AN EMPIRICAL ASSESSMENT OF THE CONTRIBUTING FACTORS FOR VULNERABILITY DETECTION USING MACHINE LEARNING

Esma Mouine

A thesis
in
The Department
of
Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Applied Science (Electrical and Computer Engineering)
Concordia University
Montréal, Québec, Canada

December 2021

Concordia University
School of Graduate Studies

This is to certify that the thesis prepared

By: **Esma Mouine**

Entitled: **An Empirical Assessment of the Contributing Factors for Vulnerability Detection using Machine Learning**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Electrical and Computer Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair
Dr. A. Hamou-Lhadj

_____ Examiner
Dr. J. Clark (CIISE)

_____ Examiner
Dr. A. Hamou-Lhadj

_____ Supervisor
Dr. Y. Liu

Approved by _____
Dr. Yousef R. Shayan,
Chair of Department or Graduate Program Director

_____
Dr. Mourad Debbabi,
Dean of Gina Cody School of Engineering and Computer Science

Abstract

An Empirical Assessment of the Contributing Factors for Vulnerability Detection using Machine Learning

Esma Mouine

There is an increasing trend to mine vulnerabilities from software repositories and use machine learning techniques to detect software vulnerabilities automatically. A fundamental but unresolved research question is: how do different factors in the mining and learning process impact the accuracy of identifying vulnerabilities in software projects of varying characteristics? Substantial research has been dedicated in this area, including source code static analysis, software repository mining, and NLP-based machine learning. However, practitioners lack experience regarding the key factors for building a baseline model of the state-of-the-art. In addition, their lack of experience regarding how transferable the vulnerability signatures from a project to another are. This study investigates how the combination of different vulnerability features and three representative machine learning models impact vulnerability detection accuracy in 17 real-world projects. This thesis proposes different machine learning methods to detect software vulnerabilities. The first part of this work consists of establishing a baseline model for vulnerability prediction using NLP. For that, two types of vulnerability representations are examined: 1) code features extracted through NLP with varying tokenization strategies and three different embedding techniques (bag-of-words, word2vec, and fastText) and 2) a set of eight architectural metrics that capture the abstract design of the software systems. The four machine learning algorithms include a random forest model, a support vector machine model, and a residual neural network model. The second part of the study is an effort to evaluate the baseline model sufficiently and fairly by using it to evaluate the performance of another model. More experiments are performed using a bidirectional long short-term memory (BiLSTM) combined with word2vec. The results are compared to the baseline results.

Overall, the first set of experiments, the models returned the following results. 95% of the learning metrics (precision, recall, f1 score, etc.) are above 0.77 in the experiments out of 10 hypothesis tests and 408 experiments. Further analysis shows a recommended baseline model with signatures extracted through bag-of-words embedding, combined with the random forest, consistently increases the detection accuracy by about 4% compared to other combinations in all 17 projects. The observations also show the limitation of transferring vulnerability signatures across domains based on the experiments. Furthermore, the baseline model is shown to perform better than the BiLSTM model.

Acknowledgments

I would like to thank all the people who contributed in some way to the work described in this thesis. First and foremost, I would like to express my sincere gratitude to my advisor Prof. Yan Liu for the continuous support, guidance, motivation, and immense knowledge and financial support during my Master's studies and research. I would also like to thank all people that contributed as co-authors of my paper. Finally, I would like to pay earnest gratitude to my family and friends for their love, support, and encouragement.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The National Institute of Standards and Technology (NIST) defines security vulnerability as a *weakness in an information system, system security procedures, internal controls, or implementation that could be exploited or triggered by a threat source* [51]. Software source code usually contains multiple vulnerabilities and errors, including syntax, semantic, communication, calculation, and logic errors. A single error is often enough to cause software failure. In software engineering, static code analysis helps to identify these bugs and flaws in the source code.

1.1 Problem statement

As software scales expand, vulnerability detection with sufficient accuracy and efficiency remains a challenge from both research [19, 37, 22, 66] and industrial perspectives [61, 78].

Currently, there is a wide variety of analysis tools that attempt to uncover common vulnerabilities in software. In software engineering, static code analysis helps to identify bugs or flaws in the source code. Code analysis techniques are embedded in security scanners and raise alerts when vulnerabilities are detected [35, 57, 2, 47, 73, 15, 30]. However, softwares have significantly increased in both size and complexity. Identifying security vulnerabilities in code is highly difficult since they are rare compared to other types of software defects. While these tools exist for program analysis, they typically only detect a limited subset of possible errors based on predefined rules. One case study [44] performed using a static analysis tool on

Java source files showed that 45.7% of discovered vulnerabilities were false positives.

Beyond these traditional tools, research has been conducted to build a feature engineering methodology that improves the precision and recall of detecting vulnerabilities. With the availability of large amounts of open-source code repositories, it has become possible to use data-driven techniques to learn the patterns of software vulnerabilities directly from mined data. Clang [74] is a static analyzer that uses a memory modelling method for static analysis of C programs. Basili et al. [9] used source code metrics to classify the C++ code as vulnerable or not vulnerable in 1996. Nagappan et al. [45] utilize complexity metrics on some Microsoft software systems to identify faulty components. The work of Perl et al. [56] classifies if commits are related to a CVE or not using machine learning.

An emerging approach is treating software code as a form of text and leveraging Natural Language Processing (NLP) techniques to extract features automatically. Dam et al. [19] used a Long Short Term Memory (LSTM) model to capture the relationships between code elements. Likewise, Russel et al. [61] developed a fast and scalable vulnerability detection tool for C and C++ based on deep feature representation learning that interprets source code. Hovsepyan et al. [29] analyzed Java source code using bag-of-words and support vector machines to classify vulnerabilities.

To test the learning of a model, a baseline model is needed. This work wants to test the learning of the vulnerability signatures and the ability of a model to classify a vulnerable file according to some factors. Such a baseline helps to establish a base to investigate techniques on feature representation, learning models, factors such as code structure, and complexity in learning vulnerability patterns. A baseline model is commonly used as in the artificial intelligence community [72, 25]. It serves as a reference point to compare the performances of other models that are usually more complex. It also relies on understanding the key factors contributing to discovering vulnerability signatures through a combination of techniques and machine learning models.

This thesis evaluates the different factors that can identify the vulnerability signatures in the source code. It investigates how the combination of different features from source code and different representative machine learning models can impact vulnerability detection accuracy in 17 real-world projects. The ultimate goal is to

develop a learning method that takes input as code embeddings from project repositories so that the learning is transferable to other projects. Hence, this thesis answers the following research question : ***What are the contributing factors in learning processes that impact the accuracy of identifying vulnerabilities across software projects?***

1.2 Contribution

In this work, a corpus composed of tokens from software repositories is created. These tokens are then used to learn vulnerability patterns. The source code tokens are embedded as numerical features for learning vulnerability classification. First, this work focuses on four aspects of the learning process: (A1) the **tokenization** of the source code, (A2) **the generation of embeddings**, (A3) **architectural metrics** and (A4) **machine learning models**.

For **tokenization** two tokenization approaches are considered —with and without symbols and comments. For **embeddings**, three types of embedding methods are investigated (namely bag-of-words [76]; word2vec [41], and fastText [34]). For **architectural metrics**, eight file-based metrics are considered to augment vulnerability representations. They measure how source files are connected to each other in a system. This is for the purpose of examining whether adding architectural metrics helps to improve detection accuracy. For **machine learning models**, three machine learning algorithms are considered, including a weak learner-based model (random forest [70]), a kernel vector-based model (support vector machines [17]), and a neural network model (residual neural network [27]).

First, this thesis evaluates the combined effects of the above four aspects on the accuracy of vulnerability classification over 17 Java projects. The results of 408 experiments are compared using a statistical test. The combination that returns the best results allows establishing a baseline model. The baseline model is further evaluated for its transferability through cross-validation and cross-domain experiments. And finally, the baseline model is used to evaluate another model more complex. Since LSTM has been dominating most NLP tasks in the last few years, achieving the state of the art results. Further experiments are performed with a BiLSTM model, which results are compared to the baseline.

1.3 Outline

The thesis is structured as follows. Chapter 2 gives background information about software vulnerabilities, natural language processing, architecture metrics and their relation to machine learning. Chapter 3 presents the studies and work previously conducted related to this thesis. Chapter 4 describes the research methodology. It starts by exposing the problem statement through 5 research questions, then discusses the detection process and the different aspects of each research question that can identify the vulnerability signatures and the different machine learning models used. Chapter 5 is a review of the datasets and the evaluation metrics used to evaluate the experiments. Chapter 6 includes an analysis of the results related to a first set of experiments that end by identifying the baseline model. Chapter 7 outlines further experiments performed to evaluate a BiLSTM model using the previously defined baseline model. Chapter 8 highlights some of the limitations in the threats of validity. Finally, Chapter 9 concludes and summarizes this thesis.

Chapter 2

Background

This chapter introduces essential terms of the research, such as the definition of software vulnerabilities and architecture metrics and their relation with code analysis and how natural language processing can be used in source code analysis.

2.1 Software vulnerabilities and Code Analysis

According to NIST [51] a software vulnerability is *A security flaw, glitch, or weakness found in software code that could be exploited by an attacker (threat source).* These flaws can impact the performance and security of the software. They can allow untrustworthy attackers to gain access to private data. That is why software vulnerabilities must be identified and prevented.

Despite academic and industrial efforts in improving software quality, vulnerabilities remain a big problem. Multiple vulnerabilities are reported every year in the Common Vulnerabilities and Exposures (CVE) database. This database is used to collect and share publicly disclosed information about security vulnerabilities. Likewise, Common Weakness Enumeration (CWE) is a community-developed list of common software and hardware security weaknesses [16]. The Open Web Application Security Project (OWASP) Benchmark is a Java test suite that contains thousands of exploitable test cases where each one maps to a specific CWE. NIST's Software Assurance Reference Dataset (SARD) [50] provides a set of known security flaws for researchers and software security assurance developers.

Within SARD, a set of test suites exist, including the Juliet Tests for Java and

C++ [49], mobile apps and Web apps [48]. These sources of information can be used to search for known vulnerabilities to identify potential exploits as part of a forensics process.

Over the last decade, researchers in software engineering have developed many source code analysis tools and techniques for handling and identifying bugs or vulnerabilities in software. These automated tools scan the code looking for potential flaws and raise alerts when vulnerabilities are detected [35, 57, 2, 47, 73, 15, 30]. These tools are called security scanners. Some are static analyzers that have the most impact when used early in the development process. These scanners scan the code before it is compiled. They can help developers identify vulnerabilities in the initial stages of development. Other tools perform dynamic analysis by exercising the application and testing it for weak spots through the user interface. They take the approach of a real attacker from the outside.

While they are undoubtedly necessary as part of any security program and despite their abundance, no tool can accurately find all weaknesses, so they typically only detect a limited subset of possible errors. Most of the static code analysis scanners are based on predefined rules. These rule-based techniques of pattern matching [73, 15, 30] are usually defined by security experts and consist in enumerating known vulnerabilities. The vulnerabilities identified by the scanner have to be confirmed by security engineers. However, these tools have shown to be limited in effectiveness. One of the limitations is the high false-positive rate [44]. For example, one case study [44] performed using a static analysis tool on Java source files showed that 45.7% of discovered vulnerabilities were false positives.

With the spread of open-source repositories and databases like CVE, it has become possible to use machine learning techniques to discover vulnerability patterns. For improvement, Clang [74] is a static analyzer that uses a memory modelling method for static analysis of C programs. In recent years, many works have used machine learning for program analysis [22]. Using machine learning to build a vulnerability prediction model, several features that represent the software are selected. The most frequent features are software metrics and developers' activity. Basili et al. [9] used source code metrics to classify the C++ code as vulnerable or not vulnerable in 1996. Nagappan et al. [45] utilize complexity metrics on some Microsoft software systems to identify faulty components. The work of Perl et al. [56] classifies if commits are

related to a CVE or not using machine learning. They combine code metrics analysis with metadata gathered from code repositories.

This work uses different machine learning techniques for classifying vulnerabilities in the source code while minimizing the false positive rate.

2.2 Natural Language Processing and Source Code

Natural language processing (NLP) refers to the automatic computational processing of human languages. Machines can use NLP to extract information from natural languages. The last decade has witnessed significant progress in deep learning in NLP applications.

Natural language is known to be repetitive and predictable, and that is the same for software code. Programming languages, in theory, are complex, flexible and powerful. However, a lot of the programs that people write are simple and somewhat repetitive. Thus they have usefully predictable statistical properties that can be captured in statistical language models and leveraged for software engineering tasks. In addition to that, the growing availability of open-source repositories creates new sources of data, thus, new opportunities for using machine learning to process source code en masse.

Methods based on NLP treat the source code as a form of text to extract features have emerged. As mentioned before, software repositories contain a large amount of source code that can form a *corpus*. The concept of a corpus, originating in linguistics, is a collection of text in one or multiple languages. Therefore, a repository is considered a corpus that contains the source code that is considered a form of text upon which feature representations can be learned. In NLP, the corpus is used to train learning models. For example, in the classic Word2Vec [41] model, a corpus is used to produce the embeddings. The tokens' contexts are learned from a corpus—the context forms the relations between the tokens in a multidimensional space.

Many research have emerged that use NLP on source code [6, 77, 19, 29, 56]. Alon et al. [6] leverages the syntactic structure of programming languages to summarize source code using Abstract Syntax Trees (AST). Zhou and Sharma [77] use commit messages and bug reports from repositories to identify software flaws. Dam et al. [19] used a Long Short Term Memory (LSTM) model to capture the relationships between

code elements. Likewise, Russel et al. [61] developed a fast and scalable vulnerability detection tool for C and C++ based on deep feature representation learning that interprets source code. Hovsepyan et al. [29] analyzed Java source code using bag-of-words and support vector machines to classify vulnerabilities. Other recent research has focused on machine learning models to mine feature representations from software repositories [56].

This work uses different NLP-based methods for feature extraction for vulnerability classification.

2.3 Architecture metrics and Source Code Analysis

Software metrics can also be used for source code analysis. Using metrics can help to see how the system is growing and expanding. According to the *IEEE Standard for a Software Quality Metrics Methodology*, [1] software metric is defined as *A function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which software possesses a given attribute that affects its quality.*

Having tools to analyze software code according to software metrics is essential to characterize the design of things that are being created, evaluate their ideas, and detect aspects of quality in order to keep it as maintainable and extensible as possible.

Software architectures are the high-level abstract of a software system. They consist of the body of a system. They include all the components and how they interact with each other, and the environment in which they operate. Software architectures are the most important determinant to systematically achieve quality attributes in a software system, including software security [13]. Poor software architectural decisions are responsible for various software quality problems, including security flaws and vulnerabilities. Software security is, for many systems, the most essential quality attribute driving the design.

Numerous previous research has underscored the impact of software architecture on security. Due to the intrinsic connections between software architecture and security, prior studies have also focused on how architecture impacts the security of a system [20, 62, 67, 5] finding an intrinsic connection between software architecture

and security. Metrics can be measured to capture the complexity of software architecture entities [20, 62, 67, 5, 65, 43]. These different architecture metrics can be used as a representation of the source code for machine learning models to detect software vulnerabilities.

Using metrics helps to evaluate how the system is growing with higher complexity and could give a hint of where maintainability will be chaos due to package relationships and can cause software flaws.

Chapter 3

Related Work

Different techniques have been used for vulnerability detection in source code, from using predefined rules to different machine learning techniques. This chapter presents other works that attempt to detect software vulnerabilities using machine learning, NLP techniques and software architectures.

3.1 Static Vulnerability Detection

The constantly increasing number of security vulnerabilities have become an essential concern in the software industry and the field of cybersecurity. The two mainly approaches are static scanners that are mainly rule-based [73, 15, 21] and ML-based [19, 59, 9, 45] detection systems.

Rule-based tools are based on pattern matching using rules predefined by security experts. Each vulnerability type has its own rule, and the violation of the predefined rule raises an alert and reports the location and the type of vulnerability. For example, some open source tools such as Flawfinder [73], RATS [30], ITS4 [71], and other commercial tools like Checkmarx [15] generate their patterns from source code. However, these tools cannot always accurately distinguish between various vulnerable codes, resulting in high false positives or high false negatives [37].

Several approaches have been developed aiming to improve the detection of vulnerabilities (e.g., [75, 19, 29, 54, 42]). One example is applying pattern recognition techniques to detect malware [46]. This technique [46] consists of visualizing malware binary gray-scale images and classifying these images according to observations that

show that malware from the same families appears to be very similar in layout and texture.

There have been a wide array of ML-based vulnerability prediction research [19, 59, 9, 45, 56, 37]. The most frequent features used in previous works are software metrics [59, 9, 45] and developers activity [66]. Basili et al.[9] used source code metrics to classify the C++ code into binary code vulnerabilities back in 1996. Nagappan et al. [45] used complexity metrics like module metrics that consist of the number of classes, functions and variables in the module M, in addition to per-function and per-class metrics. They used those metrics with some Microsoft systems to identify faulty components.

Perl et al. [56] considered metrics from developer activities by analyzing if commits were related to a vulnerability or not. The methodology of this work [56] consists of combining machine learning using a support vector machine (SVM) classifier with code metrics gathered from repository metadata. Li et al.[37] used multi-class SVM to detect a different class of vulnerabilities.

Russell et al. [61] proposed a large-scale function-level vulnerability detection system to learn deep feature representation of source code after lexical analysis. It combined the neural feature representations of function source code with a random forest as a classifier. Harer et al. [26] used machine learning methods to perform the data-driven vulnerability detection and compared the effectiveness of using the source code and the compiled code.

Other machine learning models provide alternative solutions for automating the vulnerability detection task. This method is potentially more efficient in vulnerability discovery [18, 23, 39, 69].

More specific feature representations using natural language processing and architecture metrics from previous work are described in the following sections.

3.2 Machine Learning and Natural Language Processing Detecting Vulnerabilities

This research aims to identify the factors contributing to the learning of software vulnerabilities from source code repositories using some NLP techniques. Machines can use NLP to extract information from natural languages. Considering source code

as a form of language, NLP techniques can be used to analyze source code. In addition to the large number of public repositories. It is easier now to look for vulnerability patterns in the code. Many methods that treat code as a form of text and use natural language processing-based methods for code analysis have emerged.

Zhou and Sharma [77] used commit messages and bug reports from repositories to identify software flaws using NLP techniques such as word2vec to create the embeddings used as features and machine learning classifiers. Hovsepyan et al.[29] analyzed Java source code from Android applications using a bag-of-words representation and SVM for vulnerability prediction.

