MASTER'S THESIS

Detecting software vulnerabilities in source code and the influence of variable naming
Demonstrated for C# code and CODE2VEC

Mathijssen, D

Award date:
2022

Detecting software vulnerabilities in source code and the influence of variable naming

Demonstrated for C# code and code2vec

Davey Mathijssen

Date: 02/12/2022

Open Universiteit
www.ou.nl

DETECTING SOFTWARE VULNERABILITIES IN SOURCE CODE AND THE INFLUENCE OF VARIABLE NAMING

DEMONSTRATED FOR C# CODE AND CODE2VEC

by

Davey Mathijssen

in partial fulfillment of the requirements for the degree of

Master of Science
in Software Engineering

at the Open University, faculty of Science
Master Software Engineering
to be defended publicly on 2022-12-02 at 11:00 AM.

Course code:	IM9906	
Thesis committee:	dr. ir. H.P.E. Vranken (supervisor),	Open University
	dr. A.J. Hommersom (supervisor),	Open University

Open Universiteit
de beste! www.ou.nl

Contents

ACKNOWLEDGEMENTS

In pursuit of my master's degree, I have received a lot of support from my friends and family whom I wish to thank all. During my study, I got a lot of energy and support from my fellow student Wesley de Kraker. I have learned a lot from our stimulating discussions and I am grateful for the fun we had together. Furthermore, I would like to thank Harald and Arjen for their support and critical look. Finally, I would like Tamara for her support and patience during my entire study period.

SUMMARY

Machine learning techniques could be used to prevent malicious parties from abusing security vulnerabilities in source code. Additionally, as shown by other research, developers have trouble detecting vulnerabilities themselves. However, these techniques are still impractical and inaccurate. Therefore, to adopt these techniques in the field, they must first be improved. Various studies have questioned the importance of variable names in source code when used in machine learning models trained for various tasks. However, the effect of variable names has not yet been researched for models trained on the task of vulnerability detection. Therefore, we have examined the influence of variable names on the performance of machine learning models when trained on the task of vulnerability detection. Specifically, we have investigated what influence variable obfuscation has on the code2vec model when trained on the task of detecting software vulnerabilities. The code2vec model is a model which is able to represent source code as vectors, which can subsequently be used by the model, or any other model, to perform tasks, such as detecting vulnerabilities.

To answer our research question "What influence does variable obfuscation have on the code2vec model when trained on the task of detecting software vulnerabilities?" we first extracted vulnerability samples from publicly available vulnerability test case sources.

Next, to determine the effect of a difference in variable names we applied three different variable name obfuscation methods to these samples. By variable name obfuscation, we mean substituting the variable names with another string value. The first method (type obfuscation) substitutes the variable name with type information of the variable. The second method (random obfuscation) substitutes the variable names with a random string. The third method (semi obfuscation) substitutes only 50% of the variable names with type information.

Afterwards, we trained classifiers using the code2vec model, which translates source code into vectors using an attention mechanism. These classifiers were trained on non-obfuscated- and obfuscated vulnerability samples. We have calculated various metrics to compare the classifiers and thus the effect of variable obfuscation. These metrics showed us that variable obfuscation does not affect the performance of classifiers trained on the task of vulnerability detection. We subsequently analysed the attention mechanism of code2vec and found out that the code parts which received the most attention to make a prediction, were not different between non-obfuscated classifiers and obfuscated classifiers. Additionally, no difference was shown between the various obfuscation methods.

1

Introduction

1.1. Software security

Software security is a concept in which software should be designed and written in such a way that software will continue to function properly, even when attacked by malicious parties [1]. The importance of designing and writing secure software is becoming more and more evident with the increase in losses caused by malicious parties. According to the FBI, over 3.5 billion US dollars have been lost in 2019 and 4.2 billion in 2020 due to cybercriminals [2]. Cybercriminals utilise attack vectors to gain unauthorised access to computer systems. One of these attack vectors is exploiting software vulnerabilities in source code.

1.2. Software vulnerabilities

Software vulnerabilities are weaknesses or defects in a design or implementation of a system that enables malicious parties to exploit such a system for their own purposes. For example, a software vulnerability in an Energy Management System (EMS) could allow attackers to take control of energy flows and subsequently harm energy-dependent organisations, such as emergency services and critical industrial enterprises. Some examples of well-known critical vulnerabilities are HeartBlead [52] and the more recent Log4j security vulnerabilities [53]. Multiple organisations keep track of vulnerabilities that are discovered in software, such as the National Institute of Standards and Technology (NIST) and MITRE, which maintain the National Vulnerability Database (NVD) [54] and the Common Weakness Enumeration (CWE) database [55], respectively. The MITRE also maintains the Common Vulnerabilities and Exposures (CVE) system. These projects are described in Section 2.4.

1.3. Software vulnerability detection

With ever-increasing code production, the need for automated and reliable methods to find security vulnerabilities increases. In a study conducted by Edmundson et al., 30 individual developers were instructed to find the seven vulnerabilities in a given web application without using any automated tools [3]. The results showed that none of these developers could find more than five vulnerabilities, and even 20 per cent of the developers could not find any real vulnerabilities. A study performed by Meneely et al. has even concluded that, statistically, files are more prone to contain vulnerabilities when a higher amount of

code reviews have been performed on these files and more reviewers have reviewed these files [4]. This seems counterintuitive because multiple reviews could eventually find previously undetected vulnerabilities and multiple reviewers have different experiences, which could lead to one reviewer detecting vulnerabilities that other reviewers could not detect. Hence, automated solutions are being developed to help developers find security vulnerabilities in various stages of software development.

For this reason, sophisticated systems are needed that can guide and help developers and code auditors detect and fix software vulnerabilities in source code (with the lowest amount of false negatives and false positives). Security vulnerabilities can also be found in programs without accessing their source code, like the Heartbleed bug [56–58], but it is arguably better to detect and prevent vulnerabilities before they are being published in applications. This prevents the existence of vulnerable versions of applications in the first place and therefore decreases possible attack vectors for malicious parties. Nevertheless, it should be noted that developing sound and complete systems is not possible because of Rice's Theorem [5]. Consequently, it cannot be guaranteed that a vulnerability detection system does not output false negatives and false positives.

A multitude of approaches has been proposed to detect security vulnerabilities. Ghaffarian and Shahriari [6] have categorised these approaches as follows:

1. static analysis
2. dynamic analysis
3. hybrid analysis

These approaches have been used for many years, yet these approaches still have constraints, such as a high amount of false positives [7]. Additionally, Díaz and Bermejo tested nine static analysis tools to detect vulnerabilities found in SAMATE tests, but none could detect all vulnerabilities [8]. In the last two decades, the number of machine learning approaches to detect software vulnerabilities has increased, as shown in recent surveys [6; 9; 10]. Many techniques are available to translate data into a numerical form, which is then typically used by machine- and deep learning approaches as input data. In Natural Language Processing, tokenization techniques are widely used to translate text or speech into vectors, like the word2vec [11; 12] and seq2seq [13] techniques. Natural language processing and tokenization are described in more detail in Section 2.1. Such techniques used in natural language processing can also be used to translate source code into vectors.

In 2018, Alon et al. [14] demonstrated a method to represent source code using the paths of the abstract syntax tree belonging to the source code, in which they applied Conditional Random Fiels and word2vec to the abstract syntax tree paths. In follow-up research [15; 16], Alon et al. have increased their method's effectiveness when trained on the task of predicting method names. This is accomplished by using a neural network with an incorporated attention mechanism in addition to the abstract syntax tree paths. This attention mechanism is able to put more weight on extracted paths that are most relevant to expressing the semantic meaning of a piece of code. Code2vec is a neural network that is able to represent source code snippets as a fixed-length vector [15] (as opposed to words as vectors with word2vec). Code2seq [16] is a neural network that represents source code snippets as natural language sequences (as opposed to sequences as sequences with seq2seq). Both

models can export the vector/sequence representation of code. Therefore, the resulting vector/sequence representation can be used in all kinds of classifiers.

One of the main advantages of the code2vec and code2seq models is automated feature generation. Therefore, code2vec and code2seq can be used in an automated manner without manual and laborious feature engineering. For example, code2vec could be used in an automated continuous integration build pipeline [17] to detect security vulnerabilities or other coding defects. Nevertheless, adding custom features to the code2vec or code2seq model will be out of scope for this research. Code2seq uses a long short-term memory architecture that could provide better results but also requires more processing power, memory and time to train a model (compared to the simpler architecture used by code2vec). Coimbra et al. [18] show that it is possible to train a code2vec model with a corpus of almost 22000 functions in 5 minutes on an affordable GPU for consumers [18]. Because we do not have a high-end system available for this research, code2vec will be the model we use to represent source code instead of code2seq. However, code2seq uses an interface that looks very similar to the interface of code2vec. Therefore, our research could be adapted to work with code2seq without much effort.

Compton et al. have noticed that code2vec is hugely influenced by variable naming [19]. For example, a typo has an immense effect on the prediction, as discussed in Section 2.6. They have proposed various variable obfuscation methods in which variable names are substituted by another string value. However, only two of these obfuscation methods have been tested. Additionally, not all downstream tasks benefit from variable obfuscation [19]. Nevertheless, the task of detecting vulnerabilities has not been examined yet, and therefore, we are interested to see the impact of code2vec's ability to detect various software vulnerabilities when applying different variable name obfuscation methods to our data set. Depending on the number of unique user-defined variable names used in the data set, variable obfuscation could also reduce the vocabulary used to train the model, which could result in reduced memory usage and training time.

1.4. Research goal

This research will investigate the influence of variable name obfuscation on a code2vec model trained to detect vulnerabilities. Therefore, this research will contribute to understanding the influence of variable names within representations of vulnerable source code when used to train models on the task of vulnerability detection. Although there are many successful approaches to detect security vulnerabilities, Morrison et al. concluded that machine learning models trained on the task of vulnerability prediction are still underperforming, which results in such systems not being adopted in the field [20]. Therefore, we have to find solutions to improve the performance of these models for the task of detecting vulnerabilities in source code. Research done by Compton et al. suggests that using obfuscated variable names will more accurately depict the semantic meanings of the code [19] and, therefore, could improve the results of code2vec detecting vulnerabilities. However, the research of Compton et al. has used various aggregation methods to create embeddings representing an entire class, instead of a single method. Additionally, Compton et al. have not trained a model on the task of vulnerability detection. Therefore the impact of variable name obfuscation on detecting vulnerabilities is not investigated. Furthermore, Compton et al. have suggested an obfuscation method, which they have not used themselves. Therefore, we will use this suggested obfuscation method in addition to the two methods that

have already used by Compton et al. to see whether or not the third method results in better detection performance. We will focus on vulnerabilities in web applications because web applications are accessible by anyone and therefore an easy target. More specifically, we focus on web applications written in C#, which is a popular programming language. Additionally, we will train the code2vec model on multiple vulnerabilities and different data set sizes for each vulnerability. We will elaborate on these topics in Chapters 2 and 3.

1.5. Report structure

The following chapters of this report will describe our research in more detail. Chapter 2 provides more background details, including code2vec, performance metrics and related studies. Next, Chapter 3 describes the research question and methodology of this research. In Chapter 4 the used data sources are described, together with a description of the process to create the data set for our experiments. In Chapter 5 we show the results of our trained classifiers and their performance when trained on the task of vulnerability detection. Finally, we will discuss these results in Chapter 6 and conclude this report with the conclusion and recommendations for further research in Chapter 7.

2

BACKGROUND

2.1. NATURAL LANGUAGE PROCESSING

To understand how source code can be translated and interpreted by machine- and deep learning models, we first look at Natural language Processing (NLP). NLP is a field of research in which human language (in text or speech form) is processed using computer algorithms. The goal is to understand the content of text or speech, so that information can be extracted from the content considering its context. Some applications of NLP tasks are text-to-speech, grammatical error correction and translating text from one language to another. Text can be seen as a sequence of characters or a sequence of words. Machine- and deep learning models can interpret these sequences directly. However, computers perform better using mathematical representations of these sequences. Therefore, most models use vectors, which represent these sequences. when these sequences are first converted into vectors before being interpreted by machine- and deep learning models. Some commonly used conversion methods from sequences into vectors are [21]:

- Segmenting text into words and transforming each word into a vector.
- Segmenting text into characters and transforming each character into a vector.
- Extract n-grams of words or characters, and transform each n-gram into a vector.

Units such as words, characters, and n-grams of words or characters are called tokens. The process of converting these units into tokens is called tokenization (Figure 2.1). In the following sections, we will describe some relevant terms for our research related to tokenization.

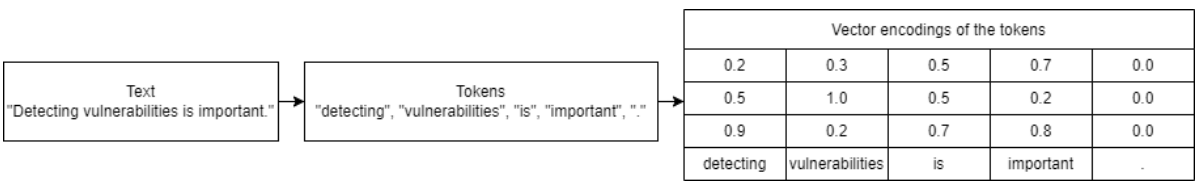


Figure 2.1: Tokenization of text and from tokens to vectors.

2.1.1. N-GRAMS AND BAG-OF-WORDS

N-grams are groups of N contiguous tokens. The 1-gram, 2-gram and 3-gram of the sentence "Detecting vulnerabilities is important" are shown in Figure 2.2, which uses words as tokens. N-grams have various applications, for example, predicting the following token in a sentence.

Another model to represent tokens is a bag-of-words. Bag-of-words are a set of words and therefore do not contain the structure of the sentence because the tokens are unordered. Bag-of-words can be used in tasks such as predicting the title of a document by using the most frequent words in the bag-of-words. Bag-of-words are commonly used for feature generation, such as the frequency of words or whether or not a word is present in a sentence. We will show one example where bag-of-words can be used to create vector inputs for a machine- or deep learning model. When we consider the following two sentences, "Detecting vulnerabilities is important" and "Vulnerabilities are bad", we can create the following vocabulary: "detecting", "vulnerabilities", "is", "important", "are", "bad". Subsequently, we can create a binary vector for each sentence showing which words are present, as shown in Figure 2.3. We use the arbitrary order of words in our vocabulary to represent each vector. Notice that the binary vectors grow when new words are added to the vocabulary, which requires more memory and processing power when used in machine- and deep-learning models. Therefore, it is beneficial to try and decrease the size of the vocabulary. There are numerous ways to accomplish this, for example, by removing punctuation and stop words (e.g. a, an, or). Additionally, vocabularies can be built by using n-grams.

```
n=1: detecting, vulnerabilities, is, important
n=2: detecting vulnerabilities, vulnerabilities is, is important
n=3: detecting vulnerabilities is, vulnerabilities is important
```

Figure 2.2: 1-gram, 2-gram and 3-gram of the sentence "Detecting vulnerabilities is important".

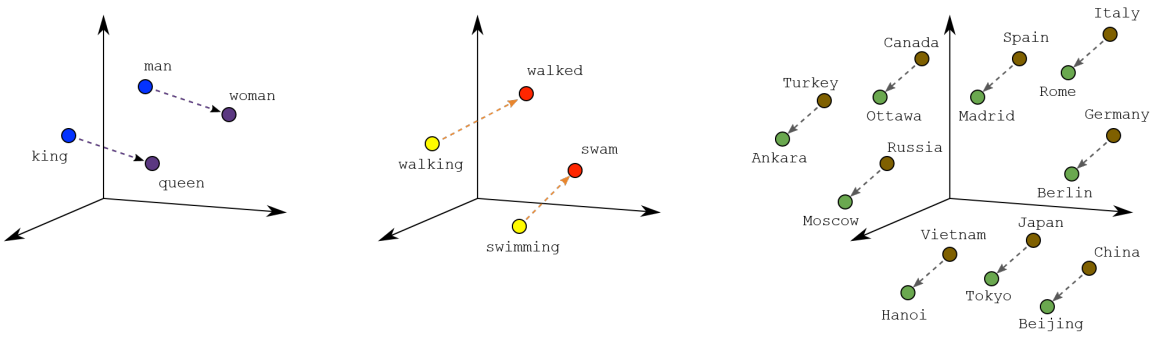
```
Detecting vulnerabilities is important: [1, 1, 1, 1, 0, 0]
Vulnerabilities are bad: [0, 1, 0, 0, 1, 1]
```

Figure 2.3: Bag-of-words model used to create binary vectors.

