2672, 2006.

Brian Chess and Gary McGraw. Static analysis for security. *IEEE security & privacy*, 2(6): 76–79, 2004.

Brian Chess and Jacob West. *Secure programming with static analysis*. Pearson Education, 2007.

Kenneth Ward Church. Word2vec. *Natural Language Engineering*, 23(1):155–162, 2017.

David Coimbra, Sofia Reis, Rui Abreu, Corina Păsăreanu, and Hakan Erdogmus. On using distributed representations of source code for the detection of c security vulnerabilities. *arXiv preprint arXiv:2106.01367*, 2021.

Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX security symposium*, volume 98, pages 63–78. San Antonio, TX, 1998.

Mark Dowd, John McDonald, and Justin Schuh. *The art of software security assessment: Identifying and preventing software vulnerabilities*. Pearson Education, 2006.

Michael D Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27, 2003.

Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. In *2015 IEEE international symposium on performance analysis of systems and software (ISPASS)*, pages 171–172. IEEE, 2015.

Samuel Gonçalves Ferreira. Vulnerabilities fast scan - tackling sast performance issues with machine learning. Master's thesis, University of Minho, 2019.

Konrad Gos and Wojciech Zabierowski. The comparison of microservice and monolithic architecture. In *2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH)*, pages 150–153. IEEE, 2020.

Alan R Hevner. A three cycle view of design science research. *Scandinavian journal of information systems*, 19(2):4, 2007.

Aram Hovsepyan, Riccardo Scandariato, Wouter Joosen, and James Walden. Software vulnerability prediction using text analysis techniques. In *Proceedings of the 4th international workshop on Security measurements and metrics*, pages 7–10, 2012.

H Jabbar and Rafiqul Zaman Khan. Methods to avoid over-fitting and under-fitting in supervised machine learning (comparative study). *Computer Science, Communication and Instrumentation Devices*, pages 163–172, 2015.

Michael I Jordan and Tom M Mitchell. Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245):255–260, 2015.

Jey Han Lau and Timothy Baldwin. An empirical evaluation of doc2vec with practical insights into document embedding generation. *arXiv preprint arXiv:1607.05368*, 2016.

Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*, 2018.

Rui Lopes, Diogo Vicente, and Nuno Silva. Static analysis tools, a practical approach for safety-critical software verification. *ESA Special Publication*, 669, 2009.

Rahma Mahmood and Qusay H Mahmoud. Evaluation of static analysis tools for finding vulnerabilities in java and c/c++ source code. *arXiv preprint arXiv:1805.09040*, 2018.

John McCarthy. What is artificial intelligence? 1998.

Lili Mou, Ge Li, Zhi Jin, Lu Zhang, and Tao Wang. Tbcnn: A tree-based convolutional neural network for programming language processing. *arXiv preprint arXiv:1409.5718*, 2014.

Ken Peffers, Tuure Tuunanen, Marcus A Rothenberger, and Samir Chatterjee. A design science research methodology for information systems research. *Journal of management information systems*, 24(3):45–77, 2007.

Mr John Peters, Keith Howard, and Mr John A Sharp. *The management of a student research project*. Gower Publishing, Ltd., 2012.

Marco Pistoia, Satish Chandra, Stephen J Fink, and Eran Yahav. A survey of static analysis methods for identifying security vulnerabilities in software systems. *IBM Systems Journal*, 46(2):265–288, 2007.

Blake Ross, Collin Jackson, Nick Miyake, Dan Boneh, and John C Mitchell. Stronger password authentication using browser extensions. In *USENIX Security Symposium*, pages 17–32. Baltimore, MD, USA, 2005.

R. W. Shirey. Internet security glossary, version 2. *RFC*, 4949:1–365, 2007a.

Robert W. Shirey. Internet security glossary, version 2. *RFC*, 4949:1–365, 2007b. doi: 10.17487/RFC4949. URL https://doi.org/10.17487/RFC4949.

Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. *arXiv preprint arXiv:1206.2944*, 2012.

Anshul Tanwar, Krishna Sundaresan, Parmesh Ashwath, Prasanna Ganesan, Sathish Kumar Chandrasekaran, and Sriram Ravi. Predicting vulnerability in large codebases with deep code representation. *arXiv preprint arXiv:2004.12783*, 2020.

Alan M Turing. Computing machinery and intelligence. In *Parsing the turing test*, pages 23–65. Springer, 2009.

Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *NDSS*, volume 2007, page 12, 2007.

REPRESENTING CODE USING VECTORS

This section aims to present a general view over Alon et al. (2019) approach on representing code as single fixed-length code vector. This representation is created in order to predict semantic properties of a code snippet.

There were other approaches that were groundbreaking in the area of neural networks for natural language processing and were also the base for *code2vec* and *code2vec*, namely *word2vec* and *doc2vec* (Church, 2017) (Lau and Baldwin, 2016).

code2vec is an attempt to create a representation of source code, using . The main goal was to represents the set of paths over the program and then uses this paths to produce each output token.

It is presented an overview over *code2vec* process to transform a code snippet into a vector. First, for each method it is constructed an *AST*. Then the paths from the *Abstract Syntax Tree* (*AST*) are extracted, having the sequence of nodes, the direction of their links in order to keep the path context. Finally each path and leaf extracted previously is transformed into a vector representation and then it is all concatenated into a single vector called path context as shown below.

```
get|network|topology|not|null,Nm0

|MarkerExpr|Mth|Cls1,string

not|null,Nm0|MarkerExpr|Mth|Nm2,METHOD_NAME

string,Cls1|Mth|Nm2,METHOD_NAME

string,Cls1|Mth|Bk|Ret|Nm0,network|topology

METHOD_NAME,Nm2|Mth|Bk|Ret|Nm0,network|topology
```