Pang et al. [54] further include n-grams in the feature vectors and used SVM for classification. Jackson and Bennett [31] using the Python Natural Language Toolkit (NLTK) to develop a machine learning agent that uses NLP techniques to convert the code to a matrix and identify a specific flaw—SQL injection—in Java byte code using decision trees and random forests for classification.

Other works focus more on using deep learning techniques. Russel et al. [61] attempt to identify vulnerabilities using C and C++ source code at the function level based on deep feature representation learning that directly interprets lexed source code. Dam et al.[19] present an approach based on deep learning using an LSTM model to learn both semantic and syntactic features of code automatically.

Apart from the work of Hovsepyan et al. [29] most of these approaches focus on the feature engineering part like Russel et al. [61] that uses a convolutional neural network to build the feature vectors.

A recent survey [38] summarizes the techniques, datasets and results obtained from vulnerability detection research that uses machine learning. According to their categories, this work falls in the text-based category since a convolutional neural network (ResNet) is used.

This thesis focuses on detecting vulnerabilities in source code using machine learning and natural language processing techniques. However, general NLP-based techniques (bag-of-word, word2vec, fastText, and tokenizing code) are used associated with different machine learning models to identify the key factors contributing to the learning of the software flaws from code.

3.3 Software Architecture and Security

Software architecture is the high-level abstract of a software system. Poor software architectural decisions are responsible for various software quality problems. Numerous previous research has underscored the impact of software architecture on security.

Software architecture is the most important determinant to systematically achieve quality attributes in a software system, including software security [13]. Software security is, for many systems, the most essential quality attribute driving the design.

Due to the intrinsic connections between software architecture and security, prior studies have investigated how software architecture impacts the security of a system [20, 62, 67, 5]. However, little work has investigated leveraging software architecture characteristics and metrics in machine learning processes to discover vulnerabilities. Researchers in software architecture have developed some measures to capture the complexity of software architecture entities [20, 62, 67, 5, 65, 43]. For example, *Fan-In* and *Fan-Out* of source files and classes are shown to impact the propagation of software quality issues through the inter-dependencies among software entities [65]. What remains unclear is whether and how different architecture metrics can be used as vulnerability representations for machine learning models to detect software vulnerabilities.

Previous research mainly focused on security assessment and evaluation from an architectural perspective. For example, Feng et al. found that software vulnerabilities are highly correlated with flawed architectural connections among source files [20]. Sohr and Berger found that software architecture analysis helps to concentrate on security-critical software modules and detect certain security flaws at the architectural level, such as the circumvention of APIs or incomplete enforcement of access control [68]. Brian and Issarny showed how software architecture benefits security by encapsulating security-related requirements at design-time [11]. Antonino et al. [52] evaluated the security of existing service-oriented systems on the architectural level. Their method is based on recovering security-relevant facts about the system and interactive security analysis at the structural level. Alkussayer and Allen [4] proposed a security risk evaluation approach by leveraging the architectural model of a system, assuming that components propagate their security risks to higher-level components in the architecture model. Alkussayer and Allen [3] assessed the level of security supported by a given architecture and qualitatively compared multiple architectures

with respect to their security support.

Despite the high recognition of an architecture's impact on security, the is little focus on using architectural metrics as vulnerability signatures for machine learning models [40, 32]. Alshammari et al. [7] is one of the few studies that investigated security metrics based on the composition, coupling, extensibility, inheritance, and design size of an object-oriented project. However, these metrics have not been compared with other vulnerability signatures, such as code features extracted using NLP. In addition, these metrics tightly tie into object-oriented concepts and may not be easy to transfer to other programming paradigms.

Motivated by the work of Feng et al., this study focuses on *eight* architectural metrics that capture how software elements, i.e. source files, are interdependent on each other [20]. Moreover, these metrics are generally applicable to software projects of different characteristics, such as the programming language used. In addition, although they are measured at the file level in this work, it is easy to roll up and down to the component level or method level following the same rationale to detect vulnerabilities at different granularities in future studies. Most importantly, this work is the first to compare architectural metrics with code features extracted through NLP as vulnerability representations to the best of our knowledge.

Chapter 4

Research Methodology

The research method considers the learning task as a classification problem to the vulnerability signature. This research considers four relevant aspects of the learning process, including:

(A1) Tokenization: Regarding how tokens are extracted from software, it allows to evaluate if code comments and symbols as tokens impact the detection results;

(A2) Embedding: Tokens are transformed into numerical values. The effects of different embedding techniques are investigated;

(A3) Architectural metrics: This aspect focuses on architectural metrics that measure the complexity of the inter-dependencies among fine-grained software architecture elements at the file level. Eight architecture metrics are considered, which will be detailed later;

(A4) Machine learning algorithm for classification: three different models are considered.

The software is considered as a corpus to develop the feature representation through token encoding. The tokens are the terms from the software code separated according to the spaces and special characters. The corpus is formed of software code from open repositories. Then the encodings are embedded in machine learning models for vulnerability detection on datasets such as OWASP benchmark, Juliet test suite for Java, and Android Study. The architectural metrics are used as additional feature representations, along with code-based representations. Based on the above rationale, the following research questions are answered:

RQ1: ***How does the filtering of tokens affect source code vulnerability detection?***

When using NLP techniques to extract features, an essential preprocessing step is the tokenization of the source code. This step involves separating the code into tokens before creating the embeddings. Generally, special symbols (including , . ; : [ ]) (+ - = — & ! ? * ˆ \¡ ¿ @ ” ’ # %) should be filtered out from the source code before separating it into tokens. Another question is: do the comments contain meaningful features and affect the features representations? To answer this question, the vulnerable files are used with all the tokens and also after removing the symbols and the comments, the classification results are compared.

RQ2: ***Does a specific embedding technique perform better across software projects?***

Embeddings are the process that maps each token to one vector, and the vector values are learned using a class of techniques such as bag-of-words [76], word2vec [41] and fastText [12]. This research question evaluates whether a particular embedding technique constantly improves the performance of vulnerability detection across all 17 software projects.

RQ3: ***Can architectural metrics that measure the structural complexity of software improve vulnerability detection?***

This question is answered in two ways. First, by comparing the learning performance separately using the NLP-based token embedding and using the architectural metric representation, respectively. Next, these representations are merged into the learning process to observe if the combination improves vulnerability detection compared to using either of them alone.

RQ4: ***Which machine learning model performs better across different projects?***

The three kinds of machine learning models, namely, decision tree-based (Random Forests), kernel-based (Support Vector Machines), and deep neural networks (Residual Neural Networks), are compared. Each model is combined with the feature representation extracted through different tokenization techniques, embedding techniques, and architectural metrics. The goal is to discover whether a particular machine learning model performs best in terms of vulnerability detection in different settings and across software projects.

RQ5: ***How transferable is the learning in predicting vulnerabilities of projects in cross-validation?***

For this last research question, which aims to evaluate the transferability of the learned features in vulnerability prediction, the learning model is fine-tuned by training projects in cross-validation. Different sets of experiments are defined where the models are trained with a project and predict the vulnerabilities of other projects.

4.1 Vulnerability Detection Process

The process of vulnerable code detection, as shown in Figure 1, contains a software repository, which provides the corpus for developing a vocabulary. Any project (even without vulnerable code labels) can be used for this purpose. Such a vocabulary is used to build the embedding of software tokens. The vocabulary is created by pre-training word2Vec and fastText with the corpus. The tokens are then converted to numerical representations by running the embedding. In addition, architecture metrics can be extracted from projects with tags. Next, architectural metrics and embedding of code tokens are the features used as input to a supervised classification model. The vulnerability detection is considered under two sources of vulnerability code labels:

(1) The labels are from code within the same project and domain as the target software for vulnerability detection (Tables 7, 8 and 9);

(2) The models are trained with software code in one project and domain with vulnerability labels and used to classify software code in a different project and domain. For example, a model is learnt with the dataset from the Juliet dataset, then used to predict the vulnerability of source code in Android projects (Tables 12, 11 and 13).

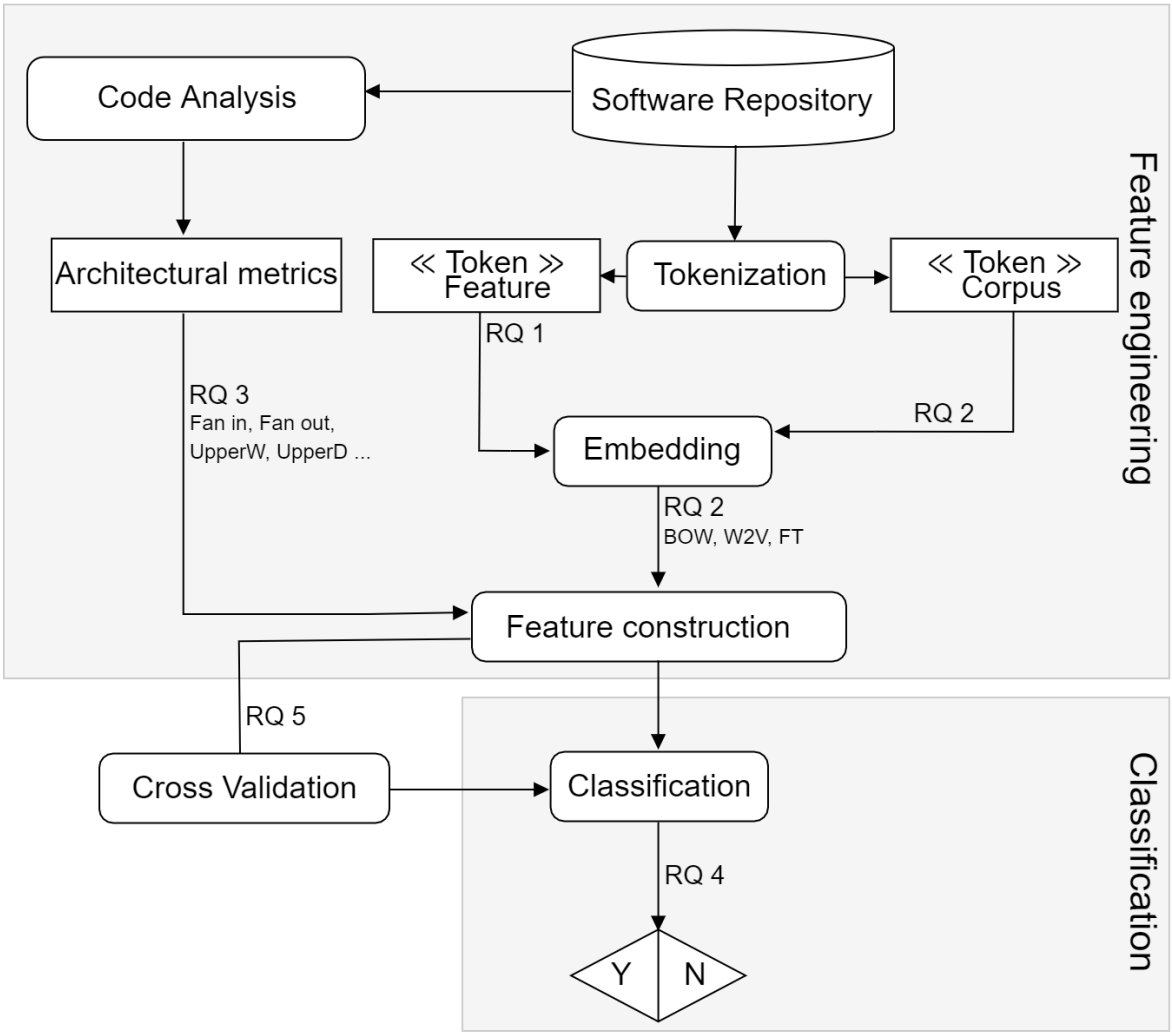

Figure 1: The process of learning software vulnerability as a classification task

4.2 Tokenization

Tokenization is a common pre-processing step in natural language processing to transform the raw input text into a format that is more easily processed. The raw code contains 1) special symbols include punctuation characters (such as, . , : ; ?) ( ] [’ ” } {), 2) mathematical and logical operators (such as, + - / = * & ! % — ¡ ¿); and 3) others (such as # \@ ˆ), in NLP special characters add no value to text-understanding and can induce noise in algorithms. In addition to other meta text that usually appears as code comments. These comments are any text that starts with two forward slashes (//) and any text between /* and */. To determine if those special characters and comments are essential in the vulnerability prediction in the source code, regular expressions are used to remove the code comments and special symbols. The Figures 2 and 3 shows an example of the two tokenization techniques.

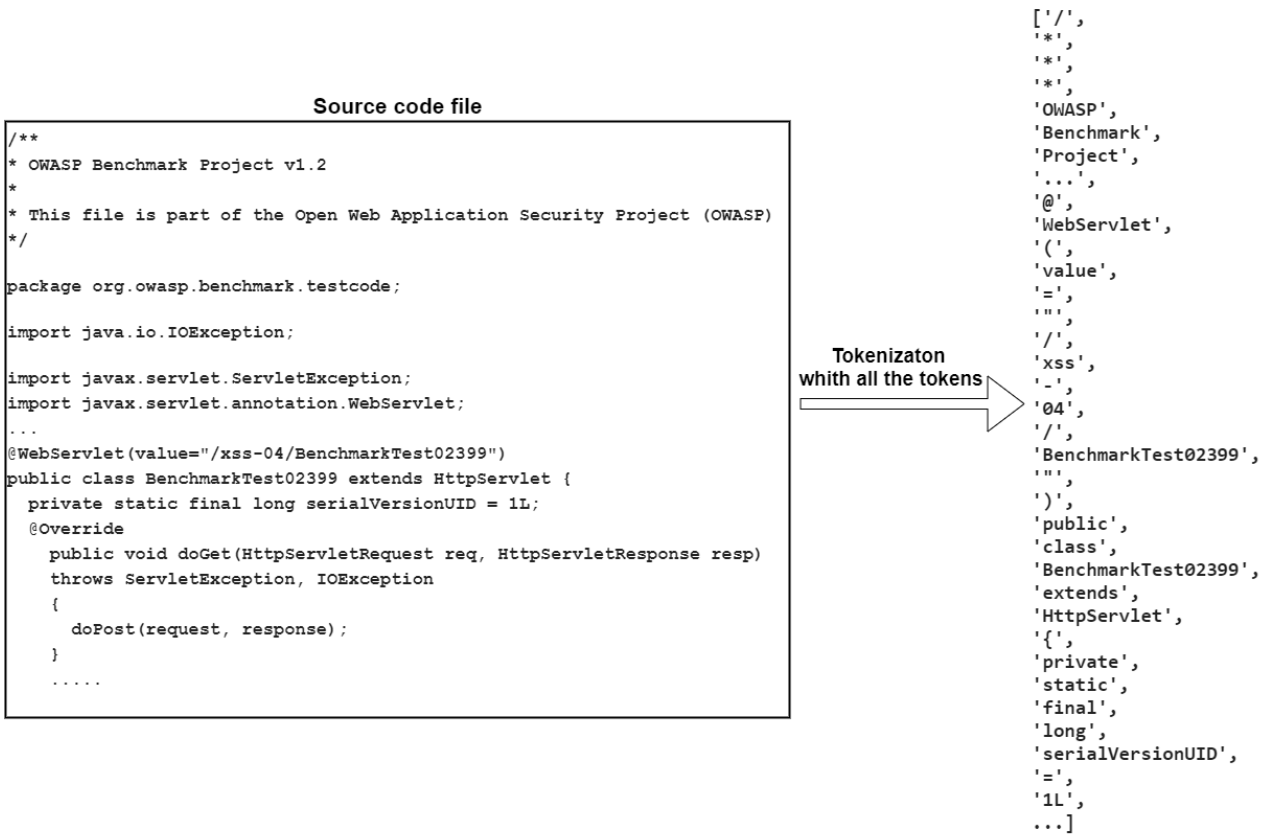

Figure 2: Example of a file from the OWASP project after tokenization with keeping all the tokens.

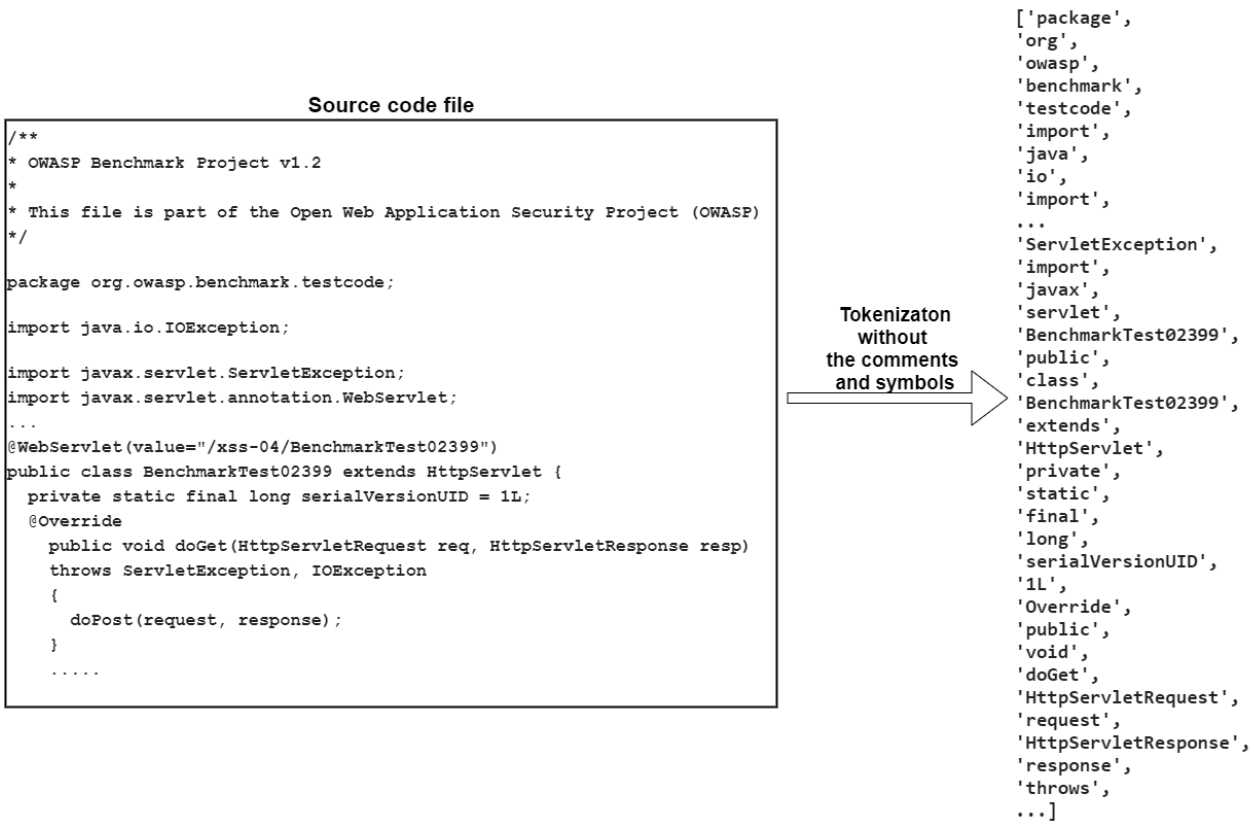

Figure 3: Example of a file from the OWASP project after the prepossessing (that consists on removing the comments and the symbols) and the tokenization.