2.1.2. WORD EMBEDDINGS

A commonly used approach in natural language processing to understand natural language data, such as text and spoken words, is to use machine- and deep learning techniques combined with word embeddings. Unlike bag-of-words, word embeddings capture semantic properties of words and present these properties as vectors in a word embedding vector space. The relations between tokens are captured in this vector space, such as meaning, context, and morphology. When mapped in a vector space, the distance between words and the location indicates the semantic similarity between these words, as depicted in Figure 2.4. Similar words would be represented as similar word embeddings.

The idea of mapping semantically related words to real-valued vectors, which are also closely related, was first introduced by Bengio et al. [22] in 2003. However, the technique



Male-Female Verb Tense Country-Capital

Figure 2.4: Different relations between tokens represented in embedding vector spaces. The distance between words and the location indicates the semantic similarity between words. For example, king is more closely related to queen than queen to man. However, king and man are also related because they both are masculine words. The same applies to woman and queen, both being female. Adapted from original image by [59].

had a breakthrough in 2013 when Mikolov et al. introduced the word2vec [11; 12] word-embedding scheme. In 2014 another popular word-embedding scheme was introduced by Stanford researchers: Global Vectors for Word Representation (GloVe) [23].

Besides natural language processing, word embeddings are also used to solve various other problems in the computer science domain, such as program property prediction [14] and vulnerability detection [24; 25].

2.1.3. SEQUENCE TO SEQUENCE

Besides word embeddings, sequence to sequence techniques are widely used in combination with machine- and deep learning in natural language processing. Sequence to sequence translates token sequences into other token sequences, which can be used for various tasks such as language translation, text summarization, image captioning, and conversational models. One such technique is seq2seq [13]. Seq2seq uses an encoder component to encode the source language to a hidden vector representing the token and its context. Subsequently, a decoder component can translate this hidden vector to a target language. The essence of seq2seq is shown in Figure 2.5.

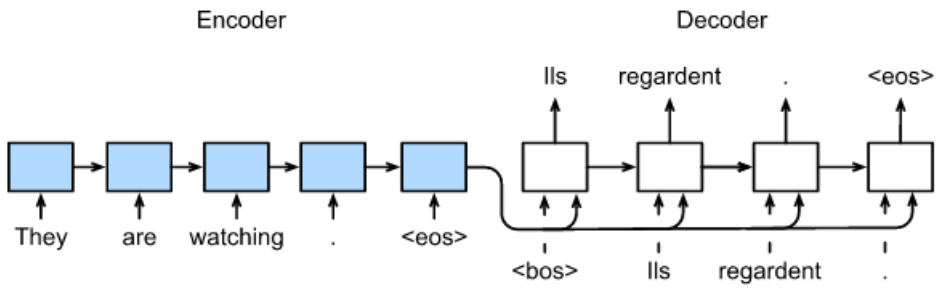


Figure 2.5: Seq2seq learning with an encoder and decoder. Adapted from original image by [26].

2.2. CLASSIFIERS AND MODEL PERFORMANCE

Machine learning models are algorithms which can 'learn' from data to perform certain tasks, such as predicting a method name for a code snippet (as seen in the code2vec study [15], which is described in Section 2.3). Models which are capable of categorising data into classes are called classifiers. This is done by labelling the data, which the model learns from. The process of learning associations between input data and the corresponding labels is called supervised learning. When a machine learning algorithm is able to classify samples without using (manually) labelled samples, this is called unsupervised learning. Two types of classifiers can be defined:

1. **Binary classifiers** are models which are able to classify samples into two classes.
2. **Multi-class classifiers** are models which are able to classify samples into any number of classes.

This study will focus on supervised learning and specifically on binary classifiers, which will predict whether a code snippet is vulnerable or non-vulnerable.

The performance of the code2vec model (or any other classifier) can be evaluated by using various metrics. Typically, a model can give a correct or incorrect prediction, for example, whether or not a code snippet contains a vulnerability or not. When a classifier predicts a code snippet is not vulnerable, we consider this prediction as negative. In contrast, we consider a 'vulnerable' prediction as a 'positive' prediction. A positive or negative prediction can subsequently be checked to be true (the code snippet is predicted correctly) or false (the code snippet is predicted incorrectly). Therefore, in case of vulnerability detection, the following possibilities can occur:

- **True Positive (TP):** both the predicted and actual class are vulnerable.
- **False Positive (FP):** the predicted class is vulnerable, but the actual class is non-vulnerable.
- **True Negative (TN):** both the predicted and actual class are non-vulnerable.
- **False Negative (FN):** the predicted class is non-vulnerable, but the actual class is vulnerable.

Table 2.1: An example confusion matrix for a total of 1000 cases.

	Vulnerable (actual)	Non-vulnerable (actual)
Vulnerable (predicted)	500 TPs	15 FPs
Non-vulnerable (predicted)	25 FNs	460 TNs

We can show the number of TP, FP, TN, and FN cases in a confusion matrix. An example confusion matrix is shown in Table 2.1. The number of TPs, FPs, TNs, and FNs can be subsequently used to calculate a number of metrics, which can be used to give indications about the performance of a classifier. We will now describe some of these metrics.

The **accuracy** metric describes the ratio of correctly classified samples divided by the total number of samples. However, the accuracy metric alone can be a misleading metric to describe the performance of a classifier. This is the case for an imbalanced data set. A data set is considered imbalanced when one of two (or more) classes has a small presence in that data set, as opposed to the other class(ses) in that data set. For example, when a data set contains 99% non-vulnerable code and 1% vulnerable code, a classifier could easily achieve an accuracy of 99% by simply classifying each input as a non-vulnerable piece of code. Accuracy is calculated by using the following formula:

$$Accuracy = (TP + TN)/(TP + FP + TN + FN)$$

In addition to accuracy, **Cohen's Kappa** (κ) can be used to describe the agreement between reality and a classifier's predictions. Although κ is related to the accuracy metric, it considers a chance factor making it more robust than accuracy [27]. However, in recent studies, the use of κ to evaluate the performance of classifiers is questioned [28]. Despite this, κ is still used in studies [19]. κ can be calculated by using the following formula [28]:

$$\kappa = \frac{2 * (TP * TN - FP * FN)}{(TP + FP) * (FP + TN) + (TP + FN) * (FN + TN)}$$

Where the minimum value is -1 and the maximum value is 1, indicating a perfectly wrong prediction and a perfect prediction respectively. A value of around 0 means the prediction was similar to random guessing.

In addition to accuracy and κ, the **precision** and **recall** metrics are used to describe the performance of classifiers. Precision describes the ratio of actual positives (in our case vulnerable code snippets) to all samples which were predicted to be positive. Recall describes the ratio of correctly identified positives out of the total positive samples, in our case recall describes the proportion of how many of the actual vulnerabilities have been found.

$$Precision = TP/(TP + FP)$$

$$Recall = TP/(TP + FN)$$

The **F_1 score** is used to combine the recall and precision metrics. A higher precision often has a negative impact on the recall and vice versa. To better understand the F_1 score, we try to answer the following question in context to our vulnerability prediction task: what false prediction is worse, an FP or an FN? An FN means a vulnerability has not been detected and therefore a developer will not be alerted by the classifier to fix this vulnerability. A high recall reduces the chance of this happening. In contrast, an FP will alert a developer to fix a vulnerability when there is none, thus wasting precious time in the process. The F_1 score can therefore be used to find a middle way for both the recall and precision of a model.

$$F_1 = 2 * ((precision * recall)/(precision + recall))$$

2.3. CODE2VEC

Code2vec was proposed in 2019 by Alon et al. and uses a neural network to represent code snippets as fixed-length code vectors, which can predict the semantic properties of the code snippet [15]. Thus, the vectors provided by code2vec capture the meaning (semantics) of the code snippet. These vectors can be used for various tasks, such as predicting descriptive method names for code snippets and detecting security vulnerabilities in code snippets.

Code2vec is able to translate code snippets of various programming languages to continuous distributed vectors (also referred to as 'code embeddings'). To achieve this, an Abstract Syntax Tree (AST) of a code snippet is given as input to the code2vec model, together with a corresponding label, which describes the semantic property of the code snippet. This way, the model can be trained to predict properties for code snippets. Code snippets which are semantically closely related are converted into vectors that are also close to each other.

The code2vec model converts the AST of a code snippet (more specifically a single method/function) into an unordered bag (multiset) of the AST's extracted paths, referred to as path-contexts by Alon et al. Subsequently, the model must learn to link corresponding bags and labels, with the additional requirement to map bags with the same label to close vectors. To accomplish this, a neural attention based feedforward network architecture is used (as depicted in Figure 2.6). This architecture learns how much focus (attention) needs to be given to each element in the bag of paths. This attention mechanism calculates a weighted average of the path-context vectors, leading to a single code vector. This code vector can be used for various tasks, such as predicting security vulnerabilities using the softmax function layer, which is already implemented in the code2vec network. However, the vector can also be extracted and used as input for a custom model.

For better interpretability, it is possible to visualize allocated weights. The code embeddings can be exported from a trained code2vec model, subdivided into token embeddings (the features) and the vectors containing the labels. These are represented in a word2vec format. Therefore, these files can be inspected using tooling, such as the open-source Gensim library [60].

According to Alon et al., a syntactical representation (in the case of code2vec, this is the AST) is more valuable for representing source code in machine learning models than n-grams and manually designed features, which is motivated by earlier work [14]. In contrast, Allamanis et al. [29] have used semantic relations to represent code snippets, which could reveal additional information that is not explicitly available in syntactic-only representations. Nonetheless, the use of semantic knowledge has a few disadvantages as well:

1. Only experts are able to choose and design semantic analyses.
2. It is hard to generalize different programming languages because semantic analyses have to be implemented for every programming language accordingly.
3. Designed semantic analyses are possibly hard to generalize for different tasks.

When compared to the syntactic approach used by code2vec:

1. No programming language expertise or manual feature design is needed;

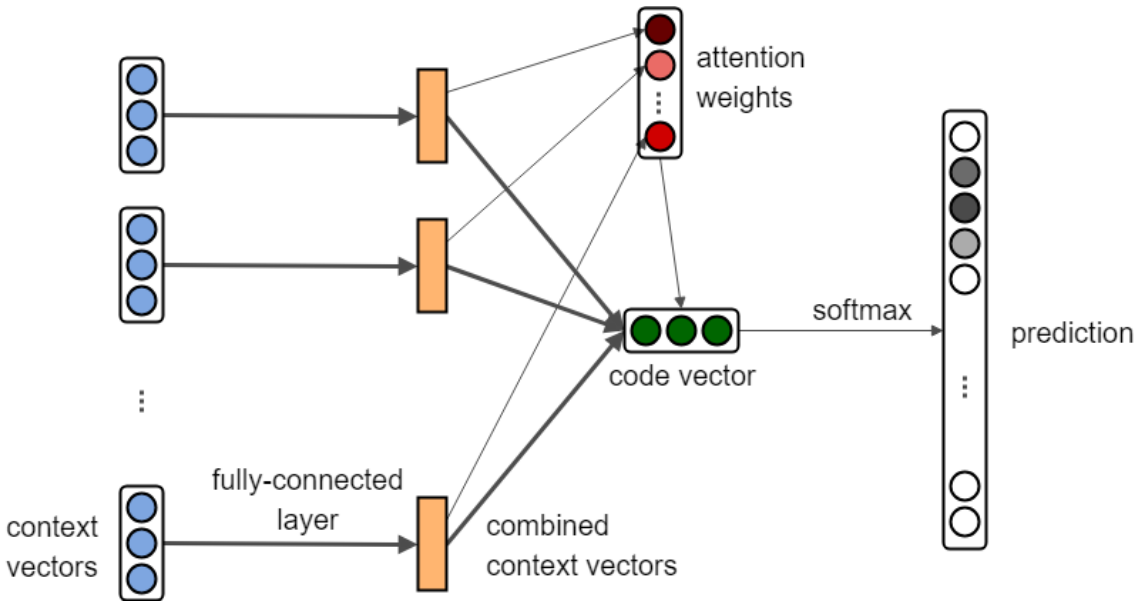


Figure 2.6: The code2vec model architecture. Adapted from original image by [15].

2. To generalize different programming languages, the language parser needs to be exchanged for the specific language, with the requirement that the same traversal algorithm is used to provide an AST;

3. As shown by Alon et al. [14], syntactic paths perform well when used for different tasks.

Alon et al. have demonstrated this property by predicting a meaningful method name for the vector created by code2vec of a given method body code snippet.

Alon et al. have distributed their source code of code2vec on GitHub [61], together with AST extractors for C# and Java code snippets. However, third parties have already published extractors for other programming languages, such as Python, C and C++ [62] and TypeScript [63]. Additionally, it is possible to create extractors for other programming languages compatible with code2vec.

2.4. Software vulnerability projects

2.4.1. Common Vulnerabilities and Exposures

The Common Vulnerabilities and Exposures (CVE) project is a list that contains publicly disclosed computer security vulnerabilities [64]. These software vulnerabilities are assigned a unique CVE ID provided by a CVE Numbering Authority (CNA). Current CNAs are parties such as software vendors, open-source projects and coordination centres, such as Microsoft Corporation, Apache Software Foundation and CERT/CC. When someone discovers a new vulnerability, the person can report this vulnerability to a CNA whose scope concerns the product in which the vulnerability was found and request a CVE ID for this vulnerability. A CVE ID is then reserved, after which the CNA can investigate the reported vulnerability.

The CVE ID contains the CVE prefix, the year in which the CVE ID was reserved or published and a sequence number portion, which can include four or more digits, resulting in the following format: CVE prefix + year + sequence number.

When the CNA has validated the vulnerability and has provided at least a brief description, name and version(s) of the affected product, a relevant reference, and a vulnerability type, root cause or impact, the CVE record will be published. However, if the reported vulnerability is invalid, the CVE record will be rejected.

2.4.2. Common Weakness Enumeration

Various taxonomies of vulnerabilities have been developed to classify vulnerabilities that have similar characteristics into categories. One of these taxonomies is the Common Weakness Enumeration (CWE), a database containing a list of software and hardware weakness types maintained by MITRE [55].

Each CWE entry is represented by information such as a CWE ID (a unique identification number containing a CWE prefix and a sequence number), a description, information about when and how the vulnerability may be introduced, and the likelihood of exploitation of the weakness.

The list helps developers learn about common vulnerabilities and aims to help developers prevent introducing these vulnerabilities in their systems. It also helps reduce risks by enabling an effective way of communicating about vulnerabilities. Likewise, security practitioners and researchers use the CWE to help them in their work, for example, by providing a standard to evaluate tools that try to find and identify weaknesses. The list contains over 900 vulnerability classes, including stack-based buffer overflows (CWE-121), Cross-Site Request Forgery (CSRF) (CWE-352), and Out-of-bounds Write (CWE-787).

2.4.3. National Vulnerability Database

The National Vulnerability Database (NVD) is a database that contains vulnerabilities found in public software, filled with entries from the CVE list [65]. The National Institute of Standards and Technology (NIST) enriches the entries with technical details, such as the corresponding CWE and a vulnerability scoring system (CVSS) [30] that indicates the vulnerability's severity. This information is valuable to various types of security professionals to learn about vulnerabilities and help them take action to prevent malicious actors exploit these vulnerabilities. The NVD can also help create insights, such as the distribution of vulnerabilities by severity over time [66], as seen in Figure 2.7.

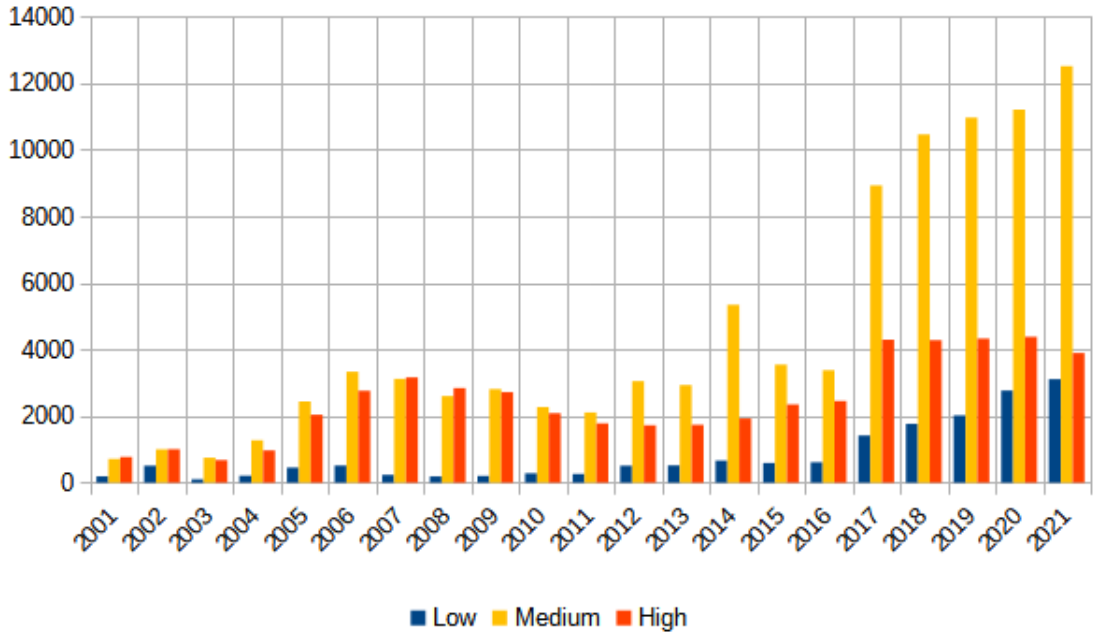


Figure 2.7: The distribution of vulnerabilities by severity over time, as reported at the NVD up to and including 2021.