4.3 Token Embeddings

Token embeddings are learned numerical representations for text where the same value approximates tokens or words with similar meanings. In the domain of NLP, a corpus is a collection of texts. All tokens or words in the multiple corpora form a high-dimensional space. The learning model calibrates the positions of each word or token according to its relations with all other tokens. Finally, each token has a numerical vector representation called an embedding.

In this work, the corpus is formed with software projects selected from the Github repositories. Three models are trained as follows to create the numerical vector representation of the source code tokens:

4.3.1 Bag-of-words

Bag-of-words (BOW) is a representation of the text [76]. It represents the text as a vector where each element is an index of a token from the vocabulary. Each token is associated with its frequency in the text. Hence, the resulting vector has the same length as the number of unique tokens. The BOW vectors are limited to the size of the text that is used for the training.

4.3.2 Word2vec

Word2vec (W2V) is a method to create word embeddings that have been around since 2003 [41]. The algorithm uses a neural network associated with a large corpus of text. Word2vec can use skip-gram or CBOW to learn the representations of tokens. Skip-gram aims to predict the context of a word given its surrounding words. Given a context, CBOW is the same as BOW, but instead of using sparse vectors (a vector with a lot of 0) to represent words, it uses dense vectors. CBOW predicts the probability of a target word. This work uses Skip-gram since it is trying to maintain the context of the token. The model takes a target term and creates a numerical vector from the surrounding terms.

4.3.3 FastText

FastText (FT) is a library for learning word embeddings and text classification created by Facebook's AI research lab [34]. Akin to word2vec, fastText supports CBOW and skip-gram. Instead of feeding individual tokens into the neural network, fastText exploits the subterms information, which means each token is represented as a bag of characters in addition to the token itself. This allows the handling of unknown tokens, which aids cases where the internal structure of the words is considered and the unseen words handled.

Word2vec and fastText use the same parameters. The dimensionality of the feature vectors equal 300, and a window size of 5 and words with a total frequency lower than two are ignored. To obtain the source code embeddings of the files, the token

vectors of the terms of the file are averaged using tf-idf[1] weighting. These embeddings are calculated by multiplying each vector by the tf-idf weight of the related term before calculating the average.

The resulting vector is the length of the vocabulary size. The feature extractor uses the Python scikit-learn [55] library to generate the bag-of-words vector, and the Gensim [60] library for word2vec and fastText models. To train these models, source code from large repositories is used to learn the similarities between the source code tokens. The vocabulary is created from the source code of three large projects: (1) the IntelliJ community project[33], (2) the Android repository [8] and (3) the Android framework project. These repositories contain more than 70,000 Java files.

4.4 Architectural Metrics

Software architecture refers to software elements, their relationships, and the properties of both [10]. As discussed in Section 3.3, prior research has revealed the significant impact of architecture design decisions on software security. In particular, the study in [20] reported that complicated architectural connections among source files in a project contribute positively to the propagation of software vulnerability issues. Hence there is a motivation to investigate whether metrics that measure the complexity of architectural connections at the file level contribute positively to detecting software vulnerabilities using machine learning models.

A set of architectural connections are modelled as a graph, namely $G = \{F, D\}$, where F is the set of source files in the system, and D is the set of structural dependencies among the source files. The graph G of a software system can be reverse-engineered using existing tools, such as Scitool Understand [2].

For each source file, $f \in F$, *eight* metrics are captured to measure the file's connections with the rest of the system G. These metrics are feature representations to learn more vulnerabilities. These eight metrics are from three different but related aspects of software architecture:

First, *Fan-in* and *Fan-out* are measured of a file f, which counts the number of direct dependencies with f, and are commonly used for various analysis:

[1] Term Frequency - Inverse Document Frequency

[2] https://scitools.com/

1. Fan-in: The number of source files in G that directly depends on f.

2. Fan-out: The number of source files in G that f directly depends on.

Next, the position of f is measured in the entire dependency hierarchy of G. Cai et al. proposed an algorithm to cluster source files into hierarchical dependency layers based on their structural dependencies in G [14]. The key features of the layers are: 1) source files in the same layers form independent modules, and 2) the source files in a lower layer structurally depend on the upper layer, but not vice-versa. This layered structure is called the Architectural Design Rule Hierarchy (ArchDRH). The rationale is that source files in a higher layer structurally impact the source files in the lower layers. Therefore, the higher the layer of f, the more influential it is for the rest of the system.

3. Design Rule Hierarchy Layer: the layer number of f in the ArchDRH clustering.

Finally, the complexity of the transitive connections to each f in G is measured. For any $f \in F$, the $Butterfly\_Space_f = \{f, UpperWing, LowerWing\}$ is defined, where f is the center of the space. $UpperWing$ is the set of source files that directly and transitively depend on f. Similarly, $LowerWing$ is the set of source files that f directly and transitively depends on. For any $f \in G$, five metrics based on the $Butterfly\_Space$ notions are calculated:

4. Space Size: the total number of source files in $Butterfly\_Space_f$. This measures the total number of source files that f is connected to directly and transitively. The higher this value, the more significant is f connected to the rest of the system.

5. Upper Width: the width of the $UpperWing$. This measures the maximal number of branches that depend on f.

6. Upper Depth: the length of the longest path in the $UpperWing$. This measures the most far-reaching transitive dependency on f.

7. Lower Width: the width of the $LowerWing$. This measures the maximal number of branches that f depends on.

8. Lower Depth: the length of the longest path in the $LowerWing$. This measures the most far-reaching transitive dependency from f.

This study investigates whether and to what extent these metrics contribute to the learning of software vulnerabilities.

4.5 Machine Learning Models

This work performs a classification task to predict if a file is vulnerable or not. The objective is to observe the effects of machine learning models. Since a machine learning model is part of the decision process of the classification task, the model's transparency to the classification is considered. A random forest model has one form of transparency as the feature importance to the classification performance. A kernel-based Support Vector Machine is useful for data with irregular distribution or unknown distribution. The residual neural network (ResNet) model has been used to examine explainability methods[24]. Three kinds of machine learning models, decision tree-based Random Forests, kernel-based SVMs and deep neural networks as ResNet, are compared.

4.5.1 Random Forest

The Random forest (RF) is an ensemble learning method for supervised classification [70]. This model is constructed from multiple random decision trees. Those decision trees vote on how to classify a given instance of input data, and the random forest bootstraps those votes to prevent overfitting.

4.5.2 Support Vector Machines

Support Vector Machines (SVM) uses a kernel function to perform both linear and non-linear classifications [17]. The SVM algorithm creates a hyper-plane in a high-dimensional space that can separate the instances in the training set according to their class labels. SVM is one of the widely used machine learning algorithms for sentiment analysis in NLP.

4.5.3 Residual Neural Network

Residual Neural Network (ResNet) is a deep neuronal network model with residual blocks carrying linear data between neural layers. In this case, the structure of a ResNet model is composed of one convolutional layer, one dense layer and 7 ResNet blocks. Each ResNet block is composed of 16 layers. The detailed ResNet structure is depicted in Figure 4. The residual block has the following structure:

$$\mathbf{x}_{l+1} = h\left(\mathbf{x}_l\right) + \mathcal{F}\left(\hat{f}\left(\mathbf{x}_l\right), \mathcal{W}_l\right) \tag{1}$$

Where $\mathbf{x}$ is the input to the residual block and $\mathbf{l}$ indicates the $\mathbf{l-th}$ residual block. $\hat{\mathbf{f}}$ is the activation function which uses ReLU here. $\mathbf{F}$ is the residual function that contains two 1×3 convolutional layers. $\mathbf{W}$ stands for the corresponding parameters. The short cut $\mathbf{h}$ is defined as one 1×1 convolutional layer if the dimension of $\mathbf{x}_l$ and $\mathbf{x}_{l+1}$ doesn't match, otherwise h will be:

$$h\left(\mathbf{x}_l\right) = \mathbf{x}_l \tag{2}$$

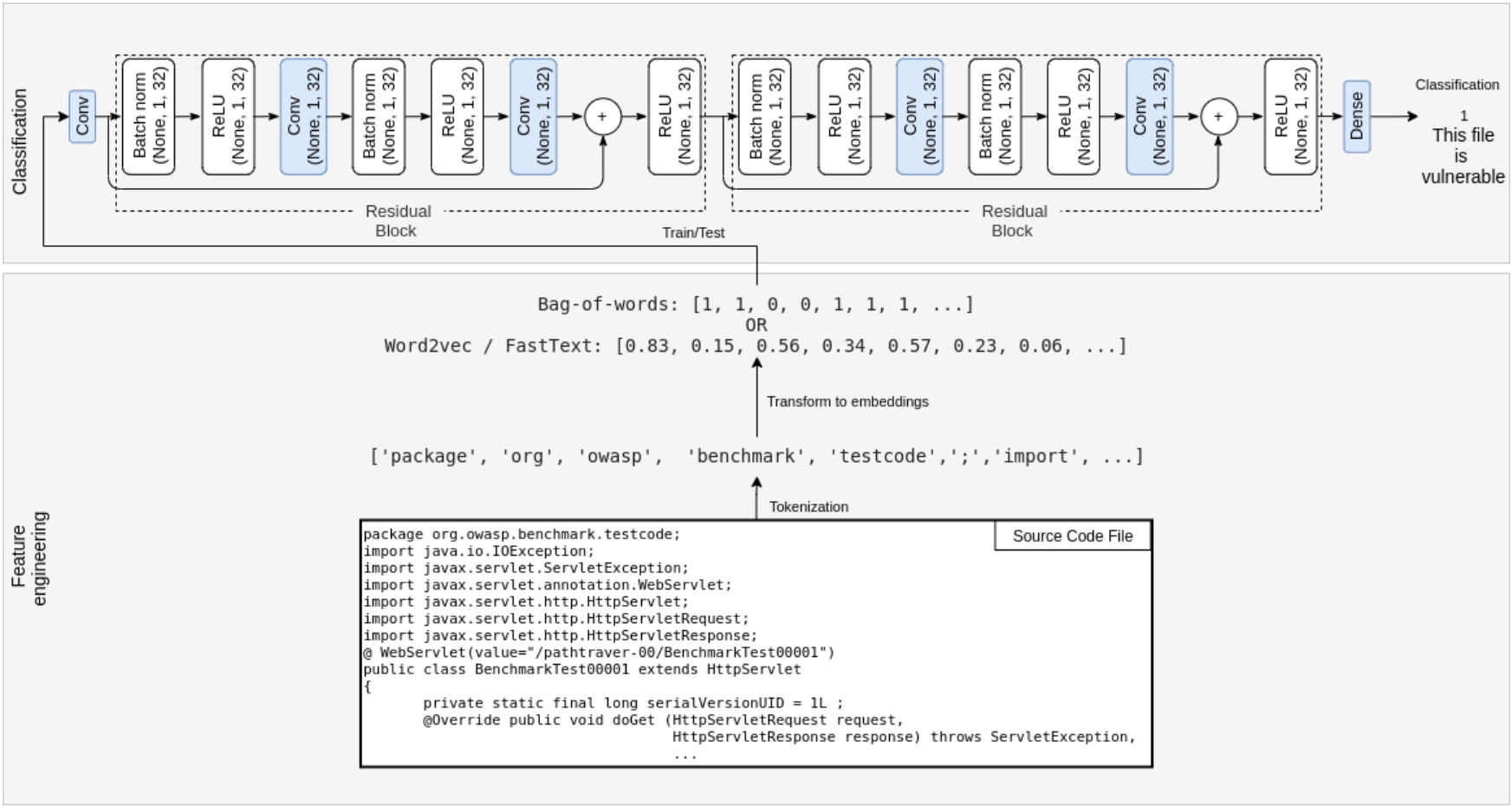

Figure 4: The data flow of the feature engineering and learning. The feature engineering is consistent for all the classification models. The modeling part illustrates the structure of the revised ResNet model that consists of one convolutional layer, one dense layer and 7 ResNet blocks. Each ResNet block is composed of 16 layers.

4.6 Experiments Design

Multiple experiments are performed to answer the research questions and evaluate each aspect (more details about these experiments and the results can be found in Chapter 6). For each research question, the aspect is evaluated using a set of experiments.

For the tokenization aspect, the experiments are performed using the two tokenization techniques combined with each machine learning model and each embedding technique. First, all the tokens from the source files are used. Then, only the tokens that are not comments or symbols.

For the embedding aspect, the tokes extracted are transformed into numerical vectors. The obtained vectors are used as input features to the three different models.

For the fourth aspect, the experiments aim to improve the learning using the architecture metrics. In the first place, the experiments use the metrics only. The second set of experiments for this aspect uses the metrics and the embeddings combined. The embedding method chosen is the one that performed the best in the previous experiments.

The three machine learning models classify the vulnerabilities with the different combinations of the four previously described aspects for the fifth aspect.

All these experiments are evaluated and compared using a set of metrics described in Section 5.2.

In total, 21 experiments per project are performed to evaluate which combination builds the best vulnerability prediction model for the used datasets. Table 1 summarizes the research questions and their related aspects that are explored in the next chapter with the experiments.

Table 1: This table summarizes the research questions and their related aspects that are explored with the experiment.

Research Question		Aspect	Experiments
RQ1	How does the filtering of tokens affect source code vulnerability detection?	Tokenization	Singular project training and testing using the two tokenization techniques. • All the tokens • The tokens after removing the comments and symbols
RQ2	Does a specific embedding technique perform better across software projects?	Token embeddings	Singular project training and testing using the three embedding methods: • Bag-of-words • Word2Vec • FastText
RQ3	Can architectural metrics that measure the structural complexity of software improve vulnerability detection?	Architecture metrics	Singular project training and testing using: • Only architecture metrics • Architecture metrics combined with embeddings
RQ4	Which machine learning model performs better across different projects?	Machine learning models	Singular project training and testing using the three models: • Random Forest • Support Vector machines • Residual Neural Network
RQ5	How transferable is the learning in predicting vulnerabilities of projects in cross-validation?	Transferability and Generalization	• Train-One-Predict-Multiple • Train-Multiple-Predict-One

Chapter 5

Datasets and Metrics

In this chapter, the different datasets used in this work are presented in the first section. The second section contains the definition of the evaluation metrics used to evaluate the experiments presented in the following chapters.

5.1 Datasets

The three datasets are labelled with vulnerabilities, including the OWASP Benchmark project [53], the Juliet test suite for Java [49] and 15 Android applications from the previous Android study [58]. OWASP and Juliet have the vulnerability types available online. Android study follows the labels published in the paper [58].

5.1.1 OWASP Benchmark Project

The OWASP Benchmark is a free test suite designed to evaluate automated software vulnerability detection tools. It contains 2740 test cases with 1415 vulnerable files (52%) and 1325 non-vulnerable files (48%). Table 2 enumerates the different types of vulnerabilities found in the OWASP project.

5.1.2 Test Suite for Java

This test suite contains 217 vulnerable files (58%) and 297 non-vulnerable files (42%). There are 112 different vulnerabilities and errors such as buffer overflow, OS injection, hard-coded password, absolute path traversal, NULL pointer dereference, uncaught

exception, deadlock, missing releases of resource and others listed in Table 3. The Juliet dataset has the most diversified vulnerability types among the datasets that are used in this work.

Table 2: OWASP vulnerability Types

Vulnerability Area	**CWE**	**# of files**
Command Injection	78	251
Weak Cryptography	327	246
Weak Hashing	328	236
LDAP Injection	90	59
Path Traversal	22	268
Secure Cookie Flag	614	67
SQL Injection	89	504
Trust Boundary Violation	501	126
Weak Randomness	330	493
XPath Injection	643	35
XSS (Cross-Site Scripting)	79	455

Table 3: Juliet Test Suite Vulnerability Types

Vulnerability Area	**CWE**	**# of files**
Integer Overflow or Wraparound	190	115
Integer Underflow	191	92
Improper Validation of Array Index	129	72
SQL Injection	89	60
Divide By Zero	369	50
Uncontrolled Memory Allocation	789	42
Uncontrolled Resource Consumption	400	39
HTTP Response Splitting	113	36
Numeric Truncation Error	197	33
Basic Cross-site scripting	80	18
Use of Externally-Controlled Format String	134	18
XPath Injection	643	12
Assignment to Variable without Use	563	12
Unchecked Input for Loop Condition	606	12
OS Command Injection	78	12
Relative Path Traversal	23	12
Unsafe Reflection	470	12
LDAP Injection	90	12
Absolute Path Traversal	36	12
Configuration Setting	15	12
Others		67

5.1.3 Android Study

The *Android Study* is a public dataset that contains 20 different Java applications that cover a variety of domains. This dataset is used in the work of Scandariato *et al.* [63]. According to [63], the source code was scanned using the Fortify Source Code Analyzer, a security scanning tool to mark the vulnerable files. The vulnerabilities that could be found are cross-site scripting, SQL injection, header manipulation, privacy violation and command injection. In total, the Android Study contains 2321 vulnerable files. Since Fortify itself may produce errors in the vulnerability scanning, the quality of labelling is not thoroughly evaluated. This is a potential threat to validity.

The labels of this dataset are binary (is vulnerable or not vulnerable), without the exact type of vulnerability for each file.

The application names, versions, and the file's paths with its vulnerable label are collected. With these references, a script is developed to retrieve 15 projects for evaluation.

5.1.4 Analysis of the Vulnerabilities

As shown in Table 4, the common vulnerabilities between the three datasets are the SQL injection (CWE 89) and the command injection (CWE 78). The vulnerabilities in common between OWASP and Juliet are command injection (CWE 78), LDAP injection (CWE 90), SQL injection and XPATH injection (CWE 643). And the vulnerability type that can be found in all three projects is Cross-site scripting (CWE 79 & 80).