2.4.4. OWASP

The Open Web Application Security Project (OWASP) is an organization that is committed to improving the security of software, specifically for web applications. They do this through various projects, which are all freely available. Examples of these projects are OWASP ZAP [67] (a penetration testing tool), WebGoat [68] (a web application with deliberately implemented vulnerabilities for education purposes), and the OWASP Top 10 [69]. The OWASP Top 10 is a document describing the top 10 most critical security risks to web applications of that moment. The document aims to bring awareness of these security risks to developers and executives. The first edition of the OWASP Top 10 was released in 2003, and updated versions were released in 2004, 2007, 2010, 2013, 2017, and 2021. Since 2010, the focus for compiling the ranking has been based on the occurrence of a vulnerability, risks, exploitability and technical impact. The ranking contains ten categories, which comprise various CWEs per category.

2.5. VULNERABILITIES

The vulnerabilities we will use in our experiments were selected from the top 10 most critical risks defined by the OWASP Top 10 [69]. Additionally, we depend on the available vulnerability data set samples described in section 4.5. The three most critical risk categories of the OWASP Top 10 (broken access control, cryptographic failures and injection) are all present in the available data sets we will use. However, not all CWEs of each OWASP category are available in these data sets, and therefore only the available CWEs are used in our experiments. Nonetheless, vulnerabilities within the cryptographic failures category can already be detected with high precision using static code analysis tools, as demonstrated by the CryptoGuard research [31]. Therefore, the added value of detecting these vulnerabilities with machine- and deep learning is not as interesting as the other two categories and will therefore not be used in our experiments. Additionally, previously deemed robust cryptographic algorithms could later become obsolete after being broken by cryptographic experts (which is also described in CWE-327: Use of a Broken or Risky Cryptographic Algorithm [70]). Consequently, we would need to modify our data set and retrain our models when an algorithm becomes obsolete. The following sections briefly explain the vulnerability categories (broken access control and injection) and corresponding CWEs used in our experiments. A complete overview of the (number of) samples per OWASP category and CWE used in this research can be found in Appendix A, specifically in Table A.1 and Table A.2.

2.5.1. BROKEN ACCESS CONTROL VULNERABILITIES

Access control enforces policies on users such that they cannot perform actions for which they do not have permission. However, malicious users can bypass access control policies and perform unauthorised actions by using Broken Access Control vulnerabilities. These actions can result in unauthorised information disclosure, modification, or deletion. Some commonly exploited CWEs in this category are CWE-22 (Path traversal), CWE-200 (Exposure of Sensitive Information to an Unauthorised Actor), CWE-201 (Insertion of Sensitive Information Into Sent Data), and CWE-352 (Cross-Site Request Forgery). The MITRE contains various broken access control vulnerability CWEs in the 2021 edition of their CWE Top 25 Most Dangerous Software Weaknesses list [71], such as CWE-22 ranked 8, CWE-352 ranked 9 and CWE-200 ranked 20. CWE-22 is briefly explained in the following section.

PATH TRAVERSAL

Path traversal vulnerabilities allow unauthorized users to access files which should be inaccessible to these users. This could provide malicious users with confidential information, but also could allow malicious users to create, modify or delete files. When referencing files with a prefix containing one or multiple "../" cases, it is possible to access files outside the current directory.

The example in Listing 2.1 shows a method which checks whether or not a file exists in the current directory. However, because line 7 is not true, the "../" cases are not removed and a user could get unauthorized access to data outside the current directory. In this case, it shows the user whether a requested file exists.

Listing 2.1: Path traversal sample from the SARD 105 Test Suite.

```
1  public static bool Bad(string[] args)
2  {
3      string tainted_2 = null;
4      string tainted_3 = null;
5      tainted_2 = args[1];
6      tainted_3 = tainted_2;
7      if ((1 == 0))
8      {
9          string pattern = "^[\\.\\.\\/]+";
10         Regex r = new Regex(pattern);
11         tainted_3 = r.Replace(tainted_2, "");
12     }
13
14     //flaw
15     return File.Exists(tainted_3);
16 }
```

2.5.2. INJECTION VULNERABILITIES

Injection vulnerabilities are ranked third in the 2021 edition of the OWASP Top Ten [69] and were even ranked first in previous editions (from 2010 until 2017) [72–74]. The MITRE contains numerous injection vulnerability CWEs in the 2021 edition of their CWE Top 25 Most Dangerous Software Weaknesses list [71], such as CWE-78 (OS Command Injection) ranked 5, CWE-89 (SQL Injection) ranked 6, and CWE-77 (Command Injection) ranked 25. Injection vulnerabilities are possible when programs accept untrusted data, such that an interpreter processes this data as part of a command or query. When this data is not sanitised, it is possible to insert data that trick the interpreter into accessing data without valid authorisation and executing unwitting commands. We will briefly explain a common injection vulnerability in the following section.

SQL INJECTION

SQL (Structured Query Language) injection vulnerabilities are known under CWE id 89. An example of a SQL injection can be seen in Listing 2.2. The example shows the vulnerability in line 12. The "data" variable is untrusted and unsanitised. Because of this, an attacker could inject a malicious SQL command via the "data" variable, allowing the attacker to execute these SQL commands on the database. This could result in unwanted effects such as information disposal, data modification or data removal. However, SQL injections like these can be prevented by using parameterised queries and not using string concatenation in SQL commands. Following these examples could have prevented the vulnerability in our example of Listing 2.2). Additionally, it is best to use the least required amount of privileges for database access by the client.

Listing 2.2: SQL injection sample from the Juliet C# Test Suite.

```
1  public override void Action(string data , HttpRequest req,
       HttpResponse resp)
2  {
3      int? result = null;
4      try
5      {
6          using (SqlConnection dbConnection = IO.GetDBConnection())
7          {
8              dbConnection.Open();
9              using (SqlCommand badSqlCommand = new SqlCommand(null,
                   dbConnection))
10             {
11                 /* POTENTIAL FLAW: data concatenated into SQL
                       statement used in ExecuteNonQuery(), which could
                       result in SQL Injection */
12                 badSqlCommand.CommandText = "insert into users (
                       status) values ('updated') where name='" +data+"'
                       ";
13                 result = badSqlCommand.ExecuteNonQuery();
14                 if (result != null)
15                 {
16                     IO.WriteLine("Name, " + data +", updated
                           successfully");
17                 }
18                 else
19                 {
20                     IO.WriteLine("Unable to update records for user:
                           " + data);
21                 }
22             }
23         }
24     }
25     catch (SqlException exceptSql)
26     {
27         IO.Logger.Log(NLog.LogLevel.Warn, "Error getting database
               connection", exceptSql);
28     }
29 }
```

2.6. Related studies

Code2vec can help with numerous programming language processing tasks, such as finding duplicate code and classifying code snippets. Additionally, recent studies have already explored using code2vec to detect security vulnerabilities. Coimbra et al. have compared the ability of code2vec to detect security vulnerabilities with various transformer-based models used in natural language processing and computer vision [18]. In their approach, code2vec reached an accuracy of 61.43%, detecting vulnerabilities from the Devign data set [32] without distinguishing the type of vulnerability by using the default code2vec hyperparameters. Results show that code2vec bested more simple models but was exceeded by more expressive models that used more parameters. However, code2vec produced results using relatively low system resources and training time. These advantages could be decisive when system resources and time are sparse.

Baptista et al. have opted to use code2seq instead of code2vec for vulnerability detection and classification [33]. By tuning the hyperparameters of code2seq, they claim to have acquired a model with an accuracy of 85%, precision of 90%, recall of 97% and an F_1 score of 93%, acquired in just one epoch (which is one training cycle of a machine learning model). The hyperparameters have been optimised using the tool Sweeps [75], which can find optimised hyperparameters for a model in an automated fashion, using various search strategies, including random-, Bayesian-, and grid search. It is claimed that humans validated the used data set and it is therefore free of false positives. However, the used data set is not stated in their paper, thus their claims cannot be validated. Additionally, it is unknown which vulnerability types have been detected, except for a short introduction to the injection vulnerability category in their paper, which makes up 33 different CWEs in the latest version of the OWASP Top 10 [76]. Subsequently, the research of Baptista et al. is based upon a master thesis, which is not publicly accessible, and as a result, no additional information or validation is available. In contrast to the code2vec approach used by Coimbra et al., the approach of Baptista et al. requires more resources than a regular personal computer could provide.

Kang et al. [34] have researched the generalisability of the code2vec token embeddings for other downstream tasks. Code2vec performs well in predicting method names, as shown in Alon et al. [15]. However, Kang et al. claim that the code2vec token embeddings do not generalise well. Kang et al. have defined three downstream tasks used in their experiments to substantiate their thesis: generating code comments, identifying code authorship, and detecting code clones. Each experiment used a different model previously successfully used by other researchers to perform one of the defined downstream tasks. These models were augmented in their experiments with the vectors generated by code2vec. Subsequently, the experiments were performed again using other vectors and features, such as GloVe vectors, for comparison and validation. However, as Kang et al. already mentioned, the used models might not be suitable for token augmentation, and also their implementations could give distorted results. Additionally, other downstream tasks, such as vulnerability detection, are not evaluated in their research.

A complementary study by Compton et al. [19] states that code2vec focuses for a large part on variable naming and therefore focus on the semantics is lost, especially when trained on the task of predicting method names. This statement is also confirmed by the findings of Kang et al. [34], as removing token embeddings representing variable names resulted frequently in better representations for their investigated downstream tasks. Additionally, the

influence of variables is also questioned by Elema et al. [35]. They have used features extracted from various graph representations of source code to detect vulnerabilities in PHP code. They have removed variables from their feature set which they claim to have 'little expressiveness', but no clear details have been provided.

Compton et al. have trained the code2vec model by using data sets that use popular GitHub projects as their source. They generally use predictive variable naming[15] and are favourable for the task used in their research of predicting method names. Compton et al. have shown the focus on variables by examining some examples depicted in Figure 2.8. A typo or a completely different variable name results in an incorrect prediction. Compton et al. have also noticed that code2vec can only create embeddings for individual methods instead of being able to produce a vector representing a class. They have performed various method selection and aggregation methods on the method embeddings produced by code2vec to produce single vector representations for classes by using simple mathematical operations on sets of vectors. Subsequently, they have only trained a classifier using these class vectors and not the separate method vectors which code2vec creates by default. Additionally, they have used a custom classifier [19] to investigate the results of their approach, ignoring the code2vec classifier. This approach of creating class embeddings could be beneficial for downstream tasks such as predicting class names. However, we do not believe this would increase the ability of code2vec to detect security vulnerabilities because the security vulnerabilities we would like to predict are much more fine-grained. During their method selection and performing aggregation operations, valuable information could be lost when using the resulting embedding to train a model on the task of detecting vulnerabilities.

By obfuscating variable names, Compton et al. could better preserve code semantics. However, the performance of predicting method names was reduced due to this approach. Nevertheless, by obfuscating the variable names, code2vec might be more generically usable for other downstream tasks that rely more on semantics instead of variable naming. Additionally, this has some extra benefits: firstly, obfuscating variable names prevents code2vec from being vulnerable to adversarial attacks via variable names [19]. Secondly, obfuscating variable names prevents code2vec from focusing too much on variable names, and therefore typos in variable names do not impact the outcome. Likewise, it would be interesting to validate the impact of user-defined function- and method names on the ability of a model to detect vulnerabilities. However, the available security vulnerability data sets we use for our research (which we will describe in Section 4.5) scarcely use user-defined method names. Additionally, these user-defined methods are primarily used to run multiple vulnerability tests at once and therefore are unrelated to the security vulnerabilities themselves.

Compton et al. have proposed three obfuscation methods:

- **Type-obfuscation**: variable names are substituted with a string indicating the type and scope of the variable. Both the type and scope could be helpful for prediction.
- **Random-obfuscation**: variable names are substituted with a random string. Therefore, the model is not able to learn any trends from variable names and focuses more on the structure of the code.

- **Semi-obfuscation**: 50% of the variables are obfuscated, and the other 50% still use their original naming. This approach may provide the model with informative variable names but potentially does not lose the semantic meaning of the code snippet as much as a non-obfuscated snippet.

Of the three obfuscation methods, only the semi-obfuscation method has not been used in their experiments, and therefore no results are available for this approach. Examples of the type-obfuscation and random-obfuscation methods are shown in Listing 2.3. The obfuscation methods were tested on a variety of downstream tasks: 1. Method name prediction. 2. Differentiate between two types of Java files (those used for image processing versus serving web pages). 3. Algorithm classification. 4. Identifying the most likely author of the code snippet. 5. Bug detection. 6. Duplicate detection. 7. Malware classification.

The results of the obfuscation methods were different for each downstream task. The bug detection task is arguably the closest to our vulnerability detection task, and therefore its results are interesting to us. Nevertheless, these results are based on using class embeddings and not using the code2vec classifier. We believe using embeddings representing single methods would provide better results for detecting vulnerabilities because the single-class embeddings were created by omitting (possibly important) details, as described previously in this section. Overall, code2vec's performance in detecting bugs was relatively low. However, the random method provided a statistically significant improvement compared to using no obfuscation [19].

Listing 2.3: Variable obfuscation methods shown for a C# method.

```
// Original method
public string IncreaseAndConcatenateCountToInput(string input) {
  int count = this.objCount;
  this.objCount++;
  return input + count;
}

// Type-obfuscated method
public string IncreaseAndConcatenateCountToInput(string
    param_string_1) {
  int local_int_1 = this.field_int_1;
  this.field_int_1++;
  return param_string_1 + local_int_1;
}

// Random-obfuscated method
public string IncreaseAndConcatenateCountToInput(string FDGHSXJF) {
  int UJGRJWMU = this.MVKHEQTR;
  this.MVKHEQTR++;
  return FDGHSXJF + UJGRJWMU;
}
```

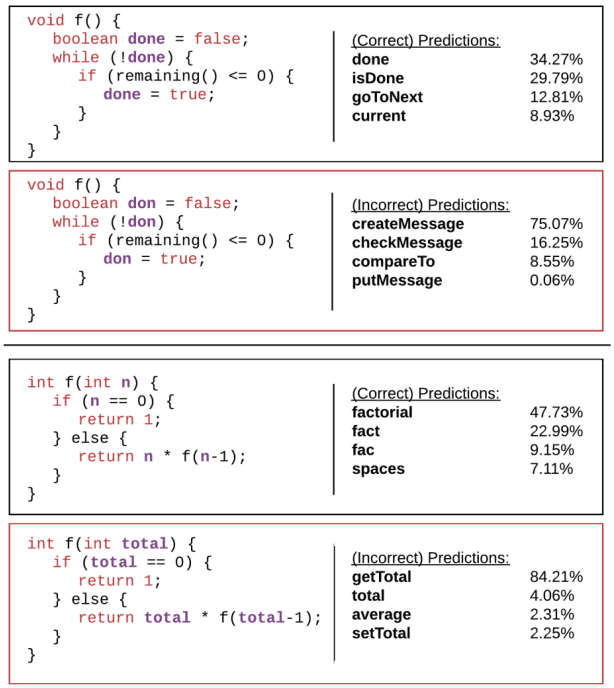


Figure 2.8: The method name predictions (shown in the right part of the figure) performed by the code2vec model for various code snippets (shown in the left part of the figure). (minor) changes in variable names result in substantially different predictions for a classifier trained on the task of method name prediction. The code is displayed in the left column, the predictions for this code in the right column. For example, the upper two rows show that when the 'e' is omitted in 'done', the model cannot produce a correct prediction. Additionally, the lower two rows show that renaming the variable 'n' to 'total' in a factorial function results in an incorrect prediction. This shows that the model is relying heavily on variable names to make predictions. Adapted from original image by [19].

3

RESEARCH DESIGN

3.1. RESEARCH QUESTIONS

The goal of this research is to answer the following research question:

What influence does variable obfuscation have on the code2vec model when trained on the task of detecting software vulnerabilities?