Table 4: Common vulnerabilities in the three datasets

Vulnerability Area	CWE	OWASP	Juliet	Android
Cross-Site Scripting	79	X	X	X
SQL injection	89	X	X	X
Command Injection	78			X
XPath Injection	643	X	X	
OS Command Injection	78	X	X	
LDAP Injection	90	X	X	

Table 5 shows the 17 applications used in this project and the vulnerability rate of the labelled source code for each. On average, 43% of the files contain at least one

vulnerability.

Table 5: Dataset Vulnerability Statistics

	Projects	Vulnerability rate	Number of files	# of tokens
1	QuickSearchBox	23%	654	4301
2	FBReader	30%	3450	6589
3	Contacts	31%	787	13438
4	Browser	37%	433	9561
5	Mms	37%	865	7965
6	Camera	38%	475	7851
7	KeePassDroid	39%	1580	2872
8	Calendar	44%	307	8003
9	ConnectBot	46%	104	4109
10	Crosswords	46%	842	4223
11	K9	47%	2660	13175
12	Deskclock	47%	127	2163
13	Coolreader	49%	423	5424
14	OWASP	52%	2740	6154
15	Email	54%	840	15454
16	Juliet	58%	514	1268
17	AnkiDroid	59%	275	8408

5.1.5 Analysis of Tokens

Each line of code is parsed to produce tokens, including variables, preserved keywords, operators, symbols and separators. Table 6 shows the number of tokens in each dataset in the vulnerable and non-vulnerable files.

Table 6: Number of tokens in each dataset according to the vulnerability of the files and number of the common tokens in the vulnerable files and non vulnerable files.

Dataset	# of tokens in vulnerable files	# of tokens in non vulnerable files	# of tokens in common
Owasp	2982	3599	605
Juliet	678	764	460
Android	54196	28698	18339

First, the analysis of the token statistics and observation of the notable characters of the tokens. For each dataset OWASP, Juliet, and Android project, the tokens are separated according to the vulnerability of the files and the token frequency distribution is plotted in Figure 5, Figure 6, and Figure 7 respectively.

For the OWASP project (shown in Figure 5), tokens are mostly grouped in the counts of occurrence that are less than 20. Beyond 20 occurrences, the counts of tokens are significantly smaller. In Juliet source code (shown in Figure 6), the distribution of the token frequency has more peaks than the OWASP token distribution. In all the Android source code (shown in Figure 7), the tokens are mostly grouped with occurrences of less than 30.

The charts show two facts:

(1) The frequency distribution of each project varies. This could be a factor in downgrading the accuracy of cross-domain learning;

(2) The high-frequency tokens are neutral such as "main", "string_builder". This indicates that the feature representation is learnt mainly from tokens with less frequency. These two facts relate to the experimental results presented in section 6.1.1.

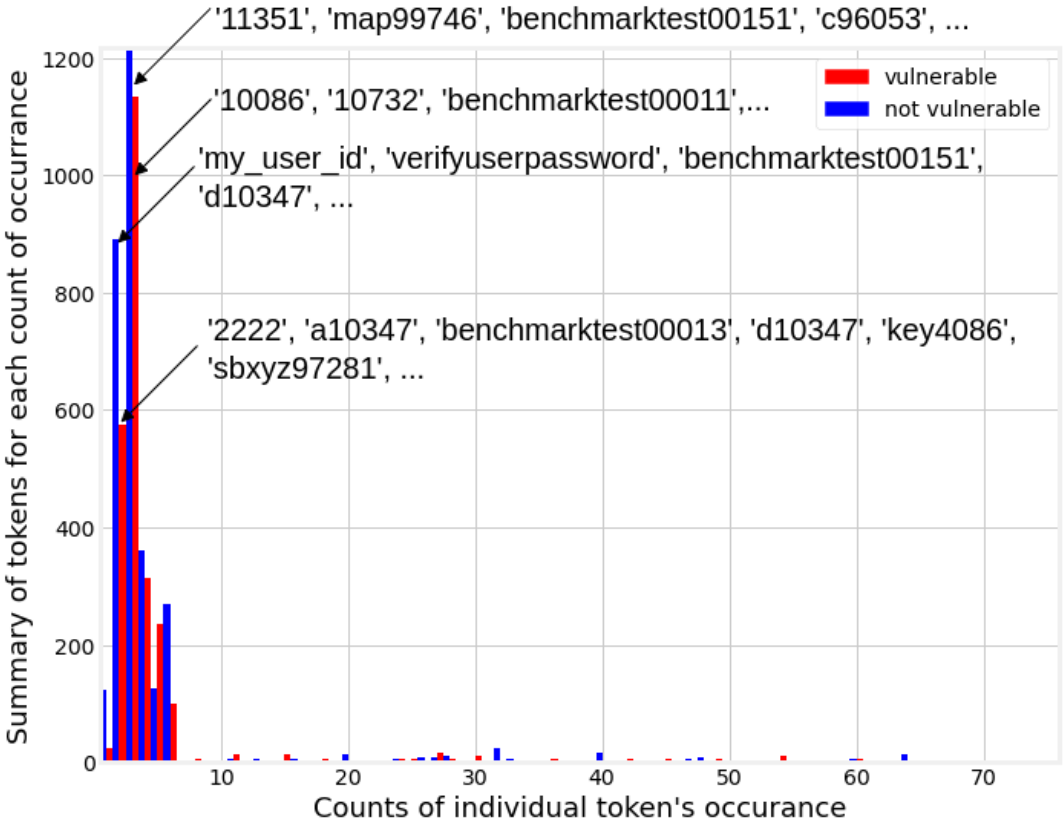

Figure 5: OWASP token distribution. Most of the tokens have fewer than 20 occurrences.

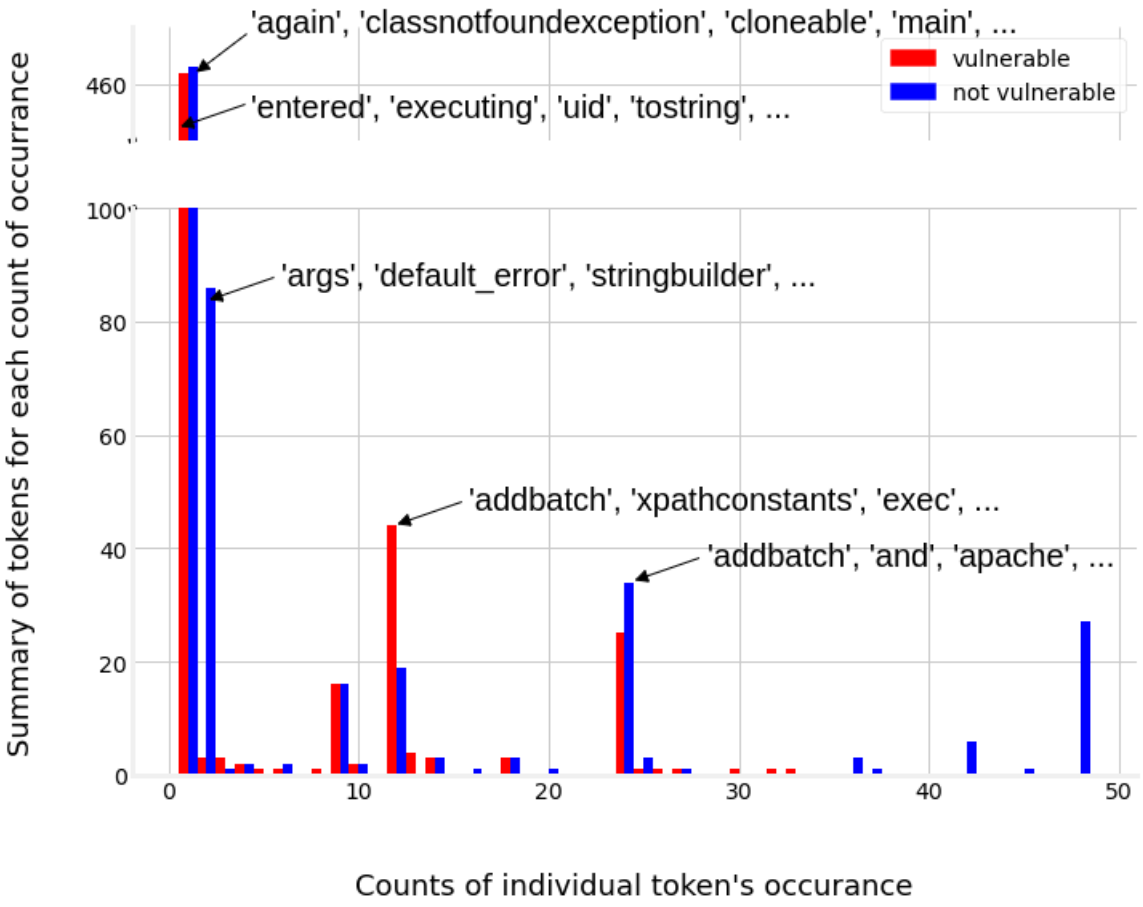

Figure 6: Juliet token distribution. Most of the tokens are have fewer than 25 occurrences.

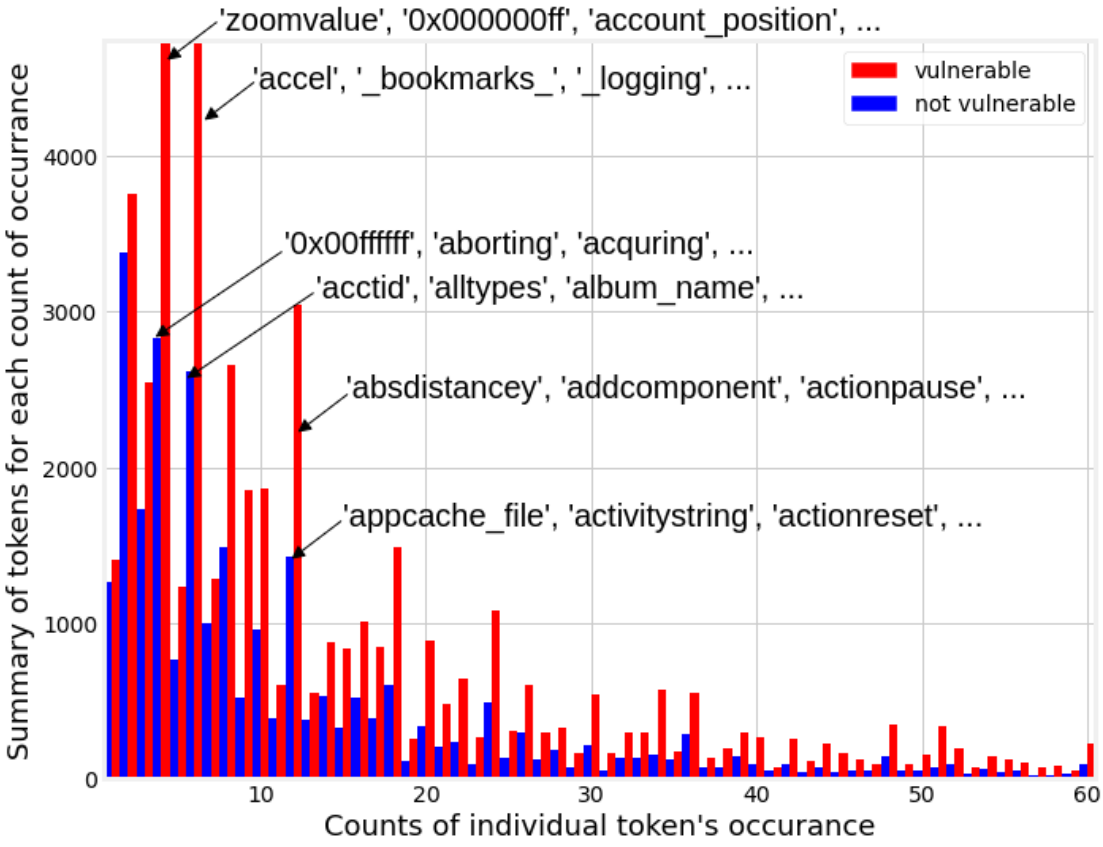

Figure 7: All Android projects token distribution. The majority of the tokens have fewer than 500 occurrences.

5.2 Evaluation Metrics

For each experiment, the evaluation of the performance for the vulnerability detection is made using traditional Information Retrieval metrics. In the context of this study, true positives (TP) are the correct identification of source files with vulnerabilities. True negatives (TN) are the correct identification of source files without vulnerability. False positives (FP) are the incorrect identification of source files with vulnerabilities. False negatives (FN) are the incorrect identification of source files without vulnerability. Based on these metrics, the following metrics are defined to measure the performance of vulnerability detection:

- The precision (P), which is the probability that a sample of code is classified vulnerable, is truly vulnerable.

$$P = \frac{TP}{(TP + FP)} \tag{3}$$

- The recall (R) which is the probability that a vulnerable sample of code is classified as vulnerable.

$$R = \frac{TP}{(TP + FN)} \tag{4}$$

- The f-measure (F1) is the harmonic average of precision and recall.

$$F1 = 2 * \frac{1}{\frac{1}{P} + \frac{1}{R}} \tag{5}$$

- The false positive rate (FPR) is the proportion of negative cases incorrectly identified as positive cases.

$$FPR = \frac{FP}{(FP + TN)} \tag{6}$$

- The area under the precision-recall curve (PR AUC) summarizes the information in the precision-recall curve.

- The area under the receiver operating characteristic curve (ROC AUC) shows the model's capability to distinguish between classes.

To compare the different experiments and evaluate the hypothesis, an aggregation formula of all the metrics is defined bu equation 7. All the metrics that are auditioned are defined above. The value is then normalized using the equation 8.

$\forall s \in S\{P, R, F1, FPR, ROCAUC, PRAUC\}$

$$\mathbf{x}_i = \sum \mathbf{s}_i \tag{7}$$

$$\mathbf{z}_i = \frac{\mathbf{x}_i - \min_{\forall k \in [1,N]}(x_k)}{\max_{\forall k \in [1,N]}(x_k) - \min_{\forall k \in [1,N]}(x_k)} \tag{8}$$

The z value is calculated according to all the experiments. The min and the max values are the minimum and maximum overall for the compared experiments. The higher the value, the better the results. If z equals 1, the experiment performed the best among all. If it is equal to 0, it performed the worst.

Chapter 6

Experiments and Analysis

In this Chapter, the different experiments are performed to evaluate various aspects presented in the previous chapters. The experiments' results are presented and analyzed. Also, cross-validation experiments are carried out to assess the transferability of the model across projects and domains.

6.1 Evaluate the Combinations of the Different Aspects

The evaluations are based on the following tasks:

1. Defining hypotheses to answer each research question;
2. Preparing appropriate datasets;
3. Defining metrics to evaluate the learning effects;
4. Running experiments and collecting results to test the hypotheses.

The hypotheses mentioned in the 4th task allow to evaluate if tokenization techniques, embedding techniques, architectural metrics, and machine learning models have significant effects on the ability to learn software vulnerabilities.

The results of these experiments are included in Table 7, Table 8, and Table 9. Table 7 contains the results obtained from the experiments using the source code files with all tokens. Table 8 gives the results obtained from the experiments using the

source code files without comments and symbols. Table 9 shows the results of using the architectural metrics as features compared to bag-of-words.

Table 7: Singular project vulnerability detection with tokenization while keeping all the comments and symbols across embeddings and machine learning models. The z value in the table aggregate the evaluation metrics. The value is calculated according to all the experiments. The higher the value, the better the results. If z equals 1, the experiment performed the best among all. If it is equal to 0, it performed the worst.