We will evaluate the influence of variable obfuscation on the code2vec [15] model when trained to detect software vulnerabilities. Morrison et al. concluded that vulnerability prediction models still produce bad results, which results in such systems not being adopted in the field [20]. Accordingly, we have to find solutions to improve the performance of these models for the task of detecting vulnerabilities in source code. Compton et al. state that when variable names are obfuscated, the model might focus more on the semantic properties of a code snippet [19]. Consequently, the models' performance to detect vulnerabilities could be increased because the semantics are better represented in the code vector which represents the code snippet. Therefore, we will investigate the influence of different variable name obfuscation methods on a code2vec model trained to detect vulnerabilities: type-obfuscation, random-obfuscation and semi-obfuscation. These three variable obfuscation methods are described in 2.6. We have several sub-questions that we will answer first to get an answer to our main research question:

RQ1: HOW DOES CODE2VEC PERFORM ON A NON-OBFUSCATED DATA SET WHEN TRAINED ON DETECTING SECURITY VULNERABILITIES?

Coimbra et al. have already shown that code2vec is a simple model that can be trained to detect security vulnerabilities in a relatively short time [18]. However, we need a baseline for our experiments to investigate our hypotheses on the influence of variable obfuscation on the vulnerability detection performance of code2vec.

RQ2: WHAT IS THE EFFECT OF USING OBFUSCATED VARIABLE NAMES ON THE CODE2VEC MODEL WHEN TRAINED TO DETECT BROKEN ACCESS CONTROL OR INJECTION VULNERABILITIES?

Compton et al. suggest that using obfuscated variable names will more accurately depict the semantic meanings of code [19]. Therefore, we will investigate whether or not vari-

able name obfuscation improves code2vec's vulnerability detection performance and compare it with our baseline results. Additionally, we will analyse the effect on the performance between the three proposed obfuscation methods.

RQ3: How does variable obfuscation affect a classifier when trained to detect multiple CWE vulnerabilities instead of one CWE vulnerability?

The results of a classifier which is trained on samples containing only one CWE category, do not directly imply these results will be the same for a classifier which is trained on samples containing multiple CWE categories. Therefore, we will examine whether or not variable obfuscation provides different results between classifiers which have been trained on only one CWE category and classifiers which have been trained on multiple CWE categories.

3.2. Research method

The hypothesis that variable names influence code2vec's performance in classifying vulnerabilities has resulted in the research questions described in the previous section. We have conducted multiple empirical experiments to validate this hypothesis and answer our main research question. Below, we first describe the data sets and tooling we have used for our experiments, along with the vulnerabilities on which our models have been trained to detect. Subsequently, we will describe the method for each sub-question.

We have created a base data set consisting of C# vulnerability test cases, which we describe in Chapter 4. In Section 2.5 we have already elaborated on the two OWASP Top 10 vulnerability categories we have focused on in our experiments (Broken Access Control and Injection vulnerabilities). These two categories are two of the top three OWASP Top 10 vulnerability categories [69]. Therefore, we have selected cases from C# vulnerability test case sources, filtered the related vulnerabilities, and stored them in our data set. Labelling the samples is done by giving the methods specific names. Therefore, we have named methods containing a vulnerability on which we have trained the model with 'bad'. Methods that do not contain a vulnerability are named 'good'. Next, we have created custom tooling to automate the creation of our data sets. This tooling is to apply the various variable obfuscation methods to the data set samples and filter unwanted methods within the data sets which are not related to vulnerabilities. The tooling provided different variants of our data set, whereby a different kind of obfuscation method is applied to each variant.

Next, the samples were converted into path-contexts which are used as input for the code2vec model, by using the preprocessor tooling published by the code2vec study team [61]. After the preprocessing part had finished, the code2vec classifiers were trained on the task of detecting vulnerabilities. The performance of the classifiers has been evaluated using metrics which are previously described in Section 2.2. This allowed us to review the performance of each classifier and make comparisons. Additionally, we have utilised code2vec's ability to evaluate the attention given to the extracted path-contexts. This allowed us to (partially) interpret the logic behind the specific classifiers and allowed us to examine the differences between non-obfuscated and obfuscated classifiers in more detail.

RQ1: How does code2vec perform on a non-obfuscated data set when trained on detecting security vulnerabilities?

We have combined and extracted cases from various C# vulnerability test case sources to create our data sets. We created a non-obfuscated data set for each CWE category for which we are able to extract sufficient samples. After the non-obfuscated data set had been created for each of the CWE categories that we have examined, the corresponding code2vec classifiers have been trained on the task of detecting vulnerabilities using the non-obfuscated data set. Subsequently, metrics indicating the performance of the classifiers have been calculated (accuracy, precision, recall, and F_1), such that we were able to compare the performance of the classifiers trained on the obfuscated data sets (as we will describe in RQ2 and RQ3). All classifiers that we have trained are binary classifiers, which are able to predict whether or not a code snippet contains a vulnerability.

RQ2: What is the effect of using obfuscated variable names on the code2vec model when trained to detect broken access control and injection vulnerabilities?

Besides the classifiers trained on non-obfuscated vulnerability samples, we have trained classifiers using obfuscated vulnerability samples, based on the same non-obfuscated data set. Therefore, we have created three additional variants (copies) of the non-obfuscated data set, which are used to answer RQ2. The variable names in each data set variant are obfuscated using one of our proposed variable name obfuscation methods, as described in Section 2.6. We have trained additional classifiers, whereby each classifier is trained using one of the available obfuscated data set variants. Finally, we have reviewed the results for each trained classifier, including the classifiers which were trained on the non-obfuscated data sets. Finally, performance metrics were calculated to enable us to compare the classifiers.

RQ3: How does variable obfuscation affect a classifier when trained to detect multiple CWE vulnerabilities instead of one CWE vulnerability?

To be able to answer RQ1 and RQ2 we have trained classifiers whereby each classifier is trained using samples describing one type of CWE vulnerability. However, this does not directly imply that a classifier trained to detect vulnerabilities of multiple CWE categories provides the same results. Therefore, we have trained additional classifiers using data sets in which multiple CWEs are combined. Again, we used a non-obfuscated, type-obfuscated, random-obfuscated and semi-obfuscated variant. Finally, performance metrics were calculated to enable us to compare the classifiers.

An overview of the steps which we have performed to answer our research questions are displayed in Figure 3.1.

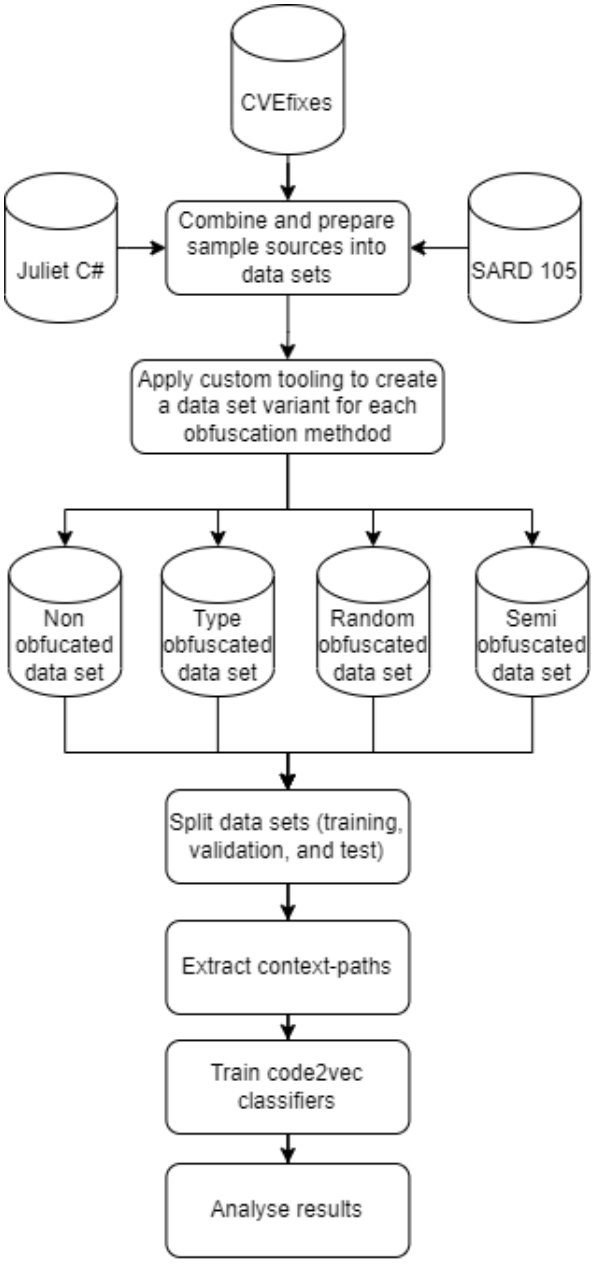


Figure 3.1: Overview describing the steps performed in our research to be able to answer our research question(s).

4

DATA SET PREPARATION

4.1. DATA ACQUISITION PROCESS

In order to answer our research questions, we have collected test cases and combined them in a data set. Specifically, we are interested in vulnerable- and non-vulnerable test cases which are classified as CWEs that are part of the Broken Access category and Injection category defined by the OWASP Top 10 [69]. The non-vulnerable test cases are related to vulnerable test cases, but the vulnerable parts are fixed [36]. In this chapter, the used test case sources for our data set are explained and the preprocessing steps are illustrated.

To create the data set for this research, we have used a **"data acquisition"** process in which we have selected and collected test cases (both vulnerable and non-vulnerable) from relevant sources. Within this process, the following steps have been performed:

1. **Extracting cases**: To help select the usable cases during the data acquisition step, we created our own tooling to take only the relevant cases from the sample sources.

2. **Data cleaning:** After the extraction, we performed operations on the retrieved data to create a homogeneous data set. An example operation is removing methods unrelated to a vulnerability, such as Main()-methods which start the program.

3. **Data labelling:** The process where we add identifiers to a case, explaining whether the case contains a vulnerability or not.

These steps are also illustrated in Figure 4.1. The acquisition process converts the test cases into samples, which we can use as input for our code2vec model. A test case can contain multiple vulnerable or non-vulnerable methods and therefore it is possible one test case can result in multiple samples. The tooling has been made available on GitHub [77].

As explained in Section 2.4 there are various parties and projects which can be used to track known vulnerabilities and make software developers more aware to prevent introducing new vulnerabilities. We have used four test case sources to create our data set, three of which are from the NIST SAMATE project (SARD Test Suite 105, Juliet C# Test Suite, and the Hasan Test cases) and one source is manually created from data retrieved from the NVD using the CVEfixes tool.

Most of these test case sources' samples are synthetic, which means that these test cases have been created manually or generated by programs. Only the cases from the NVD

project are found within production software. The SAMATE project, a project of the NIST, provides a collection of vulnerability test cases that can be used to perform evaluations on security tools that have been bundled in test suites [36]. However, some test cases are not bundled in a test suite, such as the Hamda Hasan test cases. All test cases can be filtered on CWE and programming language. The SAMATE project of the NIST currently provides two C# test suites containing various vulnerabilities [78]. The test suites and the associated data acquisition steps for these test case sources are further described in Section 4.3.1 and Section 4.3.2. Additionally, the same is done for the Hasan test cases in Section 4.3.3 and NVD/CVEFixes cases in Section 4.3.4. However, in the following three subsections we describe each processing step in general.

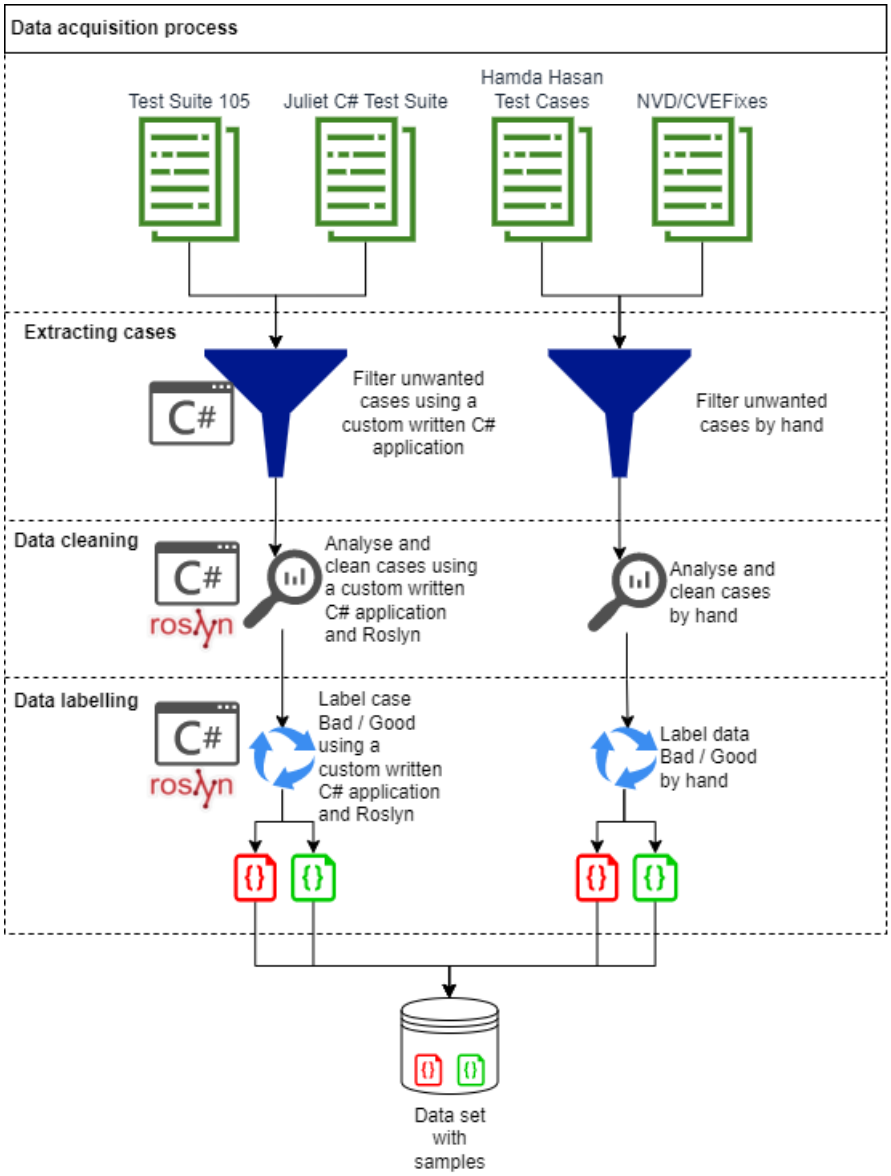


Figure 4.1: Data acquisition process.

4.1.1. Extracting cases

Each test case source contained test cases describing various CWE vulnerabilities. Therefore, during the "extracting cases" step we had to remove all unwanted test cases describing CWE vulnerabilities that were not within our scope. Additionally, the test cases could contain vulnerabilities which span multiple parts of a program, such as a vulnerability within a combination of methods or classes. Because the code2vec model only accepts individual methods, we had to remove these test cases. This was done by hand for the Hasan test cases and NVD/CVEfixes cases. We wrote custom C# software to automate this step for the SARD Test Suite 105 and Juliet C# Test Suite. The result of omitting some of these test cases for our data set is described per case source in the corresponding subsections in Section 4.3.

4.1.2. Data cleaning

Each test case source uses its own formatting and structure and therefore we had to clean the test cases before the cases could be added to our data set. This way, we are eventually able to create a homogeneous data set. This step is mainly necessary because of the Juliet C# Test Suite. This suite contains test cases which included additional classes with support methods [37]. However, we had to remove these support methods as the code2vec model only accepts single methods. The removal of these support methods is described in more detail in Section 4.3.2. Furthermore, the Juliet C# Test Suite cases contain Main- and 'runTest' methods, which can be used when testing security analysis tools. However, in our case these methods are unwanted and have been removed.

4.1.3. Data labelling

The Test Suite 105 and Juliet Test Suite contained thousands of test cases and therefore we wrote a program to label the samples by changing the method names to 'Bad' or 'Good', depending on whether or not the sample contained a specified CWE vulnerability, respectively. The Hasan and NVD/CVEfixes were smaller in size and therefore these test cases were labelled by hand using the same naming pattern.

4.2. TOOLING

Two of our four used sample sources were relatively small and therefore have been cleaned and labelled by hand for this project. However, the SARD Test Suite 105 and Juliet Test Suite contain thousands of test cases which would take too much time to clean and label by hand. A convenient solution would be to modify the generator scripts which have produced the test cases. However, no generator script files have been published for both test suites. Therefore, we needed a parser which enabled us to analyse and subsequently alter the source code of our samples for cleaning, labelling, and obfuscation purposes. We have chosen our parser by considering the two most active and up-to-date (based on GitHub commits and activity) parsers for C#: Roslyn and Antlr.

Roslyn [79] is Microsoft's open-source .Net Compiler Platform. C# is a programming language which targets the .Net framework. Therefore, Roslyn enables developers to build code analysis tools and perform syntactic- and semantic analysis of C# code (and additionally Visual Basic (VB) code, as VB also targets the .Net framework). Besides these analyses, Roslyn can also be used to perform other tasks, such as code refactoring, which are outside the scope of this project.