			Bag-of-words							Word2vec							FastText						
Project		Classifier	P	R	F1	FPR	ROC AUC	PR AUC	z	P	R	F3	FPR	ROC AUC	PR AUC	z	P	R	F2	FPR	ROC AUC	PR AUC	z
1	OWASP	RF	1.00	1.00	1.00	0.21	1.00	1.00	0.95	0.68	0.76	0.72	0.21	0.70	0.63	0.60	1.00	1.00	1.00	0.21	1.00	1.00	0.95
		ResNet	0.99	0.92	0.95	0.10	0.96	0.95	0.92	0.95	0.93	0.94	0.04	0.94	0.92	0.92	0.76	0.96	0.84	0.04	0.85	0.87	0.82
		SVM	0.99	0.99	0.99	0.24	0.99	0.99	0.93	0.91	0.93	0.92	0.19	0.93	0.89	0.86	0.82	0.90	0.86	0.08	0.93	0.92	0.85
2	Juliet	RF	1.00	1.00	1.00	0.08	1.00	1.00	0.98	0.07	0.05	0.05	0.06	0.29	0.41	0.02	0.12	0.09	0.10	0.04	0.30	0.40	0.05
		ResNet	1.00	0.73	0.84	0.13	0.86	0.84	0.80	0.21	0.18	0.20	0.02	0.34	0.38	0.13	0.38	0.68	0.49	0.11	0.44	0.64	0.42
		SVM	1.00	1.00	1.00	0.07	1.00	1.00	0.98	0.17	0.05	0.07	0.05	0.50	0.42	0.10	0.42	0.23	0.29	0.84	0.50	0.42	0.07
3	AnkiDroid	RF	0.80	0.89	0.84	0.15	0.84	0.76	0.76	0.85	0.97	0.91	0.10	0.89	0.84	0.85	0.85	0.97	0.91	0.10	0.89	0.84	0.85
		ResNet	0.79	0.85	0.82	0.21	0.82	0.75	0.72	0.88	0.50	0.64	0.08	0.71	0.71	0.62	0.80	0.13	0.23	0.06	0.55	0.57	0.35
		SVM	0.80	0.89	0.84	0.13	0.86	0.78	0.78	0.80	0.93	0.86	0.34	0.83	0.78	0.73	0.82	0.93	0.87	0.63	0.85	0.80	0.69
4	Browser	RF	0.97	0.93	0.95	0.08	1.00	1.00	0.95	0.97	0.94	0.95	0.06	0.96	0.93	0.92	0.89	1.00	0.94	0.06	0.97	0.92	0.92
		ResNet	0.94	0.88	0.91	0.09	0.92	0.87	0.87	0.38	0.97	0.55	0.10	0.56	0.38	0.47	0.83	0.91	0.87	0.02	0.90	0.88	0.85
		SVM	0.91	0.94	0.93	0.10	0.94	0.88	0.88	0.82	0.90	0.86	0.17	0.90	0.78	0.79	0.88	0.72	0.79	0.46	0.90	0.85	0.69
5	Calendar	RF	0.87	0.87	0.89	0.00	0.92	0.82	0.85	0.89	0.86	0.88	0.00	0.88	0.84	0.85	1.00	0.86	0.93	0.00	0.92	0.95	0.92
		ResNet	0.85	1.00	0.92	0.00	0.95	0.85	0.90	0.58	0.97	0.73	0.00	0.67	0.58	0.66	0.88	0.48	0.62	0.00	0.71	0.80	0.65
		SVM	0.88	0.95	0.91	0.00	0.94	0.85	0.89	0.86	0.86	0.86	0.00	0.87	0.81	0.83	0.84	0.72	0.78	0.00	0.88	0.91	0.80
6	Camera	RF	0.94	0.91	0.93	0.08	0.94	0.89	0.89	0.89	0.83	0.86	0.08	0.89	0.80	0.81	0.82	0.75	0.78	0.11	0.95	0.84	0.77
		ResNet	0.91	0.91	0.91	0.10	0.93	0.87	0.87	0.55	1.00	0.71	0.05	0.77	0.55	0.65	0.92	0.46	0.61	0.10	0.72	0.76	0.62
		SVM	0.91	0.94	0.93	0.04	0.94	0.88	0.90	0.82	0.77	0.79	0.06	0.80	0.67	0.72	0.72	0.75	0.73	0.37	0.90	0.82	0.66
7	ConnectBot	RF	1.00	1.00	1.00	0.00	1.00	1.00	1.00	1.00	0.80	0.89	0.04	0.90	0.90	0.87	1.00	0.80	0.89	0.00	0.90	0.90	0.88
		ResNet	1.00	1.00	1.00	0.00	1.00	0.90	0.98	1.00	0.07	0.13	0.12	0.53	0.55	0.33	1.00	0.80	0.89	0.12	0.90	0.90	0.85
		SVM	1.00	1.00	1.00	0.02	1.00	1.00	0.99	1.00	0.80	0.89	0.06	0.90	0.90	0.87	1.00	0.80	0.89	0.12	0.90	0.90	0.85
8	Contacts	RF	0.90	0.96	0.93	0.11	0.96	0.88	0.89	0.83	0.86	0.85	0.06	0.89	0.76	0.80	0.89	1.00	0.94	0.06	0.99	0.97	0.94
		ResNet	0.78	0.90	0.83	0.08	0.89	0.73	0.78	0.56	1.00	0.72	0.15	0.81	0.56	0.65	0.79	0.67	0.73	1.00	0.79	0.79	0.48
		SVM	0.90	0.96	0.93	0.21	0.96	0.88	0.86	0.80	0.78	0.79	0.36	0.87	0.77	0.69	0.85	0.80	0.82	1.00	0.93	0.82	0.58
9	Coolreader	RF	1.00	0.98	0.99	0.09	1.00	1.00	0.97	1.00	1.00	1.00	0.03	1.00	1.00	0.99	1.00	1.00	1.00	0.05	1.00	1.00	0.99
		ResNet	1.00	0.93	0.96	0.12	0.96	0.98	0.93	0.80	0.73	0.76	0.08	0.80	0.69	0.70	0.83	0.91	0.87	0.30	0.89	0.79	0.76
		SVM	0.97	0.97	0.97	0.03	0.96	0.93	0.95	1.00	1.00	1.00	0.11	1.00	1.00	0.97	1.00	1.00	1.00	0.39	1.00	1.00	0.91
10	Deskclock	RF	0.89	1.00	0.94	0.02	0.97	0.89	0.92	0.86	1.00	0.92	0.02	0.93	0.86	0.89	0.90	1.00	0.95	0.02	0.99	0.98	0.95
		ResNet	0.88	0.88	0.88	0.03	0.91	0.80	0.84	0.46	1.00	0.63	0.05	0.50	0.46	0.53	0.35	1.00	0.51	0.01	0.50	0.67	0.54
		SVM	0.89	1.00	0.94	0.02	0.97	0.89	0.92	0.80	1.00	0.89	0.04	0.93	0.86	0.87	0.82	1.00	0.90	0.02	1.00	0.99	0.93
11	Email	RF	0.97	0.98	0.97	0.30	0.99	0.99	0.91	0.98	0.98	0.98	0.00	0.99	1.00	0.98	0.91	0.95	0.93	0.00	0.98	0.97	0.94
		ResNet	0.96	0.90	0.93	0.23	0.93	0.96	0.87	0.74	0.88	0.81	0.33	0.75	0.84	0.69	0.76	0.91	0.83	0.56	0.80	0.86	0.67
		SVM	0.95	0.82	0.88	0.23	0.94	0.95	0.84	0.79	0.84	0.81	0.27	0.88	0.89	0.75	0.80	0.92	0.86	0.27	0.91	0.90	0.79
12	FBReader	RF	0.96	0.93	0.94	0.01	0.98	0.98	0.95	0.96	0.95	0.95	0.01	0.99	0.99	0.96	0.97	0.94	0.95	0.06	0.99	0.99	0.95
		ResNet	0.95	0.96	0.96	0.10	0.97	0.93	0.92	0.96	0.94	0.95	0.00	0.96	0.96	0.94	0.97	0.90	0.93	0.01	0.94	0.95	0.93
		SVM	0.95	0.97	0.96	0.00	0.97	0.93	0.95	0.76	0.72	0.74	0.00	0.91	0.83	0.75	0.84	0.91	0.87	0.05	0.95	0.92	0.87
13	K9	RF	0.97	0.99	0.98	0.00	1.00	1.00	0.99	0.99	1.00	0.99	0.00	1.00	1.00	0.99	0.99	1.00	1.00	0.01	1.00	1.00	0.99
		ResNet	0.94	1.00	0.97	0.00	0.97	0.97	0.96	0.94	0.85	0.89	0.01	0.90	0.94	0.88	0.99	1.00	0.99	0.01	0.99	1.00	0.99
		SVM	0.99	1.00	0.99	0.01	0.99	0.99	0.99	0.83	0.88	0.85	0.00	0.92	0.92	0.86	0.98	0.98	0.98	0.01	0.99	0.99	0.98
14	KeePassDroid	RF	0.99	1.00	1.00	0.01	1.00	1.00	1.00	1.00	0.99	1.00	0.01	1.00	1.00	0.99	0.98	0.99	0.98	0.01	1.00	1.00	0.99
		ResNet	0.99	0.99	0.99	0.05	0.99	0.98	0.97	0.97	0.84	0.90	0.03	0.91	0.94	0.89	0.99	1.00	0.99	0.08	1.00	0.99	0.97
		SVM	0.99	0.99	0.99	0.02	0.99	0.98	0.98	0.90	0.82	0.86	0.07	0.95	0.92	0.85	0.98	1.00	0.99	0.42	1.00	0.99	0.89
15	Mms	RF	0.98	0.97	0.98	0.22	1.00	0.99	0.93	0.98	0.97	0.98	0.01	0.98	0.96	0.97	0.94	1.00	0.97	0.01	0.98	0.93	0.96
		ResNet	0.98	0.93	0.96	0.44	0.96	0.94	0.84	0.57	0.98	0.72	0.17	0.78	0.56	0.64	0.86	0.95	0.90	0.32	0.94	0.91	0.82
		SVM	0.98	0.97	0.97	0.12	0.97	0.95	0.93	0.98	0.95	0.97	0.08	0.97	0.95	0.94	0.89	0.93	0.91	0.07	0.98	0.95	0.90
16	Crosswords	RF	1.00	1.00	1.00	0.00	1.00	1.00	1.00	1.00	0.97	0.99	0.00	0.99	0.99	0.98	0.99	1.00	0.99	0.00	1.00	0.99	0.99
		ResNet	1.00	1.00	1.00	0.01	1.00	1.00	1.00	0.94	0.91	0.92	0.11	0.93	0.89	0.88	0.88	0.89	0.88	0.09	0.90	0.82	0.83
		SVM	1.00	0.99	0.99	0.15	0.99	0.99	0.96	0.97	0.92	0.95	0.00	0.97	0.95	0.95	0.95	1.00	0.97	0.05	0.98	0.95	0.95
17	QuickSearchBox	RF	0.95	0.87	0.91	0.01	0.93	0.85	0.96	0.77	0.89	0.83	0.07	0.91	0.71	0.85	0.94	0.94	0.94	0.03	0.96	0.90	1.00
		ResNet	1.00	0.78	0.88	0.00	0.89	0.82	0.93	0.58	0.96	0.72	0.18	0.89	0.56	0.72	0.85	0.79	0.82	0.06	0.87	0.74	0.84
		SVM	0.95	0.87	0.91	0.01	0.93	0.85	0.96	0.74	0.63	0.68	0.06	0.79	0.54	0.66	0.86	0.86	0.90	0.06	0.94	0.84	0.92

Table 8: Singular project vulnerability detection with tokenization after removing the comments and symbols across embeddings and machine learning models. The z value in the table aggregate the evaluation metrics. The value is calculated according to all the experiments. The higher the value, the better the results.

			Bag-of-Words							Word2vec							FastText						
	Project	Classifier	P	R	F1	FPR	ROC AUC	PR AUC	z	P	R	F3	FPR	ROC AUC	PR AUC	z	P	R	F2	FPR	ROC AUC	PR AUC	z
		RF	0.99	1.00	0.99	0.21	0.99	0.99	0.94	0.71	0.73	0.72	0.21	0.72	0.65	0.60	0.75	0.83	0.79	0.21	0.89	0.89	0.75
1	OWASP	Resnet	0.82	1.00	0.90	0.17	0.89	0.82	0.83	0.79	0.93	0.86	0.19	0.85	0.77	0.77	0.57	0.58	0.58	0.19	0.57	0.68	0.48
		SVM	0.99	0.99	0.99	0.24	0.99	0.99	0.93	0.88	0.90	0.89	0.31	0.89	0.84	0.78	0.60	0.79	0.68	0.19	0.68	0.67	0.58
		RF	0.03	0.02	0.03	0.04	0.26	0.42	0.00	0.12	0.09	0.10	0.04	0.30	0.40	0.05	0.23	0.18	0.20	0.02	0.52	0.36	0.17
2	Juliet	Resnet	0.33	0.05	0.08	0.04	0.35	0.42	0.11	0.22	0.09	0.13	0.03	0.43	0.40	0.12	0.47	0.34	0.37	0.07	0.54	0.54	0.34
		SVM	0.41	0.02	0.03	0.04	0.68	0.54	0.21	0.20	0.09	0.13	0.03	0.47	0.42	0.13	0.42	0.16	0.23	0.02	0.62	0.46	0.27
		RF	0.81	0.96	0.88	0.08	0.88	0.80	0.83	0.78	0.93	0.85	0.08	0.84	0.76	0.78	0.84	0.96	0.90	0.08	0.90	0.83	0.85
3	Anki-Android	Resnet	0.82	1.00	0.90	0.11	0.90	0.82	0.84	0.78	0.93	0.85	0.05	0.84	0.76	0.79	0.82	0.52	0.64	0.00	0.71	0.66	0.61
		SVM	0.81	0.96	0.88	0.16	0.90	0.82	0.82	0.80	0.89	0.84	0.13	0.84	0.76	0.77	0.77	0.85	0.81	0.09	0.65	0.57	0.66
		RF	0.94	0.91	0.93	0.04	0.94	0.89	0.90	0.94	0.94	0.94	0.04	0.95	0.90	0.91	0.93	0.84	0.89	0.04	0.96	0.87	0.87
4	Browser	Resnet	0.88	0.85	0.87	0.03	0.89	0.81	0.83	0.88	0.91	0.90	0.03	0.92	0.84	0.86	0.81	0.91	0.86	0.07	0.89	0.77	0.81
		SVM	0.91	0.91	0.91	0.05	0.94	0.88	0.89	0.89	1.00	0.94	0.07	0.96	0.89	0.91	0.90	0.82	0.86	0.06	0.88	0.80	0.81
		RF	0.88	0.95	0.91	0.00	0.94	0.85	0.89	0.73	0.70	0.71	0.00	0.77	0.62	0.65	0.81	0.74	0.77	0.00	0.82	0.70	0.73
5	Calendar	Resnet	0.78	0.95	0.86	0.00	0.90	0.76	0.82	0.76	0.70	0.73	0.00	0.78	0.64	0.68	0.78	0.89	0.83	0.00	0.84	0.75	0.79
		SVM	0.88	0.95	0.91	0.00	0.94	0.85	0.89	0.71	0.74	0.72	0.00	0.78	0.62	0.67	0.77	0.74	0.76	0.00	0.80	0.67	0.71
		RF	0.87	0.91	0.93	0.07	0.94	0.89	0.88	0.87	0.83	0.85	0.05	0.89	0.77	0.81	0.92	0.88	0.90	0.05	0.97	0.94	0.90
6	Camera	Resnet	0.88	0.88	0.88	0.92	0.90	0.83	0.64	0.78	0.88	0.82	0.05	0.89	0.72	0.77	0.81	0.85	0.83	0.07	0.88	0.85	0.80
		SVM	0.91	0.91	0.91	0.04	0.94	0.88	0.89	0.88	0.92	0.90	0.07	0.93	0.93	0.88	0.81	0.81	0.81	0.08	0.89	0.74	0.76
		RF	1.00	1.00	1.00	0.00	1.00	1.00	1.00	1.00	1.00	1.00	0.00	1.00	1.00	1.00	1.00	1.00	1.00	0.00	1.00	1.00	1.00
7	Connectbot	Resnet	1.00	1.00	1.00	0.00	1.00	1.00	1.00	1.00	1.00	1.00	0.00	1.00	1.00	1.00	1.00	0.82	0.90	0.00	0.91	0.93	0.90
		SVM	1.00	1.00	1.00	0.00	1.00	1.00	1.00	1.00	1.00	1.00	0.00	1.00	1.00	1.00	1.00	1.00	1.00	0.00	1.00	1.00	1.00
		RF	0.85	0.96	0.90	0.06	0.98	0.95	0.90	0.93	0.91	0.92	0.05	0.94	0.88	0.89	0.96	0.89	0.92	0.06	0.93	0.89	0.89
8	Contacts	Resnet	0.83	0.92	0.87	0.08	0.92	0.89	0.85	0.90	0.67	0.77	0.15	0.81	0.72	0.70	0.81	0.87	0.84	0.06	0.88	0.74	0.78
		SVM	0.80	0.67	0.73	0.07	0.90	0.84	0.73	0.86	0.77	0.81	0.14	0.85	0.76	0.75	0.85	0.83	0.84	0.14	0.74	0.62	0.70
		RF	1.00	0.97	0.99	0.04	0.99	0.99	0.97	1.00	1.00	1.00	0.01	1.00	1.00	1.00	1.00	0.98	0.99	0.04	0.99	0.99	0.98
9	CoolReader	Resnet	1.00	0.97	0.99	0.04	0.99	0.99	0.97	0.98	1.00	0.99	0.01	0.99	0.98	0.98	1.00	1.00	1.00	0.10	1.00	1.00	0.98
		SVM	0.97	0.97	0.97	0.09	0.96	0.93	0.94	0.98	1.00	0.99	0.00	0.99	0.98	0.98	1.00	0.98	0.99	0.03	0.99	0.99	0.98
		RF	0.89	1.00	0.94	0.02	0.97	0.89	0.92	0.91	0.83	0.87	0.02	0.88	0.83	0.84	0.92	0.79	0.85	0.02	0.93	0.87	0.84
10	DeskClock	Resnet	0.98	1.00	0.89	0.01	0.94	0.80	0.91	0.75	0.75	0.75	0.01	0.77	0.68	0.69	0.50	0.07	0.13	0.01	0.49	0.54	0.23
		SVM	0.89	1.00	0.94	0.01	0.97	0.89	0.93	0.83	0.83	0.83	0.02	0.85	0.77	0.79	0.80	0.57	0.67	0.02	0.86	0.88	0.71
		RF	0.97	0.98	0.97	0.50	1.00	1.00	0.86	0.93	0.99	0.96	0.00	0.98	0.97	0.96	0.97	0.98	0.97	0.00	0.97	0.96	0.97
11	Email	Resnet	0.93	1.00	0.96	0.41	0.95	0.97	0.86	0.97	0.75	0.85	0.47	0.86	0.87	0.72	0.91	0.99	0.95	0.45	0.93	0.91	0.82
		SVM	0.96	0.85	0.90	0.50	0.96	0.97	0.80	0.97	0.98	0.97	0.45	0.97	0.96	0.86	0.94	0.95	0.94	0.45	0.93	0.92	0.82
		RF	0.96	0.96	0.97	0.01	0.98	0.95	0.96	0.98	0.95	0.97	0.02	0.97	0.95	0.95	0.97	0.95	0.96	0.02	1.00	0.99	0.96
12	FBReaderJ	Resnet	0.96	0.96	0.96	0.01	0.97	0.93	0.95	0.95	0.90	0.93	0.00	0.94	0.89	0.91	0.92	0.82	0.87	0.01	0.90	0.90	0.86
		SVM	0.95	0.97	0.96	0.02	0.97	0.93	0.94	0.93	0.85	0.89	0.01	0.91	0.83	0.86	0.75	0.50	0.60	0.01	0.86	0.69	0.62
		RF	0.99	1.00	0.99	0.00	0.99	0.99	0.99	0.98	0.99	0.98	0.00	1.00	1.00	0.99	0.99	0.99	0.99	0.00	1.00	1.00	0.99
13	K9Mail	Resnet	0.99	0.99	0.99	0.00	0.99	0.99	0.99	1.00	0.94	0.97	0.00	0.97	0.97	0.96	0.92	0.91	0.91	0.01	0.91	0.94	0.90
		SVM	0.99	1.00	1.00	0.01	0.99	0.99	0.99	0.98	1.00	0.99	0.00	0.98	0.97	0.98	0.77	0.88	0.82	0.00	0.85	0.80	0.79
		RF	1.00	1.00	1.00	0.01	1.00	1.00	1.00	0.99	1.00	1.00	0.01	1.00	0.99	0.99	0.99	1.00	1.00	0.01	1.00	1.00	1.00
14	KeePassAndroid	Resnet	1.00	1.00	1.00	0.03	1.00	1.00	0.99	0.99	1.00	1.00	0.03	1.00	0.99	0.99	1.00	0.99	0.99	0.03	0.99	1.00	0.98
		SVM	0.98	0.97	0.97	0.03	1.00	1.00	0.97	0.99	1.00	1.00	0.03	1.00	0.99	0.99	0.83	0.86	0.84	0.01	0.92	0.84	0.83
		RF	0.98	0.97	0.97	0.01	0.98	0.96	0.96	0.96	0.98	0.97	0.01	0.98	0.95	0.96	0.96	0.98	0.97	0.01	0.98	0.95	0.96
15	MMS	Resnet	0.98	0.91	0.94	0.28	0.95	0.96	0.87	0.96	0.67	0.79	0.00	0.82	0.76	0.76	0.91	0.95	0.93	0.00	0.95	0.89	0.92
		SVM	0.98	0.97	0.97	0.12	0.98	0.96	0.94	0.96	0.98	0.97	0.04	0.97	0.94	0.95	0.96	0.98	0.97	0.09	0.98	0.95	0.94
		RF	1.00	1.00	1.00	0.00	1.00	1.00	1.00	0.99	1.00	0.99	0.00	0.99	0.99	0.99	1.00	1.00	1.00	0.00	1.00	1.00	1.00
16	Xwords	Resnet	1.00	0.99	0.99	0.00	0.99	0.99	0.99	0.86	1.00	0.92	0.00	0.92	0.86	0.90	0.95	0.25	0.40	0.01	0.62	0.62	0.49
		SVM	1.00	0.99	0.99	0.01	0.99	0.99	0.99	1.00	0.99	0.99	0.00	0.99	0.98	0.99	1.00	1.00	1.00	0.00	1.00	1.00	1.00
		RF	0.95	0.87	0.91	0.01	0.93	0.85	0.96	0.88	0.81	0.85	0.03	0.89	0.76	0.87	0.93	0.95	0.94	0.03	0.96	0.90	1.00
17	QuickSearchBox	Resnet	0.95	0.91	0.93	0.01	0.95	0.89	0.99	0.77	0.89	0.83	0.07	0.91	0.71	0.85	0.86	0.99	0.92	0.07	0.96	0.86	0.97
		SVM	0.95	0.87	0.91	0.01	0.93	0.85	0.96	0.89	0.89	0.89	0.03	0.90	0.82	0.92	0.85	0.88	0.86	0.07	0.89	0.77	0.88

Table 9: Singular project vulnerability detection with Bag-of-words and the Architectural metrics. The z value in the table aggregate the evaluation metrics. The value is calculated according to all the experiments. The higher the value, the better the results. If z equals 1, the experiment performed the best among all. If it is equal to 0, it performed the worst.

			Metrics only							Metrics + bag-of-words						
Project		Classifier	P	R	F1	FPR	ROC AUC	PR AUC	z	P	R	F1	FPR	ROC AUC	PR AUC	z
1	OWASP	RF	0.66	0.78	0.71	0.38	0.70	0.62	0.56	0.82	0.85	0.84	0.17	0.95	0.96	0.73
		Resnet	0.48	1.00	0.65	1.00	0.50	0.48	0.32	0.70	0.89	0.79	0.34	0.77	0.82	0.51
		SVM	0.57	0.93	0.70	0.66	0.64	0.56	0.48	0.67	0.74	0.70	0.74	0.82	0.85	0.30
2	Juliet	RF	0.50	0.41	0.45	0.23	0.59	0.42	0.33	1.00	0.88	0.93	0.00	0.94	0.92	0.88
		Resnet	0.35	0.97	0.52	1.00	0.48	0.35	0.21	1.00	0.81	0.90	0.00	0.91	0.88	0.82
		SVM	0.00	0.00	0.00	0.00	0.50	0.36	0.01	1.00	0.84	0.92	0.00	0.94	0.92	0.86
3	Anki-Android	RF	0.62	0.73	0.67	0.26	0.74	0.55	0.55	0.87	0.91	0.89	0.08	0.92	0.82	0.76
		Resnet	0.36	1.00	0.53	1.00	0.50	0.36	0.23	0.83	0.86	0.84	0.10	0.88	0.76	0.67
		SVM	0.71	0.23	0.34	0.05	0.50	0.36	0.32	0.88	0.95	0.91	0.08	0.94	0.85	0.81
4	Browser	RF	0.72	0.62	0.67	0.16	0.73	0.60	0.59	0.94	0.91	0.93	0.04	0.94	0.89	0.84
		Resnet	0.00	0.00	0.00	0.00	0.50	0.40	0.02	0.91	0.91	0.91	0.06	0.93	0.87	0.81
		SVM	0.00	0.00	0.00	0.00	0.50	0.40	0.02	0.89	0.94	0.93	0.06	0.94	0.88	0.83
5	Calendar	RF	1.00	0.94	0.97	0.00	0.97	0.98	1.00	1.00	1.00	1.00	0.00	1.00	1.00	1.00
		Resnet	0.90	0.53	0.67	0.08	0.72	0.75	0.66	0.75	0.88	0.81	0.42	0.73	0.85	0.50
		SVM	0.90	0.53	0.67	0.08	0.72	0.75	0.66	1.00	0.88	0.94	0.00	1.00	1.00	0.94
6	Camera	RF	0.57	0.60	0.59	0.20	0.70	0.46	0.47	0.90	0.96	0.93	0.05	0.96	0.88	0.85
		Resnet	0.39	0.56	0.46	0.39	0.59	0.35	0.28	0.84	0.88	0.86	0.07	0.90	0.77	0.71
		SVM	0.00	0.00	0.00	0.00	0.50	0.30	0.00	0.90	0.96	0.93	0.05	0.96	0.88	0.85
7	Connectbot	RF	0.80	0.76	0.78	0.15	0.80	0.71	0.71	1.00	1.00	1.00	0.00	1.00	1.00	1.00
		Resnet	0.45	0.46	0.45	0.46	0.50	0.45	0.26	1.00	0.97	0.99	0.00	0.99	0.99	0.98
		SVM	0.65	0.54	0.61	0.47	0.43	0.04	0.25	0.92	0.93	0.93	0.06	0.98	0.97	0.88
8	Contacts	RF	0.89	1.00	0.94	0.06	0.97	0.89	0.95	0.89	1.00	0.94	0.06	0.89	0.92	0.85
		Resnet	0.31	1.00	0.47	1.00	0.50	0.31	0.19	0.80	1.00	0.89	0.11	0.94	0.80	0.76
		SVM	0.54	0.88	0.67	0.33	0.77	0.51	0.55	0.89	1.00	0.94	0.06	0.89	0.92	0.85
9	CoolReader	RF	0.83	0.86	0.84	0.20	0.83	0.79	0.77	0.99	0.96	0.97	0.01	0.97	0.97	0.94
		Resnet	0.61	0.84	0.71	0.62	0.61	0.60	0.48	0.98	0.96	0.97	0.03	0.97	0.96	0.93
		SVM	0.63	0.77	0.69	0.52	0.62	0.61	0.49	0.98	0.96	0.97	0.03	0.97	0.96	0.93
10	DeskClock	RF	0.83	0.68	0.75	0.06	0.81	0.67	0.71	0.98	0.96	0.97	0.01	0.97	0.95	0.94
		Resnet	0.46	0.53	0.49	0.29	0.62	0.39	0.35	0.98	0.91	0.94	0.01	0.95	0.92	0.89
		SVM	0.00	0.00	0.00	0.00	0.50	0.32	0.00	0.97	0.97	0.97	0.01	0.97	0.94	0.93
11	Email	RF	0.00	0.00	0.00	0.00	0.50	0.42	0.03	1.00	1.00	1.00	0.00	1.00	1.00	1.00
		Resnet	0.41	1.00	0.60	1.00	0.50	0.42	0.28	1.00	0.98	0.99	0.00	0.99	0.99	0.98
		SVM	0.00	0.00	0.00	0.00	0.50	0.42	0.03	1.00	1.00	1.00	0.00	1.00	1.00	1.00
12	FBReaderJ	RF	0.80	0.82	0.81	0.20	0.81	0.74	0.73	1.00	1.00	1.00	0.00	1.00	1.00	1.00
		Resnet	0.47	0.38	0.42	0.41	0.48	0.48	0.25	1.00	0.98	0.99	0.00	0.99	0.99	0.98
		SVM	0.64	0.86	0.74	0.47	0.70	0.62	0.56	0.99	1.00	0.99	0.01	0.99	0.99	0.98
13	K9Mail	RF	0.90	0.83	0.86	0.07	0.88	0.82	0.84	0.99	0.99	0.99	0.01	0.99	0.98	0.98
		Resnet	0.71	0.27	0.39	0.08	0.59	0.51	0.39	0.46	1.00	0.63	0.90	0.55	0.46	0.00
		SVM	0.00	0.00	0.00	0.00	0.50	0.43	0.03	0.99	0.99	0.99	0.01	0.99	0.98	0.98
14	KeePassAndroid	RF	0.86	0.83	0.84	0.07	0.88	0.77	0.81	0.98	0.97	0.97	0.01	0.98	0.96	0.95
		Resnet	0.43	0.71	0.54	0.45	0.63	0.40	0.36	0.95	0.95	0.96	0.01	0.97	0.95	0.92
		SVM	0.57	0.55	0.56	0.20	0.68	0.68	0.50	0.98	0.97	0.97	0.01	0.97	0.95	0.94
15	MMS	RF	0.52	0.89	0.65	0.82	0.54	0.51	0.37	1.00	1.00	1.00	0.00	1.00	1.00	1.00
		Resnet	0.51	0.49	0.50	0.46	0.51	0.50	0.31	1.00	1.00	1.00	0.00	1.00	1.00	1.00
		SVM	0.50	1.00	0.66	1.00	0.50	0.50	0.33	0.99	0.99	0.99	0.01	0.99	0.99	0.98
16	Xwords	RF	0.86	0.87	0.87	0.16	0.86	0.82	0.82	1.00	1.00	1.00	0.00	1.00	1.00	1.00
		Resnet	0.61	0.92	0.74	0.65	0.64	0.61	0.51	0.98	0.95	0.96	0.03	0.96	0.98	0.93
		SVM	0.75	0.67	0.70	0.25	0.71	0.68	0.60	1.00	0.93	0.96	0.00	0.99	0.99	0.95
17	QuickSearchBox	RF	0.65	0.74	0.69	0.08	0.83	0.53	0.67	0.95	0.87	0.91	0.01	0.93	0.85	0.96
		Resnet	0.50	0.04	0.08	0.01	0.52	0.19	0.16	0.95	0.87	0.91	0.01	0.93	0.85	0.96
		SVM	0.00	0.00	0.00	0.00	0.50	0.18	0.00	0.95	0.87	0.91	0.01	0.93	0.85	0.96

6.1.1 Experiment Design and Hypotheses

To answer the research questions, experiments were performed as a singular project experiment, then as cross-validation experiments. The singular project experiments consist of training and testing using the same project. The cross-validation was performed in three different ways:

1. Training-One-Predict-Multiple consists of training with one project and testing on multiple.

2. Training-Multiple-Predict-One, training with many projects and testing on one.

3. Cross-domain validation, where the projects are divided according to their domains. The projects from the same domain are used to train a model. This model is used to predict the vulnerabilities in projects from a different domain.

The metric described in Section 5.2 are used to calculate the z values. The z_i value is used to compute the p-value from a *Wilcoxon Sign-Ranked Test* for each comparison. When $\alpha \geq 0.05$, the hypothesis is accepted to be significantly different. Otherwise, the hypothesis is rejected. Table 10 shows the different p-values obtained for each hypothesis. A detailed analysis is presented in the following sections.

Table 10: p-value obtained from the *Wilcoxon Test* for the 10 hypotheses

	Tokenization	Embeddings			Architectural metrics			Models		
Hypothesis	1	2	3	4	5	6	7	8	9	10
p-value	0.15	4.6e−15	2.9e−12	0.4	1.9e−08	0.26	3.06e−06	8.08e−07	4.4e−11	3.9e−06
Conclusion	Reject	Accept	Accept	Reject	Accept	Reject	Accept	Accept	Accept	Accept

6.1.2 Experiment Results for Tokenization (RQ1)

Observation: These experiments aim to observe whether tokenization with removing code comments and/or special symbols may improve or create noise for the detection. The experiments are run with the tokens, including the comments and symbols (Table 7) and compared to tokens without them (Table 8). Each table shows the learning scores of 153 experiments (= (3 models × 3 embeddings) per project × 17 projects). Each experiment produces six scores that compute the value of z. Totally 306 (= 153 × the two tokenization techniques) data points of z are used to compute p-value from a *Wilcoxon Test*. The p-value is compared to the significance level α.

Hypothesis Analysis: Hypothesis (1) is defined as follows:

1. There is a statistically significant difference between the results obtained from using all tokens vs. using tokens without comments and symbols.

According to Table 10, the p-value obtained for hypothesis (1) is less than the significance level 0.05. This hypothesis is then rejected, which means there is no statistically significant difference between the two tokenization strategies.

Conclusion: Comments and symbols do not affect the learning of software vulnerabilities from the source code in the experiments.

6.1.3 Experiment Results for Feature Extraction (RQ2)

Observation: Feature extraction techniques convert tokens into a vector of features. In this experiment, the goal is to observe the effects of three feature extraction techniques, including (1) bag-of-words, (2) word2vec embedding and (3) fastText. the experiments are run with features obtained from bag-of-words (Table 7 and Table 8). For each embedding technique, there are 102 experiments (= (3 models $\times$ 2 tokenization methods) per project $\times$ 17 projects). Each experiment produces six scores that compute the value of z. Totally 306 data points of z are used to compute p-value from the *Wilcoxon Test*. The p-value is compared to the significance level α.

Hypothesis Analysis: To compare those three vector representation techniques, the following hypotheses are considered:

2. There is a statistically significant difference between the results obtained from using bag-of-words and word2vec as embeddings.

3. There is a statistically significant difference between the results obtained from using bag-of-words and fastText as embeddings.

4. There is a statistically significant difference between the results obtained from using word2vec and fastText as embeddings.

According to the Table 10 the p-values obtained from *Wilcoxon Test* of hypothesis (2) and hypothesis (3) are accepted, but hypothesis (4) is rejected. This means there is a statistically significant difference between bag-of-words and word2vec and also between bag-of-words and fastText. However, there is no statistically significant difference between word2vec and fastText. Additionally, the results obtained from the classification show that, on average, the precision and recall of the experiments with

bag-of-words are 6% more than the performance of the other embedding methods. That indicates that bag-of-words is better than the other two models in the learning process of vulnerabilities in the experiments.

Conclusion: Bag-of-words is the best way to generate embeddings for the remainder of the experiments.

6.1.4 Experiment Results using Architectural Metrics (RQ3)

Observation: In addition to the NLP-based method, architectural metrics are used as structural features to learn vulnerability patterns from the software repositories. The features are compared with code tokens only. The experiments use the architecture metrics only and are used to compare them to the architecture metrics with the bag-of-words features. The Table 9 shows the learning score of 51 (= 3 models × 17 projects) experiments for each feature used. Totally 102 data points of z are used to compute p-value from the *Wilcoxon Test*. The p-value is compared to the significance level α.

Hypothesis Analysis: The effects of using architecture metrics, extracted from the structures of the project, are explored via three hypotheses:

5. There is a statistically significant difference between 1) the results obtained from using tokens vs. 2) the results obtained from using the architectural metrics.

6. There is a statistically significant difference between 1) the results obtained from only using tokens vs. 2) the results obtained from using the combination of architectural metrics and tokens.

7. There is a statistically significant difference between 1) the results obtained from only using the architectural metrics vs. 2) the results obtained from using the combination of tokens and architectural metrics.

As shown in Table 10, the p-values indicate hypotheses (5) and (7) are accepted, while hypothesis (6) is rejected. This means there is no significant difference when using tokens as input features with or without architectural metrics. Hypotheses (5) and (7) further indicate that the tokens have a stronger influence on the learning performance than the architectural metrics.

Conclusion: The tokens without architectural metrics are used for the remainder of the experiments.

6.1.5 Experiment Results on Classification models (RQ4)

Observation: Identification of whether a certain machine learning model produces better vulnerability detection. The experiments are performed with each of the three models to compare them (Table 7 and Table 8). For each model, the table shows 102 experiments (= (2 tokenization methods × 3 embeddings) per project × 17 projects). Each experiment produces six scores that compute the value of z. Totally 306 data points of z are used to compute p-value from the *Wilcoxon Test*. The p-value is compared to the significance level α.

Hypothesis Analysis: To compare the models, these three hypotheses are defined:

8. There is a statistically significant difference between the performance of the random forest model and the SVM.

9. There is a statistically significant difference between the performance of the random forest model and the ResNet.

10. There is a statistically significant difference between the performance between the SVM and the ResNet.

According to Table 10, the p-values of the three hypotheses (8), (9), and (10) are less than the significance level, α. All three hypotheses are accepted. Overall the random forest model performs better than the SVM and ResNet in most experiments with a precision and recall higher by an average of 8%.

Conclusion: The random forest is used as the model to learn the patterns of vulnerabilities in the cross-project validation experiments.

6.2 Cross Validation

This evaluation explores the answer to the question *"How transferable is the learning method in predicting vulnerabilities in new projects?"*. Two sets of experiments are defined to investigate this question.

6.2.1 Train-One-Predict-Multiple

In this test, a learning model is trained with source code from a single project and then tested on other projects. Its learning performance is compared with existing work [19]. The 15 projects used in this experiment overlap with those used in [19]. The same score as in [19] is used to evaluate the learning performance: the number of projects with the classification metrics of precision and recall with a certain threshold is counted. Table 11 reports comparison between this work and learning with LSTM models. With the threshold value of 0.7, the results of this work are comparable to the results in [19]. With a threshold of 0.8, the results with the Random Forest degrade to the average value of 1.4 projects with both a precision and a recall equal to or greater than 80%.

Table 11: Training-One-Predicting-Multiple compared with the LSTM model in [19] with the threshold value settings.

	Projects	Random Forest (precision ≥70%, recall ≥ 70%)	Random Forest (precision≥ 80%, recall ≥ 80%)	LSTM[1] [19] (precision≥80%, recall ≥ 80%)
1	Camera	7	4	6
2	FBReader	6	3	6
3	Mms	6	2	6
4	Contacts	6	2	2
5	KeePassDroid	6	2	4
6	ConnectBot	6	2	5
7	AnkiDroid	5	1	5
8	Email	5	0	4
9	Crosswords	4	1	1
10	Browser	4	1	1
11	Coolreader	4	1	6
12	Calendar	4	0	5
13	K9	3	2	8
14	DeskClock	0	0	1
15	QuickSearchBox	0	0	3

6.2.2 Train-Multiple-Predict-One

To further improve the learning performance, a 15-fold cross-validation is conducted by choosing 14 projects from the same domain of the Android project for training. The remaining project is reserved for testing. Table 12 contains the cross-validation results, ordered by precision and recall values. 5 out of the 15 experiments have both precision and recall values equal to or greater than 80%; 10 out of 15 experiments

have both precision and recall equal to or greater than 70%. Referring to Table 5, the five experiments with precision and recall below 70% have the ratio of vulnerable files below 40%.