Another commonly used tool to parse source code is Antlr [80]. Antlr can be used to parse source code into an Abstract Syntax Tree (AST), is able to subsequently modify the AST, but lacks a pretty-printer, which is able to convert ASTs back to source code. Therefore, a custom C# pretty-printer should be created when using Antlr. Accordingly, we have used Roslyn in favor of Antlr, because Roslyn does come with a pretty-printer for C# code. Additionally, Roslyn is the official open-source implementation of the C# compiler and with each new version of C#, Roslyn is upgraded consistently and is therefore always up-to-date.

4.2.1. ROSLYN

To better understand why we needed a parser like Roslyn and to understand how we were able to obfuscate our samples, we describe how Roslyn works in this subsection.

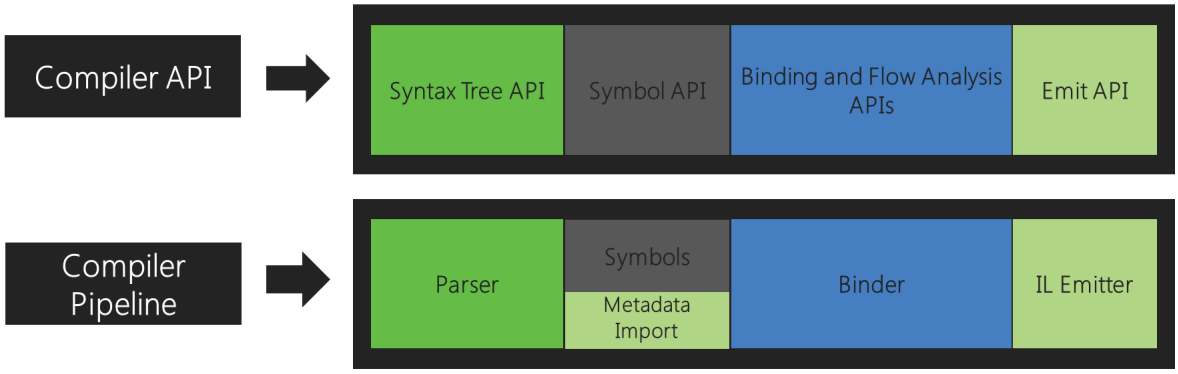


Figure 4.2: Roslyn compiler pipeline and (mirrored) compiler API. Adapted from original image by [79].

Every stage of the Roslyn compiler pipeline is an individual element, as shown in Figure 4.2). The first (parse) stage tokenizes and parses source code into syntax following the C# (or Visual Basic) grammar. The second (declaration) stage forms named symbols by analysing source and imported metadata. The third (bind) stage matches identifiers, found in the source code, to symbols. The final (emit) stage releases an assembly containing all

information aggregated by the compiler.

Roslyn exposes parts of the compiler via APIs, mirroring the compiler pipeline (as shown in the top row of Figure 4.2). The Syntax Tree API allows us to create syntax trees from source code, modify syntax trees and transform them back to source code. The Syntax Tree API is therefore an API exposing parts of the Parser. Additionally, for the obfuscation of variables we have used the Symbol- and Binding APIs. These APIs are able to obtain information about symbols. All namespaces, types, methods, properties, fields, parameters and local variables are represented as a symbol. In particular, this information is needed during our type obfuscation process (see Section 4.4), as (for example) the type of implicit typed local variables are determined by the compiler and cannot be determined by lexical analysis alone. For example, this can be seen in Listing 4.1. Line three of Listing 4.1 contains an explicitly typed integer local variable, whereas the next line contains an implicit typed local variable, which is also an integer type variable. Additionally, it is not clear what the type of the methodResult variable is without looking up the return type of the GetFifty()-method.

Listing 4.1: It is easier to detect the type of an explicitly typed local variable compared to an implicitly typed local variable.

```
1  public static int GetFifty()
2  {
3      int x = 5;
4      var y = 10; // implicitly typed local variable
5      return x * y;
6  }
7
8  public static void Main()
9  {
10     var methodResult = GetFifty(); // implicitly typed local
           variable
11 }
```

Roslyn uses syntax trees as the primary structure to represent code. These syntax trees are full fidelity trees, meaning syntax trees can be converted into code and vice-versa, without losing any elements, including lexical tokens, whitespaces, and comments. The syntax trees are also immutable and thread-safe, and therefore to modify trees Roslyn provides methods to modify and create new syntax trees. Each syntax tree is composed of three elements [79]:

1. Syntax nodes: represent syntactic constructs such as statements, declarations, expressions, and clauses. All syntax nodes are non-terminal nodes and therefore always have other nodes and tokens as children.

2. Syntax tokens: represent keywords, identifiers, literals, and punctuation. These tokens are terminal nodes and therefore are never parents of other nodes or tokens.

3. Syntax trivia: represent parts of the source code that are mostly unimportant for understanding the code, such as whitespace, comments, and preprocessor directives. These trivia are terminal nodes and therefore are never parents of other nodes or tokens.

The Main()-method of Listing 4.1 is shown in Figure 4.3 as a tree parsed by Roslyn. The nodes are represented as blue elements, the tokens as green elements, and the trivia as white and grey elements.

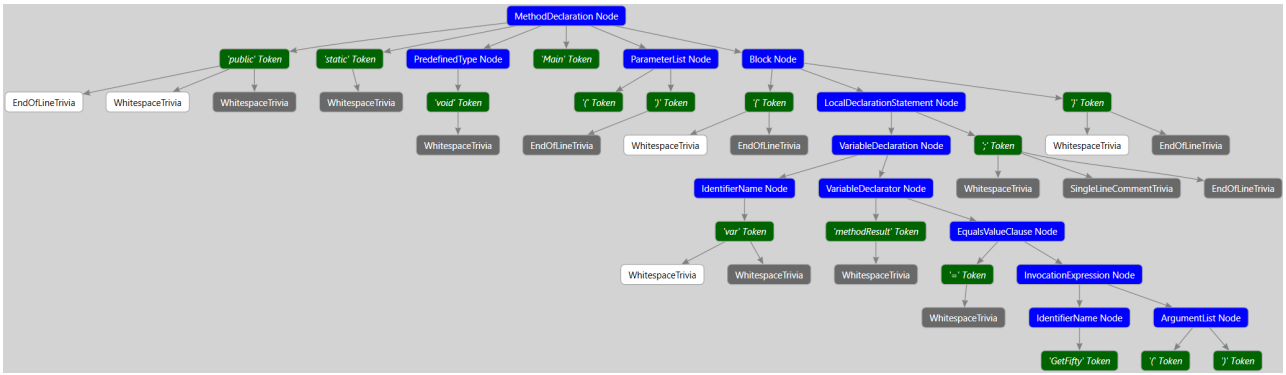


Figure 4.3: Roslyn tree of the Main()-method of Listing 4.1.

We have created a .Net C# project which reads all files of the Juliet Test Suite and SARD Test Suite 105, filters and skips samples which describe a vulnerability distributed over more than one method, and performs cleaning and labelling operations. We use the Roslyn APIs to perform the cleaning and labelling operations. This is described in more detail for the Juliet Test Suite and SARD Test Suite 105 respectively in Section 4.3.2 and Section 4.3.1.

4.3. Test case sources

In this section, we further elaborate on the test case sources and highlight the specific data acquisition steps required for the particular test case source. We first discuss the two SAMATE test suites. Next, we discuss the SAMATE Hasan test cases and the NVD/CVEFixes cases.

4.3.1. SAMATE: SARD Test Suite 105

To better represent a wider variety of programming languages, the NIST has developed a test case generator that can produce test suites for multiple programming languages without the need of rewriting a program's basis [38]. SARD Test Suite 105 is a follow-up project of the SARD Test Suite 103 [81], released at the end of 2015 and provides test cases for C# instead of PHP. It consists of 32003 test cases that cover nine different CWEs. However, four of these CWEs are not included in the Broken Access Control and Injection OWASP categories and therefore we have omitted these cases. This reduces the number of test cases we have used to 31992. The NIST provides the data set [82] and an XML file that describes every test case, including the file location and whether or not the specific test case contains a flaw, and if so, on which line it can be found. The source code of the C# test case generator that has been used to create Test Suite 105 can be found on GitHub [83].

Data acquisition steps

During the data acquisition, we cleaned and labelled each sample using our custom tooling. For the SARD Test Suite 105, we read the supplied XML manifest file to find and get every test case categorized under a CWE we are interested in. Next, we removed all test cases that span multiple methods, resulting in test cases containing only one method. Additionally, we labelled all test cases containing a vulnerability with 'Bad' and all other test cases as 'Good'. We did not have to change anything else in the samples, even though the potentially vulnerable variables were all named 'tainted', with each variable made unique by using a number suffix (e.g. tainted_1). However, both the safe and vulnerable test cases used this naming pattern and therefore it is not possible to determine a vulnerable or safe sample by examining the variable naming. The total number of suitable samples from the SARD Test Suite 105 that we have extracted and the corresponding number of safe and vulnerable samples can be found in Table 4.1.

Table 4.1: SARD Test Suite 105 used samples per CWE

CWE	Number of test cases	Number of extracted samples	Extracted safe samples	Extracted vulnerable samples	OWASP Category
CWE 22 - Path Traversal	2232	2088	792	1296	Broken Access Control
CWE 78 - OS Command Injection	1860	1740	570	1170	Injection
CWE 89 - SQL Injection	20460	19140	7440	11700	Injection
CWE 90 - LDAP Injection	1860	1740	570	1170	Injection
CWE 91 - XML Injection	5580	5220	2880	2340	Injection

4.3.2. SAMATE: Juliet Test Suite

The Juliet Test Suite [39] is a test suite originally developed by the National Security Agency's Center for Assured Software (CAS) and contains over 90 000 test cases covering Java and C/C++ vulnerabilities, and since August 2020, the NIST has released their first version of the Juliet C# Test Suite, which contains 28 942 test cases. Wagner and Sametinger have demonstrated how to use the Juliet Test Suite by utilising various free security scanners [40]. The C# Juliet Test Suite [84] contains 105 different CWE category vulnerabilities, divided into separate folders. The folders are subdivided into subfolders whenever more than 1000 test case files are available for one CWE.

Data acquisition steps

Just as the SARD Test Suite 105, we have used our custom tooling to clean and label each sample. To remove test cases that span multiple methods, we had to utilise the naming of the files. Each file contains methods which are formatted according to a specified template, described by the NSA as "Flow Variants" [37]. We have used these flow variants to determine whether or not the test case relied on multiple methods and removed the test cases that contained unwanted flow variants. Additionally, we removed all Main()- and runTest() methods because their only purpose was to run the test cases, which, in our case, is not needed.

For the SARD Test Suite 105 we were able to provide the total number of available test cases and the number of samples we could use in our data set (see Table 4.1). However, we were not able to provide the total number of available test cases for the Juliet C# Test Suite. This is because we are unable to make a distinction between methods which make up a (vulnerable or non-vulnerable) sample and methods which are helper methods and therefore not samples on its own. In addition, there is no documentation published for the C# version of the Juliet Test Suite which could provide this information.

The next step was to replace all references to methods from the custom 'IO' class defined in the Juliet Test Suite. This class contains methods to print data to the console, log data to a logger-library (NLog), and obtain a database connection. The method references for logging and printing data have been replaced by the Console.WriteLine()-method which is a default method defined in the .Net framework. Additionally, we have replaced the methods to obtain a database connection with an SqlConnection object creation, which

is a default class defined in the .Net framework.

The last cleaning step involved a naming convention for some variables, as some variables could indicate whether a method was vulnerable or not. For example, variable names were either named goodSqlCommand or badSqlCommand. This phenomenon can be seen in the Juliet Test Suite sample of Listing 2.2. We have renamed these flaw indicators by omitting the 'good' or 'bad' part from the variable name.

After the cleaning process, we have labelled each method by validating whether the method name contained 'bad' or 'good' and renamed the method accordingly, omitting all other parts from the method name. The resulting number of samples are shown in Table 4.2.

Table 4.2: Juliet C# Test Suite 1.3 used samples per CWE

CWE	Number of extracted samples	safe samples	vulnerable samples	OWASP Category
CWE 23 - Relative Path Traversal	170	100	70	Broken Access Control
CWE 78 - OS Command Injection	170	100	70	Injection
CWE 80 - Basic XSS	306	180	126	Injection
CWE 83 - Improper Neutralization of Script in Attributes in a Web Page	153	90	63	Injection
CWE 89 - SQL Injection	729	540	189	Injection
CWE 90 - LDAP Injection	170	100	70	Injection
CWE 94 - Code Injection	270	200	70	Injection
CWE 113 - HTTP Request/Response Splitting	729	540	189	Injection
CWE 284 - Improper Access Control	30	18	12	Broken Access Control
CWE 470 - Unsafe Reflection	153	90	63	Injection
CWE 566 - Authorization Bypass Through User-Controlled SQL Primary Key	17	10	7	Broken Access Control
CWE 601 - URL Redirection to Untrusted Site ('Open Redirect')	153	90	63	Broken Access Control
CWE 643 - XPath Injection	270	200	70	Injection

4.3.3. SAMATE: HASAN TEST CASES

Not all test cases are combined in a test suite within the SAMATE project, such as the Hasan test cases. Individual test cases not united in a test suite are donated by various parties and describe a variety of security vulnerabilities in various programming languages [36]. For this research, a small amount of test cases (15 unique cases) donated by Hamda Hasan are interesting, because they are written in C# and contain some vulnerabilities which we are interested in (CWE 78, CWE 79, and CWE 89). We have inspected these samples, but encountered various problems:

Three out of six CWE 89 test cases (61774, 61775, and 61776) did not contain any CWE 89 vulnerability. For example, Listing 4.2 describes the code snippet of test case 61774 that was claimed by Hasan to contain an SQL injection vulnerability on line 4 [85]. However, this snippet is not vulnerable for SQL injection attacks because it uses SQL parameters. We have informed the NIST SAMATE team and proposed to change the status of these test cases to 'deprecated'. As of 23 August 2022, the NIST team has done so, and published new versions of test cases 61774 [86], 61775, and 61776 with state 'good' instead of 'bad'.

Furthermore, some of the Hasan test cases could not be used in this project because these test cases were '.aspx' files. These '.aspx' files describe ASP.NET web pages which combine the frontend web languages HTML and CSS, together with server code written in C# or Visual Basic. Therefore, the vulnerabilities in these test cases depend on the combination of frontend- and backend (C#) code which are outside the scope of this project (we only use separate methods describing a vulnerability).

Because of these problems, we had to omit all three CWE 79 vulnerability samples and three out of six CWE 89 vulnerability samples. All three CWE 78 test cases are used in our data set. An overview of the used samples per CWE is shown in Table 4.3.

Table 4.3: Used Hasan samples per CWE

CWE	Number of used samples	safe samples	vulnerable samples
CWE 78 - OS Command Injection	3	0	3
CWE 89 - SQL Injection	3	3	0

Listing 4.2: Test case 61774 (version 1.0.0) falsely claimed to have an SQL injection vulnerability

```
1  public int updateTable(SqlConnection conn, string username, string
       newEmail)
2      {
3          SqlCommand command = conn.CreateCommand();
4          string updateQuery = "UPDATE Account SET [email]=@newEmail
               WHERE user=@username";
5          command.CommandText = updateQuery;
6
7          SqlParameter dbPramUser = new SqlParameter();
8          dbPramUser.ParameterName = "@username";
9          dbPramUser.SqlDbType = SqlDbType.VarChar;
10         dbPramUser.Value = username;
11         command.Parameters.Add(dbPramUser);
12
13         SqlParameter dbPramEmail = new SqlParameter();
14         dbPramEmail.ParameterName = "@newEmail";
15         dbPramEmail.SqlDbType = SqlDbType.VarChar;
16         dbPramEmail.Value = newEmail;
17         command.Parameters.Add(dbPramEmail);
18         int rowsAffected = 0;
19         try
20         {
21             rowsAffected = command.ExecuteNonQuery();
22         }
23         finally
24         {
25         }
26         return rowsAffected;
27     }
```

4.3.4. NVD AND CVEFIXES

The NVD contains natural vulnerabilities and is therefore interesting to use together with the artificial SAMATA code samples. The NVD, however, does not register the programming language of the code in which the vulnerability was found. Therefore, researchers have to pick vulnerabilities for a specific programming language by hand and the code examples containing the vulnerability [41]. Additionally, this would require knowledge about which language is used in the software projects available in the NVD. The CVEfixes research [42] has provided a tool to automate this manual labour. CVE records of open-source projects are updated with a link to the relevant code change (commit) on a source code repository (such as Gitlab, GitHub, and Bitbucket) whenever a vulnerability is fixed. The tool uses these links and downloads this information for vulnerabilities fixed in open-source projects, together with the information provided by the NVD. Next, the vulnerable and fixed code is subject to the computation of code-level metrics, e.g. complexity, and further details are added, such as the programming language of the source code. All this data is subsequently transformed into a queryable SQLite3 database. The Entity-Relationship Diagram of the resulting database is shown in Figure 4.4.