Referring to Table 12, cross-project validation improves the learning performance under the threshold of 80% to 5 projects out of 15 projects. This approach of transfer learning, in which the features are combined from the Android project repository to tune the random forest model, achieves comparable learning performance to the deep learning models ResNet and LSTM [19] (4.2 projects out of 15 projects).

Table 12: The cross project validation from 15 Android projects, with 5 projects having both precision and recall higher than 80% (ConnectBot, Email, Coolreader, Crosswords, AnkiDroid)

Projects		P	R	F1	FPR	ROC AUC	PR AUC
1	**ConnectBot**	**0.90**	**0.86**	0.88	0.08	0.89	0.84
2	**Email**	**0.90**	**0.81**	0.85	0.10	0.86	0.83
3	**Coolreader**	**0.88**	**0.82**	0.85	0.11	0.86	0.81
4	**Crosswords**	**0.81**	**0.87**	0.84	0.14	0.86	0.76
5	K9	0.94	0.60	0.74	0.05	0.78	0.78
6	**AnkiDroid**	**0.81**	**0.86**	0.83	0.29	0.78	0.78
7	**Calendar**	**0.75**	**0.88**	0.81	0.24	0.82	0.71
8	**Camera**	**0.74**	**0.87**	0.80	0.32	0.77	0.71
9	**FBReader**	**0.73**	**0.71**	0.72	0.11	0.80	0.61
10	Contacts	0.69	0.92	0.79	0.39	0.77	0.67
11	KeePassDroid	0.64	0.90	0.75	0.34	0.78	0.62
12	Deskclock	0.64	0.88	0.74	0.33	0.77	0.61
13	Browser	0.70	0.70	0.70	0.17	0.76	0.60
14	Mms	0.68	0.70	0.70	0.20	0.76	0.59
15	QuickSearchBox	0.45	0.93	0.60	0.46	0.74	0.44

6.2.3 Cross Domain Validation

A final set of cross-validation experiments is performed. These one consider the transferability across the domain. The three datasets presented in this paper—OWASP, Juliet, and Android—are from different domains. According to this, three models are trained, one with each domain. The models are used to predict projects from other domains. Table 13 shows the results of these experiments. Table 13 shows the learning performance has degraded. A key contributing factor is the disparateness of vulnerability signatures. The previous discussion of the vulnerable files and types in Table 2, Table 3 and Table 5 show this heterogeneity.

Table 13: Cross domain comparison to observe how transferable the vulnerability signature is

Train \ Predict	Juliet	OWASP	Android
Juliet	Table 7	P: 0.54 R: 0.77	P: 0.44 R: 0.53
OWASP	P: 0.4 R: 0.8	Table 7	1 out of 15 project (precision and recall greater than 0.7) P: 0.74 R: 0.39
Android	P: 0.4 R: 0.46	P: 0.49 R: 0.64	Table 7

6.3 Discussion

The approach in this thesis consists of comparing the different factors that contributed to the detection of vulnerabilities in source code. Tables 7, 8 and 9 show the results of the 408 experiments performed to evaluate the classification on three domains of the source code.

These tables and the statistical test performed in section 6.1.1 demonstrate 95% of the learning metrics are above 0.77 after over 400 experiments. The tokenization choice, which consists of removing the comments or not, shows that the comments and symbols do not affect the learning of the vulnerabilities by the model. Using the architectural metrics as features, in this case, has no significant improvement on the learning of vulnerabilities. One reason is that the complexity of the code and its dependencies are not captured by the tokens only. As a result, a baseline emerges with feature representations extracted through bag-of-words embedding and using the random forest model. This baseline increases the accuracy by about 4% compared to other combinations of examined factors.

Further cross-validation experiments were conducted to observe how transferable across domains the vulnerability signatures are. The training of a single project and predicting multiple projects method achieves an average of 4.4 projects with a precision and recall higher than 70%. With the 15-fold cross-validation method of training multiple projects and predicting one project, the baseline model slightly outperforms the LSTM model [19] with a proprietary embedding method.

Chapter 7

Use of the Baseline Model

In the following sections, the baseline model is the random forest combined bag-of-words since it is the model that achieves the best performances in the previous experiments. In this Chapter, the baseline model established in the previous Chapter is used to evaluate the performances of a BiLSTM model with word2vec as embedding to transform the source code.

7.1 Purpose of a Baseline Model

A baseline model serves as a reference point to compare the performances of other models that are usually more complex. Experimenting with a baseline model is usually quick, and low cost, so the model has to be simple to set up and has a reasonable chance of providing decent results. It is commonly used as in the artificial intelligence community [72, 25].

In this case, the baseline model relies on understanding the key factors contributing to the discovery of vulnerability signatures through a combination of techniques and machine learning models. A baseline model can be used to evaluate the classification performance of another model in learning the vulnerability signatures. Such a model helps to establish a base to investigate techniques on feature representation, learning models, factors such as code structure, and complexity in learning vulnerability patterns.

According to Chapter 6, the baseline model is established according to the combination of factors that show the best performance in learning the vulnerability signatures. The model that shows the best performance is the Random Forest model that takes as input source code files with all the tokens transformed into numerical values using bag-of-words. This chapter aims to use the predefined baseline model to observe how well a model performs. A Bidirectional Long Short-Term Memory model is built to perform the classification task, and the results are compared to the baseline.

7.2 Bidirectional Long Short-Term Memory for Vulnerability Detection

For this part of the thesis, a Bidirectional Long Short-Term Memory (BiLSTM) model is used. A BiLSTM consists of two Long Short-Term Memory (LSTM) models [64]. LSTM has been shown to be effective in learning the structure of source code. Karpathy et al.[36] train an LSTM to predict the next character in a sample sequence of characters from the Linux source code, then look for interpretable activations within the LSTM. They found that the LSTM model has the ability to understand the underlying structure of statements for loops, while loops, functions, and nesting, among other things.

An LSTM[28] model is a type of recurrent neural network (RNN) that is able to capture long-term dependencies. It consists of a sequence of connected cells where each cell takes a vector as input and outputs a vector state. Since the code files in the dataset can relatively be long sequences of text compared to other common NLP tasks, a BiLSTM is used for this work.

A traditional LSTM goes through a sequence forward, so it can only see each input once and from one direction. That can lead to the loss of important information at the beginning of long sequences by the time the LSTM reaches the end of the sequence. BiLSTMs solve this problem by running two separate LSTMs, one that goes through the sequence from start to end and another that goes backward through the sequence from end to start.

The model in this work is constructed of one embedding layer that is responsible for converting the tokens into their vector representation, one BiLSTM layer that learns

the context of words and carries the contextual meaning, one convolution layer that creates a convolution kernel, one global max-pooling operation that down-samples the input representation by taking the maximum value over different dimensions, and finally one dense. The last dense layer has a sigmoid activation to transform the input to a number between 0 and 1. The detailed structure is depicted in Figure 8.

The embedding layer provides the numerical representation of the tokens and their meaning according to the context. The word's context is learned from a corpus of code created and used to train an embedding model. Here, to initialize the embedding layer, a pre-trained word2vex model is used (More details are in section 4.3). Using this embedding model allows saving time in training the classification model.

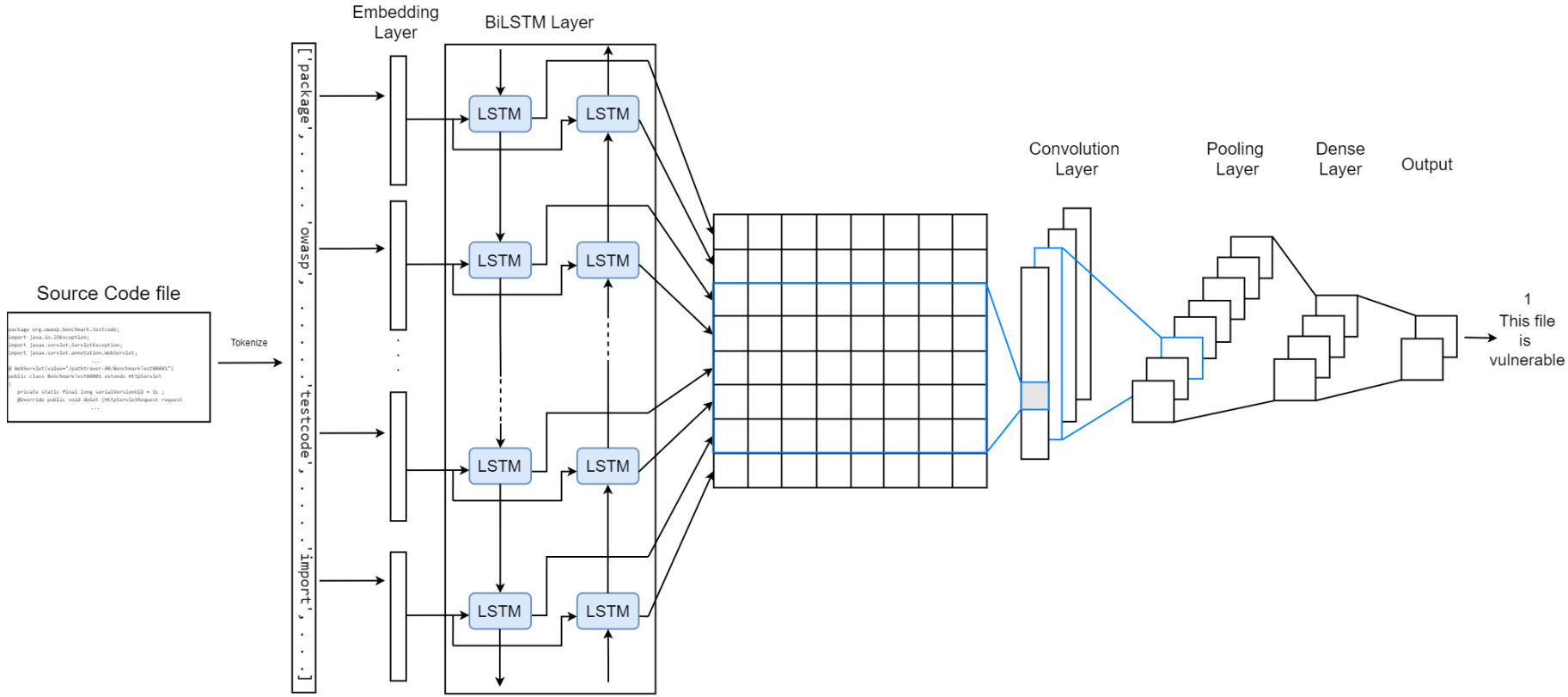

Figure 8: The data flow for the classification task performed by the BiLSTM model. This figure shows the data flow of the feature engineering and learning for the BiLSTM model. It illustrates the different layer that defines the model in this thesis.

7.3 Comparing the BiLSTM to the Baseline model

7.3.1 Singular Project Vulnerability Detection

In this section, singular project training experiments are performed. The model is trained and tested using the same project. The experiments are run using all the tokens of the files, including comments and symbols, and the embedding layer of the BiLSTM model is initialized with a pre-trained word2vec model.

Observation: These experiments aim to observe how well the BiLSTM performs compared to the baseline model. Table 14 shows the results of the 17 experiment. Each experiment produces six scores that are used to compute the z values. The value is calculated according to all the experiments. The higher the value, the better the results. If z equals 1, the experiment performed the best among all. If it is equal to 0, it performed the worst.

Analysis of the results: According to Table 14, overall, the performance of the two models are comparable. For 10 out of 17 projects, the BiLSTM precision and recall are higher than 80%. However, the average of the z for the baseline model is equal to 0.91, compared to 0.69 for the BiLSTM model. For Juliet, the z value equals 0; it is the project that performs the worst compared to the other experiments (according to both baseline and BiLSTM). Juliet has the lowest number of tokens and the project with the more diversified vulnerability types (more than 12 according to Table 3). The model has trouble learning the signature of each type.

Conclusion: The baseline model performs better than the BiLSTM model associated with word2vec in the case of singular project vulnerability detection.

Table 14: Singular project vulnerability detection results. Comparison of the baseline model (Random Forest + Bag-of-words) with the BiLSTM model (BiLSTM + Wor2Vec). The z value in the table aggregate the evaluation metrics. The value is calculated according to all the experiments. The higher the value, the better the results. If z equals 1, the experiment performed the best among all. If it is equal to 0, it performed the worst.

	Projects	Baseline							BiLSTM						
		P	R	F1	FPR	ROC AUC	PR AUC	z	P	R	F1	FPR	ROC AUC	PR AUC	z
1	OWASP	1.00	1.00	1.00	0.01	1.00	1.00	1.00	**0.85**	**0.92**	0.88	0.19	0.87	0.82	0.68
2	Juliet	1.00	1.00	1.00	0.00	1.00	1.00	1.00	1.00	0.08	0.14	0.00	0.54	0.54	0.00
3	Anki-Android	0.80	0.89	0.84	0.21	0.84	0.76	0.60	**1.00**	**0.89**	0.94	0.00	0.94	0.96	0.90
4	Connectbot	1.00	1.00	1.00	0.00	1.00	1.00	1.00	**1.00**	**0.94**	0.97	0.00	0.97	0.99	0.95
5	CoolReader	1.00	0.98	0.99	0.00	1.00	1.00	0.99	**0.97**	**0.97**	0.97	0.02	0.98	0.96	0.94
6	Xwords	1.00	1.00	1.00	0.00	1.00	1.00	1.00	**0.97**	**0.99**	0.98	0.03	0.98	0.96	0.94
7	FBReaderJ	0.96	0.93	0.94	0.02	0.98	0.98	0.92	**0.88**	**0.96**	0.92	0.05	0.95	0.85	0.82
8	K9Mail	0.97	0.99	0.98	0.02	1.00	1.00	0.97	**0.96**	**0.99**	0.98	0.04	0.98	0.96	0.93
9	KeePassAndroid	0.99	1.00	1.00	0.00	1.00	1.00	0.99	**1.00**	**1.00**	1.00	0.00	1.00	1.00	1.00
10	Camera	0.94	0.91	0.93	0.04	0.94	0.89	0.84	0.78	0.75	0.75	0.17	0.81	0.68	0.49
11	Browser	0.97	0.93	0.95	0.02	1.00	1.00	0.94	0.74	0.76	0.75	0.16	0.80	0.65	0.45
12	Calendar	0.87	0.87	0.89	0.08	0.92	0.82	0.74	0.88	0.70	0.78	0.05	0.83	0.71	0.57
13	Contacts	0.90	0.96	0.93	0.05	0.96	0.88	0.84	0.68	0.68	0.68	0.15	0.77	0.56	0.34
14	DeskClock	0.89	1.00	0.94	0.06	0.97	0.89	0.87	0.73	0.92	0.82	0.29	0.82	0.71	0.52
15	Email	0.97	0.98	0.97	0.04	0.99	0.99	0.95	**0.89**	**0.92**	0.90	0.12	0.90	0.86	0.76
16	MMS	0.98	0.97	0.98	0.01	1.00	0.99	0.96	**0.91**	**0.91**	0.91	0.05	0.93	0.86	0.80
17	QuickSearchBox	0.95	0.87	0.91	0.01	0.93	0.85	0.82	0.86	0.73	0.79	0.03	0.85	0.69	0.59
	Average	0.95	0.96	0.96	0.03	0.97	0.94	0.91	0.89	0.83	0.83	0.08	0.88	0.81	0.69

7.3.2 Cross Validation

The cross-validation experiments are separated into three sets of experiments: Train-One-Predict-Multiple, Train-Multiple-Predict-One and finally, the cross-domain validation. These are conducted to further analyze the transferability of the learned vulnerability signatures of the BiLSTM compared to the baseline model.

Train-One-Predict-Multiple Results

Observation and Analysis: In this test, a learning model is trained with source code from a single project and then tested on other projects. The learning performance is compared to the baseline model. 15 models are built, each trained with one of the Android projects. The number of tested projects that have a good performance is counted. The model has good performance if the precision and recall are higher than or equal to 0.7. Table 15 reports the comparison between the baseline model and the BiLSTM model.

The prediction results of the baseline model are better than the BiLSTM. The average value of 2.13 for the BiLSTM and 4.28 for the baseline is twice as good. The only case where BiLSTM performs better than the baseline model is when trained

with the QuickSearchBox project files.

Conclusion: The baseline outperforms the BiLSTM model in predicting vulnerabilities in files from other projects in the case of Train-One-Predict-Multiple.

Table 15: Training-One-Predicting-Multiple. Comparison of the baseline model (Random Forest + BOW) to BiLSTM model (BiLSTM + W2V) with the threshold of 0.7. The table contains the number of projects with precision and recall higher or equal to 70% that each model trained with only one project predicted.

Projects		Baseline (P $\geq$70%, R $\geq$ 70%)	BiLSTM (P $\geq$70%, R $\geq$ 70%)
1	Camera	7	3
2	FBReaderJ	6	0
3	Contacts	6	5
4	KeePassAndroid	6	1
5	Connectbot	6	4
6	Anki-Android	5	3
7	Email	5	1
8	Xwords	4	3
9	Browser	4	1
10	CoolReader	4	2
11	Calendar	4	3
12	K9Mail	4	3
13	DeskClock	3	1
14	MMS	0	0
15	QuickSearchBox	0	**2**
Average		4.27	2.13

Train-Multiple-Predict-One Results

Observation and Analysis: As an attempt to improve the learning performance, multiple projects are used for training the models. The experiments consist of 15-fold cross-validation using the 15 projects of the android study. Table 16 reports the experiments results. By comparing the z values averages, the baseline performs better with 0.64 compared to 0.3 for the BiLSTM. For the BiLSTM, 4 out of the 15 experiments have both precision and recall values equal to or greater than 70%. In comparison to 7 out of 15 for the baseline model. This indicates that the baseline model generalizes better than the BiLSTM model on these datasets.

Conclusion: In this case and training and testing using the Android project

dataset, the baseline model outperforms the BiLSTM.

Table 16: Training-Multiple-Predicting-One. Comparison of the baseline model (Random Forest + BOW) to BiLSTM model (BiLSTM + W2V). Each model is trained with all the projects minus one. The table contains the results of predicting the project that was not used in the training with the model trained with all the other projects. The z value in the table aggregate the evaluation metrics. The value is calculated according to all the experiments. The higher the value, the better the results. If z equals 1, the experiment performed the best among all. If it is equal to 0, it performed the worst.

	Projects	Baseline							BiLSTM						
		P	R	F1	FPR	ROC AUC	PR AUC	z	P	R	F1	FPR	ROC AUC	PR AUC	z
1	Anki-Android	0.81	0.86	0.83	0.29	0.78	0.78	0.74	0.80	0.67	0.73	0.13	0.77	0.68	0.61
2	Connectbot	0.90	0.86	0.88	0.08	0.89	0.84	1.00	**0.70**	**0.70**	0.70	0.29	0.72	0.63	0.42
3	CoolReader	0.88	0.82	0.85	0.11	0.86	0.81	0.91	**0.71**	**0.83**	0.77	0.27	0.78	0.67	0.60
4	Xwords	0.85	0.73	0.78	0.11	0.81	0.74	0.75	0.82	0.60	0.69	0.10	0.75	0.66	0.56
5	FBReaderJ	0.72	0.68	0.70	0.11	0.78	0.59	0.53	0.53	0.68	0.60	0.27	0.70	0.46	0.20
6	K9Mail	0.92	0.82	0.87	0.07	0.88	0.85	0.99	**0.79**	**0.72**	0.75	0.23	0.75	0.72	0.60
7	KeePassAndroid	0.72	0.55	0.63	0.14	0.71	0.58	0.38	0.61	0.34	0.44	0.15	0.60	0.48	0.00
8	Camera	0.77	0.60	0.67	0.08	0.76	0.58	0.50	0.78	0.62	0.69	0.21	0.72	0.68	0.49
9	Browser	0.70	0.69	0.70	0.17	0.76	0.60	0.49	0.60	0.67	0.63	0.35	0.70	0.53	0.24
10	Calendar	0.74	0.82	0.78	0.22	0.80	0.68	0.65	0.58	0.65	0.61	0.38	0.63	0.53	0.16
11	Contacts	0.66	0.71	0.68	0.17	0.77	0.55	0.45	0.64	0.54	0.58	0.29	0.63	0.57	0.18
12	DeskClock	0.91	0.50	0.65	0.04	0.73	0.69	0.57	**0.75**	**0.75**	0.75	0.17	0.79	0.66	0.62
13	Email	0.90	0.81	0.85	0.10	0.86	0.83	0.93	0.81	0.67	0.74	0.20	0.73	0.73	0.56
14	MMS	0.67	0.67	0.70	0.22	0.76	0.59	0.44	0.61	0.85	0.71	0.33	0.76	0.57	0.44
15	QuickSearchBox	0.79	0.41	0.54	0.03	0.69	0.46	0.28	0.50	0.57	0.53	0.23	0.67	0.41	0.08
	Average	0.80	0.70	0.74	0.13	0.79	0.68	0.64	0.68	0.66	0.66	0.24	0.71	0.60	0.39

Cross Domain Validation

These cross-domain validation experiments are performed to evaluate the transferability across the domain of the learning. The three datasets presented in this paper—OWASP, Juliet, and Android—are from different domains. According to this, three models are trained, one with each domain. The three models are used to predict each of the domains. Table 17 shows the results of these experiments. The learning performance has degraded. The learning performance has degraded due to the disparateness of vulnerability signatures.

Table 17: Cross domain comparison to observe how transferable the vulnerability signature is for the BiLSTM model

Train \ Predict	Juliet	OWASP	Android
Juliet	Table 14	P: 0.50 R: 0.50	P: 0.13 R: 0.44
OWASP	P: 0.42 R: 0.94	Table 14	P: 0.51 R: 0.70
Android	P: 0.42 R: 0.73	P: 0.53 R: 0.74	Table 14

7.4 Discussion

In this chapter, the baseline model is used to evaluate the performance of a more complex model, a BiLSTM, in detecting vulnerabilities. In Table 14 that contains the results for the singular project detection, 10 out of 17 projects have precision and recall above 80%. Moreover, in 90% of the cases, those metrics are above 70%. The results are comparable to the baseline model. However, the baseline model outperforms the BiLSTM by an average precision and recall of 10%. The results presented in cross-validation (Table 15 and Table 16) shows that the baseline model performs better in most of the cases. In training a single project and predicting multiple experiments, the baseline model achieves twice as well as the BiLSTM. Moreover, the 15-fold cross-validation, the baseline model, performs better on three more projects than the BiLSTM.

Chapter 8

Threats to Validity

This section summarizes the different threats to validity, including the datasets and limited architectural metrics relative to internal validity; and the domains of experiments relative to external validity. In the following, some threats to the validity of this study are discussed.

Regarding the threats to **external validity**, the study is limited to specific projects, and thus, the results in this work might not be generalizable to other projects or contexts. In this section, further Cross-Domain Validation experiments are performed to evaluate if the model is transferable to other projects and if the obtained results are generalizable over projects from different domains. The study is limited to 17 java projects. Fifteen real Android applications are used; however, the results might not be generalizable to other projects or contexts.

Cross-domain validation means training a model on datasets from one domain and predicting vulnerabilities on datasets from a different domain. The three datasets presented in this work—OWASP, Juliet, and Android—are from different domains. The previous discussion of the vulnerable files and types in Table 2, Table 3 and Table 5 show this heterogeneity.

Table 13 and Table 17 show the learning performance has degraded. A key contributing factor is the disparateness of vulnerability signatures. The cross-domain validation is also limited because it only assesses three different domains.

Regarding **construct validity**, the choice was to use publicly available datasets that were previously labelled. The OWASP dataset and the Juliet dataset contain both source code files and vulnerability labels. The Android Study dataset only

includes information on the tagged file but without the source code files. The source code files have to be retrieved according to the file names and project versions. In addition to that, the vulnerable code labels for the Android Study project followed the data in [63]. The labels are determined by Fortify [21]. It has been recognized that static code analysis tools may contain false-positive labels. In the literature, [56, 77] path and commit data have been mined to identify vulnerable code. In the security development and operation process, this was addressed by manual correction. This work focuses on the factors that contribute to creating a baseline and thus assumes that the labels are of stable quality.

Regarding **internal validity**, this work only considers source code files written in Java because many C++ source codes lack data labels. The number of projects examined for transferability is still limited to reach a statistically significant conclusion. Also, the token-based feature representation is considered a flattened structure. Such a token-based feature representation is combined with aggregated architectural metrics. The architecture metrics have not contributed significantly to the learning, which indicates either the current learning representation has not utilized the architectural metrics in the optimal embedding or other kinds of learning models should be applied to architectural metrics. This remains further research.

Chapter 9

Conclusion

The work in this thesis proposes to reveal the most contributing factors for detecting software vulnerabilities. The observations from 17 Java projects and over 400 experiments led to define a model that can be used as a baseline by comparing tokenization techniques, embedding methods, and machine learning models. The baseline model under a cross-validation training approach on the same project domain achieves comparable and slightly better learning performance than the models using deep learning networks. This provides the reference as the minimum learning performance that a future vulnerability detection approach should achieve. Further experiment to evaluate a BiLSTM using the established baseline model shows that the defined baseline model outperformed it. The observations show that cross-domain learning is subject to the extent of vulnerability signature disparateness. This work envisions a promising research direction that integrates transfer learning techniques to a software DevOps process and feeds target domain inputs to augment the training from the source domain.

Bibliography

[1] Ieee standard for a software quality metrics methodology. *IEEE Std 1061-1992*, pages 1–96, 1993.

[2] Acunetix. Acunetix web vulnerability scanner, .

[3] A. Alkussayer and W. H. Allen. A scenario-based framework for the security evaluation of software architecture. In *2010 3rd International Conference on Computer Science and Information Technology*, volume 5, pages 687–695, July 2010.

[4] A. Alkussayer and W. H. Allen. Security risk analysis of software architecture based on ahp. In *7th International Conference on Networked Computing*, pages 60–67, Sep. 2011.

[5] Mohamed Almorsy, John Grundy, and Amani S Ibrahim. Automated software architecture security risk analysis using formalized signatures. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 662–671. IEEE, 2013.

[6] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code, 2019.

[7] B. Alshammari, C. Fidge, and D. Corney. Security metrics for object-oriented designs. In *2010 21st Australian Software Engineering Conference*, pages 55–64, April 2010.

[8] Android. Android development platform - git repository, .

[9] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, Oct 1996.

[10] Len Bass, Paul Clements, and Rick Kazman. *Software architecture in practice*. Addison-Wesley Professional, 3 edition, 2012.

[11] C. Bidan and V. Issarny. Security benefits from software architecture. In David Garlan and Daniel Le Métayer, editors, *Coordination Languages and Models*, pages 64–80, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[12] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *arXiv preprint arXiv:1607.04606*, 2016.

[13] Fabian Brosig, Nikolaus Huber, and Samuel Kounev. Automated extraction of architecture-level performance models of distributed component-based systems. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 183–192. IEEE, 2011.

[14] Yuanfang Cai, Hanfei Wang, Sunny Wong, and Linzhang Wang. Leveraging design rules to improve software architecture recovery. In *Proceedings of the 9th international ACM Sigsoft conference on Quality of software architectures*, pages 133–142. ACM, 2013.

[15] Checkmarx. Checkmarx software security platform, .

[16] MITRE Corporation. The common weakness enumeration community, 2006.

[17] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Mach. Learn.*, 20(3):273–297, September 1995.

[18] Rory Coulter, Qing-Long Han, Lei Pan, Jun Zhang, and Yang Xiang. Data-driven cyber security in perspective—intelligent traffic analysis. *IEEE transactions on cybernetics*, 50(7):3081–3093, 2019.

[19] Hoa Khanh Dam, Truyen Tran, Trang Thi Minh Pham, Shien Wee Ng, John Grundy, and Aditya Ghose. Automatic feature learning for predicting vulnerable software components. *IEEE Transactions on Software Engineering*, 2019.

[20] Qiong Feng, Rick Kazman, Yuanfang Cai, Ran Mo, and Lu Xiao. Towards an architecture-centric approach to security analysis. In *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 221–230. IEEE, 2016.

[21] Fortify. Fortify, .

[22] Seyed Mohammad Ghaffarian and Hamid Reza Shahriari. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM Comput. Surv.*, 50(4):56:1–56:36, August 2017.

[23] Seyed Mohammad Ghaffarian and Hamid Reza Shahriari. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM Computing Surveys (CSUR)*, 50(4):1–36, 2017.

[24] Leilani H. Gilpin, David Bau, Ben Z. Yuan, Ayesha Bajwa, Michael Specter, and Lalana Kagal. Explaining explanations: An overview of interpretability of machine learning, 2019.

[25] Google. Machine learning glossary, .

[26] Jacob A Harer, Louis Y Kim, Rebecca L Russell, Onur Ozdemir, Leonard R Kosta, Akshay Rangamani, Lei H Hamilton, Gabriel I Centeno, Jonathan R Key, Paul M Ellingwood, et al. Automated software vulnerability detection with machine learning. *arXiv preprint arXiv:1803.04497*, 2018.

[27] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[28] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 11 1997.

[29] 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. ACM, 2012.

[30] Secure Software Inc. Rough audit tool for security, .

[31] K. A. Jackson and B. T. Bennett. Locating sql injection vulnerabilities in java byte code using natural language techniques. In *SoutheastCon 2018*, pages 1–5, April 2018.

[32] Smriti Jain and Maya Ingle. A review of security metrics in software development process. 2011.

[33] JetBrains. Jetbrains/intellij-community, Jun 2019.

[34] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. Bag of tricks for efficient text classification. *CoRR*, abs/1607.01759, 2016.

[35] Stefan Kals, Engin Kirda, Christopher Krügel, and Nenad Jovanovic. Secubat: a web vulnerability scanner. pages 247–256, 01 2006.

[36] Andrej Karpathy, Justin Johnson, and Li Fei-Fei. Visualizing and understanding recurrent networks. *CoRR*, abs/1506.02078, 2015.

[37] 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.

[38] Guanjun Lin, Sheng Wen, Qing-Long Han, Jun Zhang, and Yang Xiang. Software vulnerability detection using deep neural networks: A survey. *Proceedings of the IEEE*, 108(10):1825–1848, 2020.

[39] Liu Liu, Olivier De Vel, Qing-Long Han, Jun Zhang, and Yang Xiang. Detecting and preventing cyber insider threats: A survey. *IEEE Communications Surveys & Tutorials*, 20(2):1397–1417, 2018.

[40] Daniel Mellado, Eduardo Fernández-Medina, and Mario Piattini. A comparison of software design security metrics. In *ECSA '10*, 2010.

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

[42] Nikola Milosevic, Ali Dehghantanha, and Kim-Kwang Raymond Choo. Machine learning aided android malware classification. *Computers & Electrical Engineering*, 61:266 – 274, 2017.

[43] Ran Mo, Yuanfang Cai, Rick Kazman, Lu Xiao, and Qiong Feng. Decoupling level: A new metric for architectural maintenance complexity. In *2016 International Conference on Software Engineering (ICSE)*. IEEE, 2016.

[44] Muhammad Nadeem, Byron J. Williams, and Edward B. Allen. High false positive detection of security vulnerabilities: A case study. In *Proceedings of the 50th Annual Southeast Regional Conference*, ACM-SE '12, page 359–360, New York, NY, USA, 2012. Association for Computing Machinery.

[45] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 452–461, New York, NY, USA, 2006. ACM.

[46] Lakshmanan Nataraj, Shanmugavadivel Karthikeyan, Grégoire Jacob, and B. Manjunath. Malware images: Visualization and automatic classification. 07 2011.

[47] Netsparker. Netsparker web vulnerability scanner, .

[48] National Institute of Standards and Technology. Applications, 2015.

[49] National Institute of Standards and Technology. Juliet test suite for java v1.3, 2017.

[50] National Institute of Standards and Technology. Software assurance reference dataset, .

[51] National Institute of Standards and Technology. Vulnerability definition. Computer Security Resource Center, . [online] `https://csrc.nist.gov/glossary/term/vulnerability`.

[52] Pablo Oliveira Antonino, Slawomir Duszynski, Christian Jung, and Manuel Rudolph. Indicator-based architecture-level security evaluation in a service-oriented environment. pages 221–228, 01 2010.

[53] OWASP. Owasp benchmark project, 2018.

[54] Y. Pang, X. Xue, and A. S. Namin. Predicting vulnerable software components through n-gram analysis and statistical feature selection. In *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*, pages 543–548, Dec 2015.

[55] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[56] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 426–437, New York, NY, USA, 2015. ACM.

[57] PortSwigger. Burp suite web vulnerability scanner, .

[58] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen. Android study, 2014.

[59] Danijel Radjenović, Marjan Heričko, Richard Torkar, and Aleš Živkovič. Software fault prediction metrics: A systematic literature review. *Information and software technology*, 55(8):1397–1418, 2013.

[60] Radim Řehůřek and Petr Sojka. Gensim: Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta, May 2010. ELRA. http://is.muni.cz/publication/884893/en.

[61] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 757–762. IEEE, 2018.

[62] Adam Sachitano, Richard O Chapman, and JA Hamilton. Security in software architecture: a case study. In *Proceedings from the Fifth Annual IEEE SMC Information Assurance Workshop, 2004.*, pages 370–376. IEEE, 2004.

[63] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen. Predicting vulnerable software components via text mining. *IEEE Transactions on Software Engineering*, 40(10):993–1006, Oct 2014.

[64] Mike Schuster and Kuldip Paliwal. Bidirectional recurrent neural networks. *Signal Processing, IEEE Transactions on*, 45:2673 – 2681, 12 1997.

[65] Robert Schwanke, Lu Xiao, and Yuanfang Cai. Measuring architecture quality by structure plus history analysis. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 891–900. IEEE, 2013.

[66] Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE transactions on software engineering*, 37(6):772–787, 2010.

[67] Karsten Sohr and Bernhard Berger. Idea: Towards architecture-centric security analysis of software. In *International Symposium on Engineering Secure Software and Systems*, pages 70–78. Springer, 2010.

[68] Karsten Sohr and Bernhard Berger. Idea: Towards architecture-centric security analysis of software. In Fabio Massacci, Dan Wallach, and Nicola Zannone, editors, *Engineering Secure Software and Systems*, pages 70–78, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[69] Nan Sun, Jun Zhang, Paul Rimba, Shang Gao, Leo Yu Zhang, and Yang Xiang. Data-driven cybersecurity incident prediction: A survey. *IEEE communications surveys & tutorials*, 21(2):1744–1772, 2018.

[70] Tin Kam Ho. Random decision forests. In *Proceedings of 3rd International Conference on Document Analysis and Recognition*, volume 1, pages 278–282 vol.1, 1995.

[71] John Viega, Jon-Thomas Bloch, Yoshi Kohno, and Gary McGraw. Its4: A static vulnerability scanner for c and c++ code. In *Proceedings 16th Annual Computer Security Applications Conference (ACSAC'00)*, pages 257–267. IEEE, 2000.

[72] Sida Wang and Christopher Manning. Baselines and bigrams: Simple, good sentiment and topic classification. pages 90–94, 07 2012.

[73] D. A. Wheeler. Flawfinder, .

[74] Zhongxing Xu, Ted Kremenek, and Jian Zhang. A memory model for static analysis of c programs. In *Proceedings of the 4th International Conference on Leveraging Applications of Formal Methods, Verification, and Validation - Volume Part I*, ISoLA'10, pages 535–548, Berlin, Heidelberg, 2010. Springer-Verlag.

[75] Fabian Yamaguchi, Felix Lindner, and Konrad Rieck. Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning. In *Proceedings of the 5th USENIX Conference on Offensive Technologies*, WOOT'11, pages 13–13, Berkeley, CA, USA, 2011. USENIX Association.

[76] Yin Zhang, Rong Jin, and Zhi-Hua Zhou. Understanding bag-of-words model: A statistical framework. *International Journal of Machine Learning and Cybernetics*, 1:43–52, 12 2010.

[77] Yaqin Zhou and Asankhaya Sharma. Automated identification of security issues from commit messages and bug reports. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 914–919, New York, NY, USA, 2017. ACM.

[78] Thomas Zimmermann, Nachiappan Nagappan, and Laurie Williams. Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, pages 421–428, Washington, DC, USA, 2010. IEEE Computer Society.