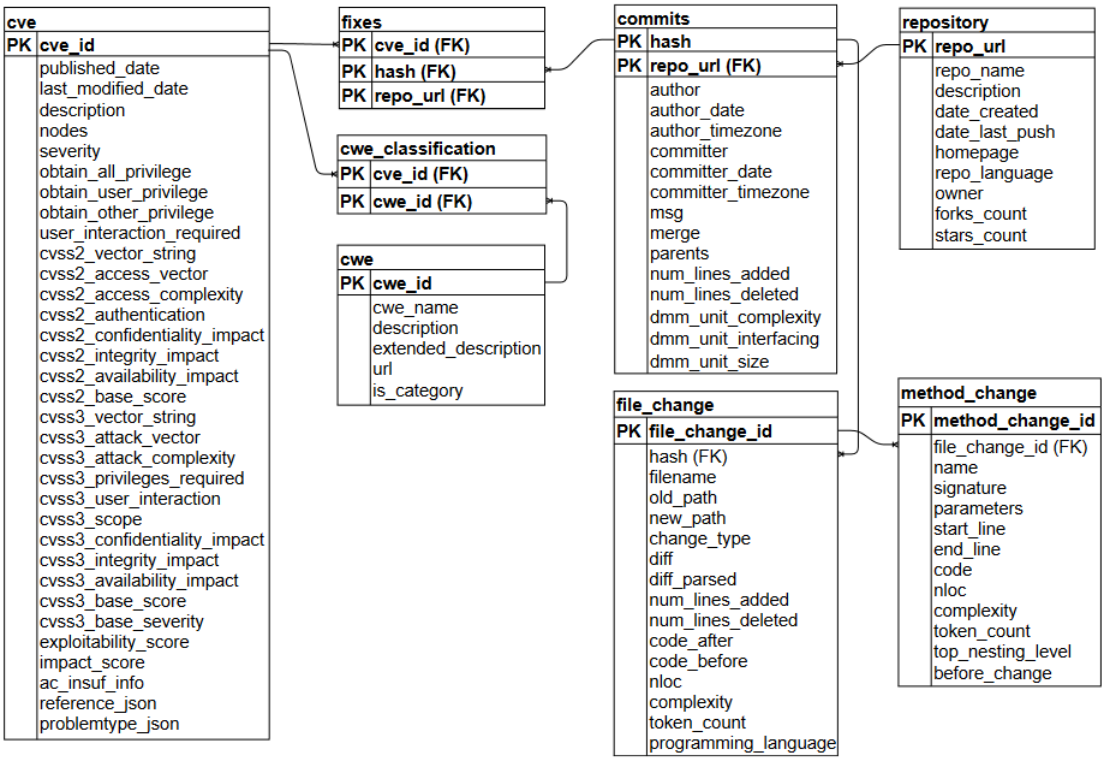


Figure 4.4: Entity-Relationship Diagram of the CVEfixes database. However, the Primary Keys (PKs) and Foreign Keys (FKs) have not been applied to their database, as described below. Adapted from original image by [42].

DATA ACQUISITION STEPS

The CVEfixes researchers have provided a copy of their database (which was generated by Bhandari et al. during their research) which is claimed by Bhandari et al. to contain all CVE records up to 9 June 2021, with a total of 5365 CVEs (in comparison to the over 150000 CVE records stored within the NVD at that time [66]). This is because in the final processing step the CVE records for which no additional data could be found are omitted by the tool. However, due to a memory error bug, the final cleanup process did not trigger each time we ran the tool. Therefore, our database copy contained more than 180000 CVE records, which corresponds to the number of records in the NVD at our latest run of 24 July 2022 [54].

We ran into various problems using the CVEfixes tool and their database copy. These are as follows:

1. **.Git suffix problem:** Because the final cleanup process was not triggered, the repository URLs were inserted with the '.git' suffix in the 'commits' table, but without the '.git' suffix in the 'repository' and 'fixes' tables (see Figure 4.4). Therefore, joining all tables is more cumbersome. We have fixed our problem by manually concatenating the '.git' suffix in some of our SQL queries.

   This '.git' suffix problem is present because Bhandari et al. have not applied primary- and foreign key constraints to the database, despite being present in their Entity Relationship Diagram of the database (Figure 4.4). Consequently, the tool could insert the URLs of the repositories differently between the commits, repository, and fixes tables. Also, using primary- and foreign keys would improve the querying performance and would prevent problems like the '.git' suffix error. Bhandari et al. could have prevented this problem also by not saving the address with the '.git' suffix and adding the '.git' suffix only temporally when needed.

2. **Memory errors:** Because of the limited amount of physical RAM (8 GB) in our system, we ran into memory errors (among other things causing the '.git' suffix problem). These errors occurred because the CVEfixes tool loads entire database tables within the memory. However, as the NVD database is continuously growing, so will the database tables. The CVEfixes tool loads all table data into memory and simultaneously saves all calculation information in memory when processing the table data and does not take into account the possibility of a flooded memory. These memory errors could have been prevented by performing more calculations within the database on disk or processing batches, instead of loading the entire database into memory.

3. **Incorrect programming language identification:** we have compared our database copy with the Bhandari et al. copy and found that some CVEs are incorrectly assigned the incorrect programming language (within the file_change table). Our database copy contained C# vulnerabilities that were incorrectly identified as another programming language in the Bhandari database copy and vice versa. The reason for this is that Bhandari et al. have used Guesslang to determine the programming language of the files containing the vulnerabilities and corresponding fixes. However, because the maintainers of Guesslang claim an accuracy of 90% [87], it is possible the tool incorrectly assigns a programming language to a CVE entry. To mitigate this problem, we additionally queried every repository which indicated it contains C# code from

the repository table and also queried all related CVE records. The programming language stored in the repository table was retrieved from the related repository hosting platforms of each repository and was not defined by the CVEfixes tool using Guesslang. Therefore, we were able to collect more C# vulnerability samples without relying too much on Guesslang.

Utilising CVEfixes databases We have used the database copy provided by Bhandari et al. and additionally generated our own database copy by using the CVEfixes tool. We initially used the query shown in Listing 4.3 on both database copies. We used both databases to check for differences because while generating our database, we encountered memory issues (as previously mentioned) and wanted to make sure this did not have an influence on the results. This query should get all CVE fixes for C# source code. However, we did encounter problems, but not related to the memory issues. The main problem was related to the "incorrect programming language identification", as explained previously in this section. Because the identification of the programming language was unreliable and provided different results (as further explained in Appendix B), we have used the query of Listing 4.4. This query retrieves all CVE records which are related to C# repositories. This resulted in 70 unique CVE records. When combined with the results of Listing 4.3, we were able to find a total of 73 CVE records.

All 73 CVE records contained fixes which were stored on GitHub. CVEfixes extracts the repository programming language by extracting the metadata provided by GitHub. GitHub uses the open-source library Linguist [88] to discover all programming languages of the files within the repository. However, some repositories contain files written in multiple programming languages. Therefore, Linguist is able to determine the percentage of programming languages represented in the repository. Because CVEfixes only stores the programming language which is most present in the repository, we found two CVE records within the C# repositories, but actually were fixes for JavaScript code. Additionally, our queries contained one vulnerability (CVE-2010-4254) which is part of a .Net C# project. However, this CVE described a vulnerability found in a file containing C-code and therefore could not be used in our research. Therefore, our extracting method produced 70 C# CVE records, which contained 39 CVE records describing a Broken Access Control or Injection category CWE which we could potentially use to create samples for our data set.

The number of vulnerabilities per CWE category is displayed in Figure 4.5. Almost all CVEs are categorized as a single CWE, however, CVE-2022-24774 is categorized as CWE-20, CWE-22, and CWE-35.

Listing 4.3: Query to get all C# CVE fixes together with the associated CWE category

```
SELECT cve.cve_id, cwe.cwe_id, fc.programming_language
FROM file_change fc
INNER JOIN method_change mc ON fc.file_change_id = mc.file_change_id
INNER JOIN commits c ON c.hash = fc.hash
INNER JOIN fixes f ON f.hash = c.hash
INNER JOIN cve ON cve.cve_id = f.cve_id
INNER JOIN cwe_classification cwe_c ON cve.cve_id = cwe_c.cve_id
INNER JOIN cwe ON cwe.cwe_id = cwe_c.cwe_id
WHERE fc.programming_language = 'C#'
ORDER BY cve.cve_id;
```

Listing 4.4: Query to get all repositories containing C# code and related CVE ids

```
SELECT DISTINCT cve.cve_id, cwec.cwe_id, cve.published_date, f.hash,
    r.repo_url
FROM repository r
INNER JOIN fixes f ON f.repo_url = r.repo_url
INNER JOIN cve ON cve.cve_id = f.cve_id
INNER JOIN cwe_classification cwec ON cwec.cve_id = cve.cve_id
WHERE r.repo_language = 'C#'
ORDER BY cve.cve_id;
```

Transforming C# CVE records into samples To transform the CVE records into samples, we have examined each CVE record and corresponding commit(s) by hand. Each C# CVE record potentially contains both vulnerable and non-vulnerable cases we can use for our data set. However, because these cases are snapshots of production software, this also means some cases are hard to use as samples for our data set and are therefore omitted. This has multiple reasons:

- Vulnerabilities span multiple methods/classes. The code2vec model only accepts individual methods as input.
- The code containing the vulnerability/fix contains a very high cyclomatic complexity. Wallace et al. [43] suggest using a cyclomatic complexity of 10 at max and in some cases a maximum of 15. However, we encountered vulnerable code that is more than ten times the recommended complexity.
- A Git commit fixing a vulnerability also contained a lot of other non-related changes in the code.

Many times the CVEfixes database C# entries contained code which contained a combination of the reasons shown above. Also, the last two reasons can make it hard to understand the code logic and therefore makes extracting samples time-consuming and error-prone. Nonetheless, we were able to produce samples by modifying some methods to contain all logic within one method instead of multiple. Some CVE records contained multiple methods containing a vulnerability which provided multiple samples for our data set.

In the end, we were able to produce a total of 90 samples. The number of samples per CWE is shown in Table 4.4. A distinction has also been made between good and bad samples (respectively non-vulnerable and vulnerable samples). What stands out, is that we were only able to use 40 per cent of the CWE-79 CVEs. However, this can partially be explained by the fact that the unused cases included parts of front-end code such as JavaScript and cshtml files (C# HTML), which we cannot describe in single C# methods. What is also striking, is that we were not able to produce CWE-352, CWE-601, and CWE-706 samples, despite having found CVE fixes for these CWEs. This is because these fixes contain one or multiple reasons as described in the list with exclusion criteria above.

Table 4.4: Number of CVE records and extracted samples per CWE.

CWE	OWASP Category	CVEs	Used CVEs	Vulnerable samples	Safe samples	Total samples
CWE 20	Injection	6	4	13	13	26
CWE 22	Broken Access Control	13	12	25	23	48
CWE 74	Injection	1	1	1	1	2
CWE 79	Injection	15	6	7	7	14
CWE 352	Broken Access Control	2	0	0	0	0
CWE 601	Broken Access Control	1	0	0	0	0
CWE 706	Broken Access Control	2	0	0	0	0

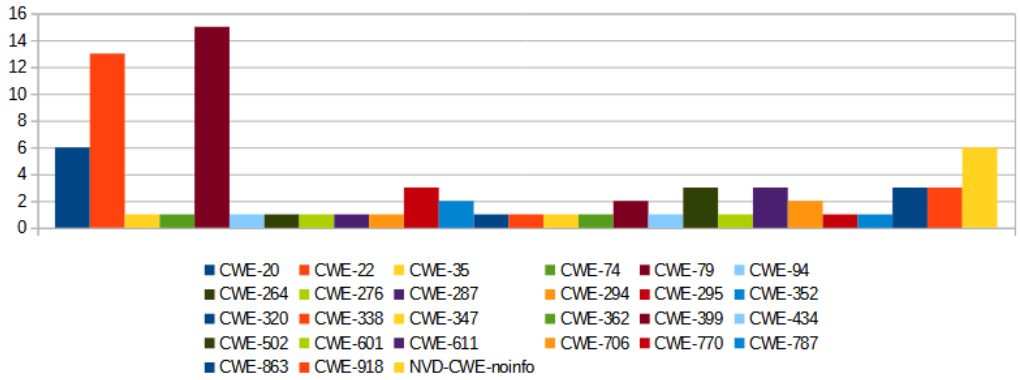


Figure 4.5: C# vulnerability CWE categories found in the CVEfixes dataset.

4.4. VARIABLE OBFUSCATION

To answer the research question of this study, we had to apply various variable obfuscation methods to the sample data set. In this section, we describe each obfuscation method we have used in more detail. However, we will first describe what kind of variables we will obfuscate in the extracted code samples.

We subdivide the following variable scopes which C# code can contain:

1. Fields: A field is a variable type which is declared directly within a class or struct. The field may be of any type (e.g. string, int, bool).

2. Property: A property encapsulates private fields (which are fields that are only accessible by code which is defined in the same class or struct), which ensures that these fields can only be accessed and updated by using the corresponding property. A property is used like a public field (therefore is accessible by code from the same assembly), but is really a special method, called an accessor.

3. Param: A parameter is a variable which allows programmers to pass data to a method. Parameters are defined after the method name, inside parentheses. The data passed to a method is called an argument, which corresponds to the parameter.

4. Local: A local variable is a variable type which is declared within the body of a method. Its scope is within the body in which it was declared. More specifically, the scope of a local variable is within the statement block in which it is declared. For example, a local variable declared within a statement block of a for-each loop is not accessible outside this for-each loop.

See Listing 4.5 for an example of each type of variable. In the following subsections we elaborate more on the specific obfuscation methods we have implemented.

Listing 4.5: C# variable types.

```
public class VariableExamples
{
    // A private field
    private string _privateString = string.Empty;

    // A property, encapsulating the _privateString field
    public string StringProperty
    {
        get { return _privateString; }
        set { _privateString = value; }
    }

    // A method with two parameters
    void Method(string testParameter, int intParameter = 5)
    {
        // modify property and therefore the encapsulated private
           field
        StringProperty = "New Value" + testParameter;

        // Declaration of a (explicitly typed) local string array
           variable
        string[] testStringArray = { "test1", "test2" };

        // Declaration of a (implicitly typed) local integer
           variable
        var counter = 0;

        foreach(var testString in testStringArray)
        {
            // Declaration of a (explicitly typed) local integer
               variable, with a scope limited to the for-each loop
               statement block.
            int arbitraryNumber = 6;
            counter += arbitraryNumber;
        }

        // The arbitraryNumber variable is inaccessible at this
           point
    }

    public void CallMethod()
    {
        // Passing arguments to a method.
        this.Method("test", 6);
    }
}
```

4.4.1. TYPE OBFUSCATION

The type obfuscation method renames each variable such that the name contains the type and scope of the variable. Therefore, the model should be able to rely more on the variable structure for making predictions (in theory). The structure for each variable scope consists out of three parts: the variable scope, the variable type, and an incremental number to make each variable name unique. The type obfuscated copy of the source code from Listing 4.5 is shown in Listing 4.6. Our obfuscation method is able to determine the type of implicitly typed local variables (which are described previously in Section 4.2) and adds this information to the variable name, further improving the potential of the type obfuscation method. An example of this can be seen on lines 17 and 19 of Listing 4.6. Additionally, our type obfuscation method also handles nullable reference types (such as "int?") and goto statements, such that the corresponding variable names are type obfuscated accordingly.

Listing 4.6: Type obfuscated copy of Listing 4.5. The comments are omitted for clarity.

```
1  public class VariableExamples
2  {
3      private string field_string_0 = string.Empty;
4
5      public string property_string_0
6      {
7          get { return field_string_0; }
8          set { field_string_0 = value; }
9      }
10
11     void Method(string param_string_0, int param_int_1 = 5)
12     {
13         property_string_0 = "New Value" + param_string_0;
14
15         string[] local_stringArray_0 = { "test1", "test2" };
16
17         var local_int_1 = 0;
18
19         foreach(var local_string_2 in local_stringArray_0)
20         {
21             int local_int_3 = 6;
22             local_int_1 += local_int_3;
23         }
24     }
25
26     public void CallMethod()
27     {
28         this.Method("test", 6);
29     }
30 }
```

4.4.2. RANDOM OBFUSCATION

We have used a random obfuscation method to theoretically make the model rely more on the code structure instead of relying on trends learned from variable names. This is done by replacing each variable with a string of 10 random characters using the following alphabet: "ABCDEFGHIJKLMNOPQRSTUVWXYZ". An example of a random obfuscated code snippet can be found in Listing 4.7.

Listing 4.7: Random obfuscated copy of Listing 4.5. The comments are omitted for clarity.

```
public class VariableExamples
{
    private string ZDTWCDBWIJ = string.Empty;

    public string MIVZBWYKIK
    {
        get { return ZDTWCDBWIJ; }
        set { ZDTWCDBWIJ = value; }
    }

    void Method(string CCGLCTMXOE, int XIHUPUDPMM = 5)
    {
        MIVZBWYKIK = "New Value" + CCGLCTMXOE;

        string[] QJBZCJFZQD = { "test1", "test2" };

        var EVBWAZNQKA = 0;

        foreach(var TRWUIQSNME in QJBZCJFZQD)
        {
            int FPQEAPBLOR = 6;
            EVBWAZNQKA += FPQEAPBLOR;
        }
    }

    public void CallMethod()
    {
        this.Method("test", 6);
    }
}
```

4.4.3. SEMI TYPE OBFUSCATION

For our semi obfuscation method we have obfuscated 50% of the variables using the type obfuscation method. The other 50% have been left non-obfuscated. Therefore, the model may be able to use variable names when they are informative, but will also be able to rely on the variable structure for making predictions. An example of a semi obfuscated code snippet can be found in Listing 4.8.

Listing 4.8: Semi (type) obfuscated copy of Listing 4.5. The comments are omitted for clarity.

```
public class VariableExamples
{
    private string field_string_0 = string.Empty;

    public string StringProperty
    {
        get { return field_string_0; }
        set { field_string_0 = value; }
    }

    void Method(string param_string_0, int intParameter = 5)
    {
        StringProperty = "New Value" + param_string_0;

        string[] testStringArray = { "test1", "test2" };

        var local_int_1 = 0;

        foreach(var testString in testStringArray)
        {
            int arbitraryNumber = 6;
            local_int_1 += arbitraryNumber;
        }
    }

    public void CallMethod()
    {
        this.Method("test", 6);
    }
}
```

4.5. DATA SETS

In this section, we describe the structure of our data sets, which were used to train our classifiers. We have created our data sets by utilising the data acquisition process (Section 4.1) on the four selected test case sources. We have used various CWE categories to train models, depending on the number of samples for each CWE (as summarised in Appendix A). However, most CWE sample categories were omitted. First, CWE sample categories which were relatively small (e.g. CWE 80, CWE 83, CWE 94, CWE 284, CWE 470, CWE 566, CWE 601, and CWE 643) were not used during model training. This is done because the low number of samples has a high change of classifiers underfitting, meaning they do not have enough data to learn a trend within the data to classify the data correctly.

Next, we have omitted CWE sample categories which were imbalanced. This is the case for the CWE78 and CWE90 samples. Even when the samples from both the Juliet Test Suite and Test Suite 105 are combined, the safe samples only represent 35% of the data set. We could have reduced the number of vulnerable samples to increase the percentage of safe samples, but by doing so, the data set would become too small, increasing the chance of underfitting. In addition, the CWE78 and CWE90 sample categories contained less than 2000 samples and therefore could be classified as relatively small sets as well.

Finally, the CWE22, CWE89, and CWE91 samples remain and therefore have been used to train our classifiers. An overview of these samples are shown in Table 4.5, which also shows our final base data sets.

Table 4.5: The data sets used for our experiments. These were created using the SARD Test Suite 105 cases, Juliet Test Suite cases, Hasan Test cases, and NVD cases combined.

CWE	Sources	Total samples	Safe samples	Vulnerable samples	OWASP Category
CWE 22 - Path Traversal	Test Suite 105	2088	792	1296	Broken Access Control
CWE 22 - Path Traversal	NVD/CVEfixes	48	25	23	Broken Access Control
CWE 89 - SQL Injection	Test Suite 105 + Juliet Test Suite + Hasan	19872	7983	11889	Injection
CWE 91 - XML Injection	Test Suite 105	5220	2880	2340	Injection
CWE 89 + 91 - SQL + XML Injection	Test Suite 105 + Juliet Test Suite + Hasan	25092	10863	14229	Injection

Each CWE category is represented in its own dedicated data set. However, we have created two separate CWE22 data sets, separated in only synthetic samples and natural samples. The first is used to train, validate, and test our CWE22 classifiers. The second is to test the performance of these trained classifiers on natural samples, compared to the synthetic samples of the first CWE22 data set. The results are shown in Section 5.4. Furthermore, we have combined the CWE89 and CWE91 data sets to create an additional data set which we have used to train classifiers with which we should be able to answer RQ3. We have chosen

to combine CWE89 and CWE91 samples into one data set because they are both best represented in terms of quantity and are also the best balanced CWE categories of our extracted CWE category samples. Additionally, CWE89 and CWE91 are both injection-based vulnerabilities. Although the CWE22 category contains just over 2000 samples, we have used this set to be able to (partly) answer RQ2, as this is the best represented CWE category within the Broken Access Control OWASP category. Additionally, from the low amount of natural samples we have extracted from the NVD, CWE22 was best represented (48 samples). The performance of the trained CWE22 classifiers when validated on these natural samples is described in Section 5.4.

For each of our four data sets we have generated three additional obfuscated versions in order to examine the impact of variable names (type-, random-, and semi obfuscated), as depicted in Figure 4.6. These obfuscated versions have been obfuscated by using a custom .Net C# application utilising Roslyn.

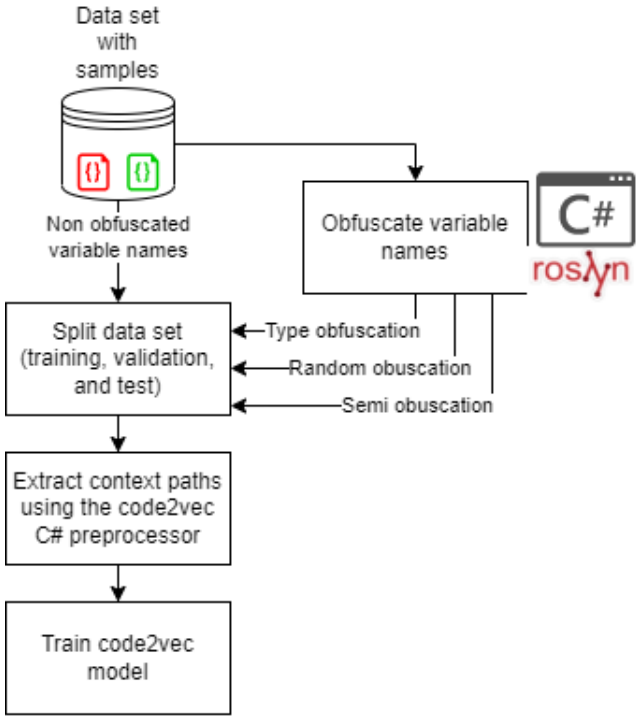


Figure 4.6: Training the code2vec model utilising our data set.

4.5.1. Splitting the data set

The data sets have been subdivided in a train-, validation-, and test subset. The train set is used to train the model, whereas the validation set will be used to fine-tune the model and can be used to measure the performance of the model in an early stage. Finally, the test set is an unseen set for the model and therefore can be used to measure the final performance of a fully trained model (as explained by Russel and Norvig [44]). There is no unanimity about which data set split percentage is best, but related studies [15; 16; 18; 19] use a split percentage of 80-10-10, whereby the 80% split refers to the train set. Therefore, we have randomly picked samples and created a data set split with the 80-10-10 split percentage.

4.6. CODE2VEC PREPROCESSING

Now that we have our data sets with samples, we have used the code2vec C# preprocessor tool to transform these samples into context paths that are used as input by the code2vec model. To better understand what happens during the preprocessing, we describe some definitions from the code2vec paper [15]:

- **AST path:** An AST path is a path between two terminal nodes within an AST, passing through at least one intermediate nonterminal node.
- **Path-context:** Given an AST path p, its path-context is captured in the triplet $\langle x_s, p, x_t \rangle$, where x_s is the start node and x_t the end node of path p.

According to these definitions we can capture the statement: 'y = 15' in the following path-context:

$$\langle y, (NameExpr \uparrow AssignExpr \downarrow IntegerLiteralExpr), 15 \rangle$$

Within this path-context, the $\uparrow$ and $\downarrow$ are movement directions in the AST (up or down the AST). A representation of the path-context of 'y = 15' is visualised in Figure 4.7.

The code2vec C# preprocessor parses the code snippet into an AST (using Roslyn) and subsequently separates all present methods. Afterwards, the path-contexts of each method are generated. Next, the path-contexts can be used as input for the code2vec model to learn which paths are most important using the attention mechanism [15] and store this information in a single code embedding. This code embedding is ultimately used by the (code2vec) classifier to make a prediction, in our case vulnerable or non-vulnerable.

Alon et al. [15] have experimented with alternative path-context and attention designs and concluded that omitting the start node and end node of a path resulted in lesser model performance when trained on predicting method names. All variable names only appear as start node or end node in a path-context and therefore removing variable names would decrease the performance of the model.

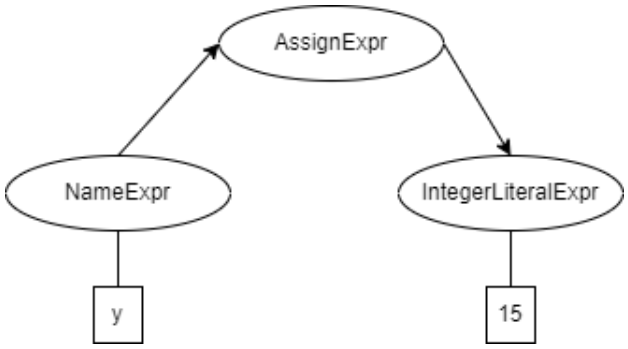


Figure 4.7: Path-context of 'y=15' visualised in a (partial) AST. The arrows indicate the movement directions in the AST for this path-context.

4.6.1. PREPROCESSOR MODIFICATIONS

We have made some alterations to the code2vec C# preprocessor which we believe are beneficial for our purpose:

- **Retain numbers in variable names:** The code2vec preprocessor removes numbers in variable names by default. We have contacted one of the code2vec model maintainers, who has informed us that this is most likely done to decrease the embedding vocabulary and by doing this also decreases the training time [89]. However, this information is crucial in our research and therefore we changed the preprocessor to keep all variable name information in the path-contexts.
- **Omitting code comments:** Our synthetic test case samples contain comments indicating whether or not a vulnerability is present. Furthermore, fixed vulnerabilities in production test cases (extracted from the NVD) are sometimes marked via a comment. An example of a commented vulnerability is shown in Listing 4.9, a commented fix for a vulnerability is shown in Listing 4.10. Additionally, our test cases may contain informative comments such as test case description, author, and copyrights. The code2vec preprocessor extracts code comments and adds these to the bag of path-contexts. However, these comments provide information that we would rather not provide to the model, because this would enable the model to easily predict whether or not the test case contains a vulnerability (fix) based on a comment, without using the semantics of the code. Also, this information is normally not available in production code. Therefore, we have omitted all comments during the path-context generation process.

Listing 4.9: Code snippet of the vulnerability within a SARD Test Suite 105 test case

```
1 //flaw
2 string query = "SELECT * FROM '" + tainted_3 + "'";
```

Listing 4.10: Code snippet of the vulnerability fix for CVE-2016-16806

```
1 //prevent directory traversal, we should only allow access to the
      root directory of the HTTP Server.
2 if (!targetPath.IsSubPathOf(_rootDirectory))
3 {
4   context.Response.StatusCode = (int)HttpStatusCode.Forbidden;
5   return;
6 }
```

5

MODEL TRAINING AND RESULTS

5.1. TRAINING ENVIRONMENT

Training machine learning models requires a relatively large amount of computing power. Because of the many vector calculations used in machine learning, powerful Graphical Processing Units (GPUs) are commonly used for these tasks. We have used a system with 8 GB memory, an Intel i7-4700HQ Central Processing Unit (CPU) and NVIDIA GeForce GTX 770M GPU. However, this GPU uses a CUDA 3.0 architecture not supported by code2vec (as it uses TensorFlow and Keras for calculations) [90]. Therefore, calculations can only run on our CPU. Because we only have an outdated CPU available (released in 2013), we have looked at alternatives to fall back on in case training a model would take too much time. Therefore, we have considered using Google Colab [91], a service provided by Google for students, data scientists and AI researchers. However, the computing power resources provided by Google Colab are not guaranteed and can vary over time [92]. Additionally, Google Colab has a time limit of 12 hours for continuous use of a virtual machine and will also disconnect when left idle for too long. Ultimately, training the code2vec model only took 15 to 20 minutes on our system, so we did not have to use an alternative training system. Different system configurations may result in (slightly) different machine learning results. This applies to differences in hardware, software and software versions. This can be caused by differences in rounding floating point numbers, bug fixes in software, and changing functionality in software. Therefore, we have described the software (versions) we have used to train our model in Table 5.1. First, we have used our forked code2vec copy, which is available via our GitHub repository [93]. Next, the latest versions (as of 21 September 2022) of all the required packages for Tensorflow are automatically downloaded and installed while installing Tensorflow. Finally, we have used the default code2vec model hyperparameters, such as the max number of contexts to use per sample and the maximum size of the token vocabulary.

5.2. METRIC CALCULATIONS

As explained in Section 2.2, we calculate various metrics to describe the performance of classifiers. Code2vec already has built-in metric calculations, however, these are modified to better describe the performance of classifiers trained on the task of method name prediction [15]. Method names, which are used as labels, are decomposed into smaller sub-

Table 5.1: Used software to train the code2vec model

Software/library	Versions
Windows 10	21H2
Python	3.7
Tensorflow (Python library)	2.0.0
Numpy (Python library)	1.21.6

tokens by splitting multiple words when they begin with a capital letter. Thus, a method such as getProperty is decomposed into the two sub-tokens 'get' and 'property'. Therefore, the code2vec metric calculations can take into account that when the classifier predicts the method name to be 'getValue', some sub-tokens are equal (in this case 'get') and therefore the prediction is considered to be a correct prediction. This behaviour does not match our task of vulnerability detection. Therefore, we had to modify the built-in code2vec metric calculations in favour of our vulnerability detection task. We have used only two labels in our data set and did not want to consider multiple sub-tokens. Additionally, a prediction of 'good' is considered a negative prediction and a prediction of 'bad' is considered a positive prediction. The number of positive and negative predictions is used to calculate the metrics, as explained in Section 2.2.

5.3. Classifier results

We have used the data sets as described in Section 4.5 to train classifiers for CWE22, CWE89, CWE91, and a combination of CWE89 and CWE91 vulnerabilities. More specifically, we have trained four different classifiers for each data set (CWE22, CWE89, CWE 91, and CWE89 + CWE91), where each variant is obfuscated using a specific obfuscation method (non-, type-, random-, and semi-obfuscated), which we have previously described in Section 4.4. The classifiers have been trained using the respective training set and evaluated using the respective evaluation set. Next, we select the best-performing classifiers based on their F_1 score. Finally, we evaluate the performance using their respective unseen test set. Furthermore, we have trained our classifiers for 20 epochs. Most classifiers we trained and validated with the corresponding validation sets showed the best F_1 score around the tenth epoch and therefore we do not believe training the classifiers for a longer period would improve the results. Training 20 epochs on the largest data set (CWE89 + CWE91) took around 20 minutes, smaller data sets took less time. The results are shown in Table 5.2 and Figures 5.1, 5.2, 5.3, and 5.4.

In related studies, the performance of classifiers is normally compared by comparing the metric value results [15; 16; 35; 41; 45–47]. The classifier with the highest value for a given metric, for example, the accuracy or F_1 score, is deemed to be the best-performing classifier. When we compare the F_1 scores (rounded to two decimal places) of our classifiers we conclude that variable obfuscation does not improve performance. However, to be able to statistically substantiate our conclusion, we should determine whether or not the (minimal) differences in the F_1 scores are significant or not. To determine this for the F_1 scores, we need to apply a compute-intensive randomisation test and cannot use tests like matched-pair t, sign and Wilcoxon tests [48]. However, we lack the computing power and

Table 5.2: Trained classifier metric results

Classifier	Accuracy	Precision	Recall	F_1
CWE22 non-obfuscated	**0.90**	**0.86**	**1.0**	**0.93**
CWE22 type-obfuscated	**0.90**	**0.86**	**1.0**	**0.93**
CWE22 random-obfuscated	**0.90**	**0.86**	**1.0**	**0.93**
CWE22 semi-obfuscated	**0.90**	**0.86**	**1.0**	**0.93**
CWE89 non-obfuscated	0.93	0.90	**1.0**	**0.95**
CWE89 type-obfuscated	**0.94**	0.99	0.90	**0.95**
CWE89 random-obfuscated	0.93	0.89	**1.0**	0.94
CWE89 semi-obfuscated	**0.94**	**1.0**	0.90	**0.95**
CWE91 non-obfuscated	**0.96**	0.92	**1.0**	**0.96**
CWE91 type-obfuscated	**0.96**	0.95	0.97	**0.96**
CWE91 random-obfuscated	**0.96**	0.97	0.94	**0.96**
CWE91 semi-obfuscated	0.95	**1.0**	0.89	0.94
CWE89 + CWE91 non-obfuscated	**0.93**	**0.90**	**1.0**	**0.94**
CWE89 + CWE91 type-obfuscated	**0.93**	0.89	**1.0**	**0.94**
CWE89 + CWE91 random-obfuscated	0.92	**0.90**	0.98	**0.94**
CWE89 + CWE91 semi-obfuscated	**0.93**	**0.90**	0.99	**0.94**

required time to perform this test. Therefore, we have compared our results in the same manner as done by related studies [15; 16; 35; 41; 45–47] and hence we have compared the metric values based on two decimal places.

Nonetheless, the following observations are made (excluding the CWE22 classifiers):

- Non-obfuscated classifiers and type-obfuscated classifiers appear to perform slightly better than random- and semi-obfuscated classifiers (based on the F_1 score).
- Type-obfuscated classifiers have the highest accuracy (marginally)
- Semi-obfuscated classifiers appear to have the lowest recall (marginally).

When we count the number of times an obfuscation method outperforms the other classifiers for each metric, we conclude that type obfuscation achieves the best performance, and semi-obfuscation the lowest performance, as shown in Table 5.3. However, overall, obfuscation does not seem to matter, as the metric value results are very close to each other.

The CWE22 classifiers do not show any differences between non-obfuscated samples and the various obfuscated samples. However, to further analyse the impact of variable obfuscation on the CWE22 classifiers, we have conducted additional experiments regarding the nature of the samples and the size of the data set in Sections 5.4 and 5.5 respectively.

Compton et al. have examined the influence of variable obfuscation on various tasks [19]. They have used Cohen's Kappa (as described in Section 2.2) to compare the non-obfuscated and obfuscated classifiers. The task of detecting security vulnerabilities has not been analysed by Compton et al., however, we have also calculated Cohen's Kappa (κ) in order to

Table 5.3: The number of times an obfuscation method (marginally) outperformed the other obfuscation methods for a set of classifiers of the same CWE

Obfuscation method	# highest accuracy	# precision	# recall	# F_1	Total #
non-obfuscated	1	0	2	1	4
type-obfuscated	2	0	1	2	5
random-obfuscated	1	1	1	0	3
semi-obfuscated	0	2	0	0	2

compare our results with the results of Compton et al. The κ of our classifiers is shown in Table 5.4. The results show that the κ does not show significant differences between the obfuscation methods and non obfuscation. The most notable difference, however still insignificant, is the CWE91 type-obfuscated classifier which shows a κ difference of 0.05 compared to the non-obfuscated classifier.

Table 5.4: Cohen's Kappa results for our classifiers.

Classifier	Kappa
CWE22 non-obfuscated	0.81
CWE22 type-obfuscated	0.81
CWE22 random-obfuscated	0.81
CWE22 semi-obfuscated	0.81
CWE89 non-obfuscated	0.87
CWE89 type-obfuscated	0.87
CWE89 random-obfuscated	0.86
CWE89 semi-obfuscated	0.87
CWE91 non-obfuscated	0.86
CWE91 type-obfuscated	0.91
CWE91 random-obfuscated	0.87
CWE91 semi-obfuscated	0.89
CWE89 + CWE91 non-obfuscated	0.86
CWE89 + CWE91 type-obfuscated	0.86
CWE89 + CWE91 random-obfuscated	0.85
CWE89 + CWE91 semi-obfuscated	0.86

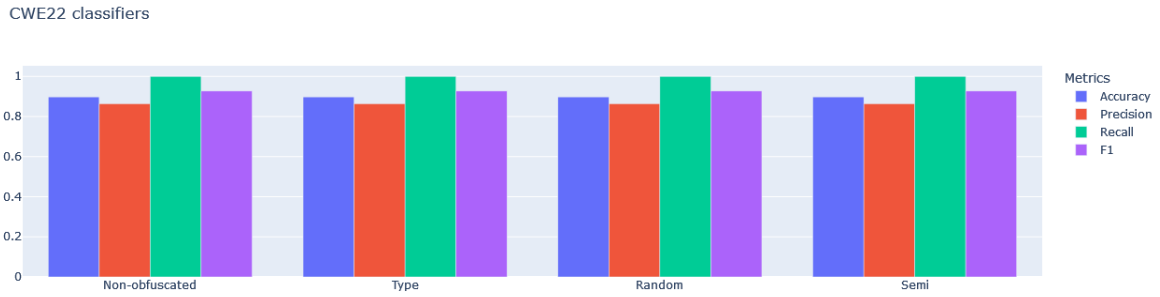


Figure 5.1: CWE22

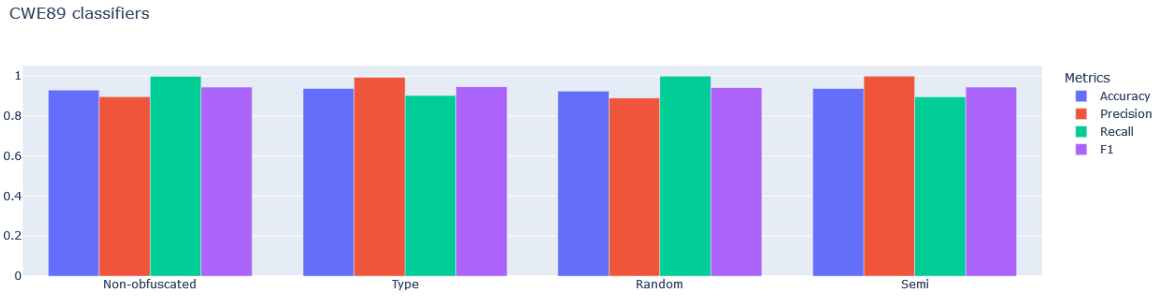


Figure 5.2: CWE89

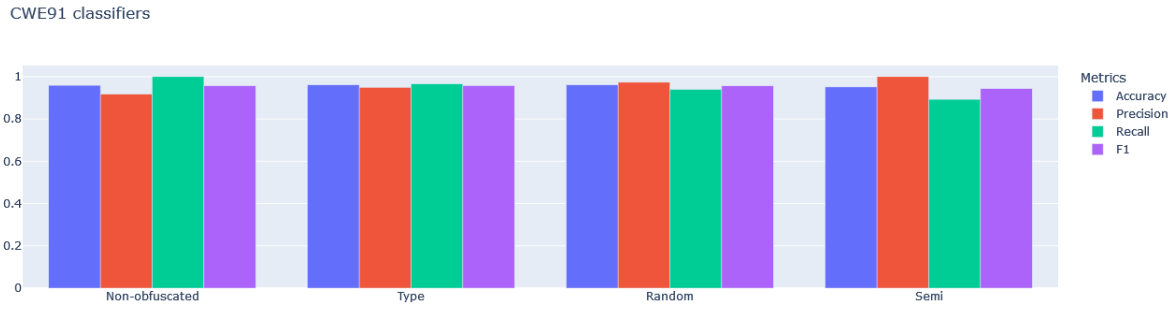


Figure 5.3: CWE91

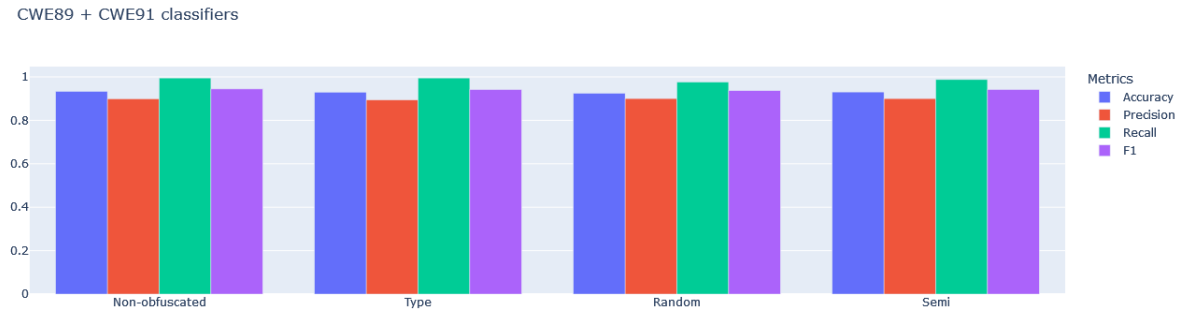


Figure 5.4: Combined CWE89 and CWE91 classifiers

5.4. Evaluation with natural samples

As described in Section 4.3.4, we have extracted C# vulnerability cases from the NVD. We believe it would be interesting to test our classifiers on natural samples, instead of synthetic samples, and compare the results. However, we could not find any CWE89 and CWE91 cases in the NVD. Therefore, we have only tested the CWE22 classifiers on natural samples. These natural samples were not used during the training (and evaluation) of our classifiers and are therefore unseen to the classifiers (Section 4.5). The results are displayed in Table 5.5 and Figure 5.5. When comparing these results with our synthetic test sets (Table 5.2), we find that the recall is still 1.0. However, accuracy and precision are almost halved (by 49% and 47% respectively) and the F_1 score is diminished by 32%. Interestingly, our observation of variable obfuscation having no impact on CWE22 vulnerabilities still applies to the natural cases, as the metric results are the same for each CWE22 classifier.

Table 5.5: CWE22 classifier results. The classifiers were trained on NVD extracted natural samples.

Classifier	Accuracy	Precision	Recall	F_1
CWE22 non-obfuscated	**0.46**	**0.46**	**1.0**	**0.63**
CWE22 type-obfuscated	**0.46**	**0.46**	**1.0**	**0.63**
CWE22 random-obfuscated	**0.46**	**0.46**	**1.0**	**0.63**
CWE22 semi-obfuscated	**0.46**	**0.46**	**1.0**	**0.63**

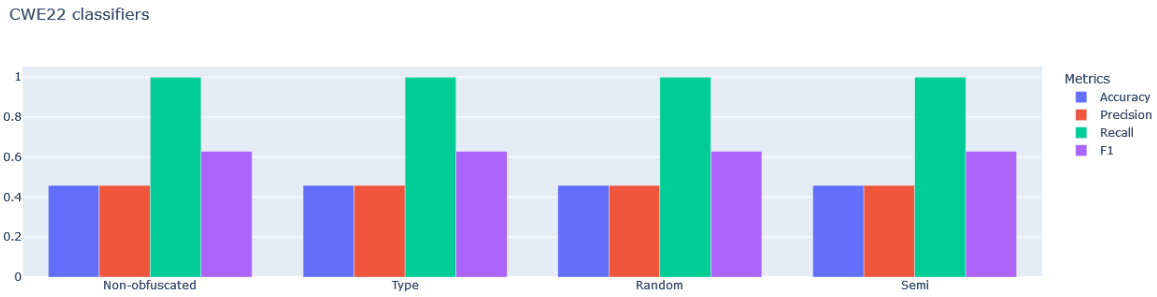


Figure 5.5: CWE22 classifier results. The classifiers were trained on NVD extracted natural samples.

5.5. Evaluating data set size

Variable obfuscation does not seem to have any effect on the results of the CWE22 classifiers (as seen in Section 5.3 and 5.4). We hypothesise this has to do with the CWE22 samples. These samples appear to be less complex than the other CWE samples and also contain fewer variables overall. To validate this hypothesis, we have reduced our CWE89 data set sizes to the size of our CWE22 data sets by randomly removing cases. Subsequently, we trained new classifiers using these reduced CWE89 data sets. The results are shown in Table 5.6 and Figure 5.6. These results show that obfuscation does have an impact on our CWE89 samples, even if the same data set size is used as our CWE22 data set. Therefore, we believe the complexity (cyclomatic complexity and number of paths) of the sample and number of variables do have an impact on the influence of variable obfuscation. In addition, the results show that variable obfuscation decreases the performance significantly on the smaller CWE89 data set, compared to our non-reduced CWE89 data set. Especially the performance of the type obfuscated classifier is reduced. However, the random-obfuscated classifier does not seem to have a significant impact on the performance and even has a slightly improved precision. The cause of this could be determined by conducting further research. For example, by applying multiple varying methods to randomly obfuscate the variable names. Nonetheless, we do not have any signs that random obfuscation would improve the overall classifier performance, as random obfuscation also does not improve the performance of our classifiers described in Section 5.3.

Table 5.6: CWE89 classifier results. The classifiers were trained on CWE89 data sets which were reduced to the same size as our CWE22 data sets.

Classifier	Accuracy	Precision	Recall	F_1
CWE89 non-obfuscated	**0.88**	0.90	**0.85**	**0.87**
CWE89 type-obfuscated	0.70	0.86	0.44	0.58
CWE89 random-obfuscated	0.87	**0.93**	0.78	0.85
CWE89 semi-obfuscated	0.76	0.76	0.70	0.73

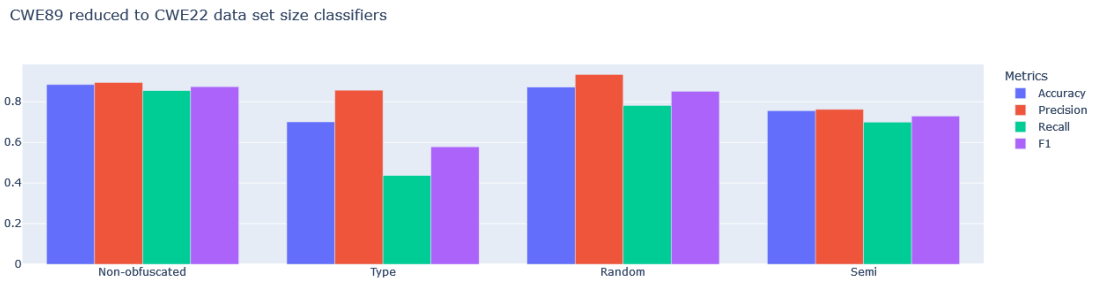


Figure 5.6: CWE89 classifier results. The classifiers were trained on CWE89 data sets which were reduced to the same size as our CWE22 data sets.

5.6. Evaluating CWE89 data set cleanup

To train the CWE89 classifiers, we have used the Test Suite 105, Juliet Test Suite, and Hasan Test cases. However, we have cleaned the samples from the Juliet Test Suite from variable names like goodSqlCommand and badSqlCommand, as described in Section 4.3.2. This is done to prevent the non-obfuscated classifier to learn from obvious indications whether the sample was vulnerable or not. However, to allow future studies to better compare their results with our results, we have also trained CWE89 classifiers using a non-cleaned data set. Compared to the "cleaned" data set, this new non-cleaned data set still contains variable names like badSqlCommand and goodSqlCommand, together with all references to methods from the custom "IO" class defined in the Juliet Test Suite. The results of our evaluated CWE89 classifiers are shown in Table 5.7. We see no essential difference when we compare these results with the metric value results of the previously trained CWE89 classifiers (Table 5.2).

Table 5.7: CWE89 non cleaned classifier results. The classifiers were trained on the non cleaned CWE89 data sets.

Classifier	Accuracy	Precision	Recall	F_1
CWE89 non-obfuscated	**0.93**	0.91	0.99	**0.95**
CWE89 type-obfuscated	**0.93**	**1.0**	0.89	0.94
CWE89 random-obfuscated	**0.93**	0.89	**1.0**	0.94
CWE89 semi-obfuscated	**0.93**	**1.0**	0.88	0.94

5.7. Interpreting attention

Normally, a neural network can be seen as a 'black box' which makes it hard to understand the decisions the network makes. However, the code2vec models' attention mechanism allows us to examine the weights given to the path-contexts and therefore makes the model partially explainable. We have performed the prediction for a vulnerable and non-vulnerable code sample (Appendix C) for each trained classifier. Thus, we were able to examine the top path-contexts with the most attention for the non-obfuscated, type-

obfuscated, random-obfuscated, and semi-obfuscated classifiers.

First, we have examined the behaviour of the attention mechanism for our CWE89 classifiers. In Figure 5.7 we have visualised the top 4 paths for a non-vulnerable sample, which were given the most attention. The top path-context is shown below and displayed in red in Figure 5.7. Path-contexts are previously described in Section 4.6. The first part is the tokenised variable name, followed by the path, and concluded with the tokenised SqlDbType enum type member VarChar. The '^' sign indicates moving up the AST, and the '_' sign indicates going down the AST.

sql|db|type,IdentifierName1^SimpleMemberAccessExpression0
^SimpleAssignmentExpression_SimpleMemberAccessExpression1_IdentifierName1,var|char

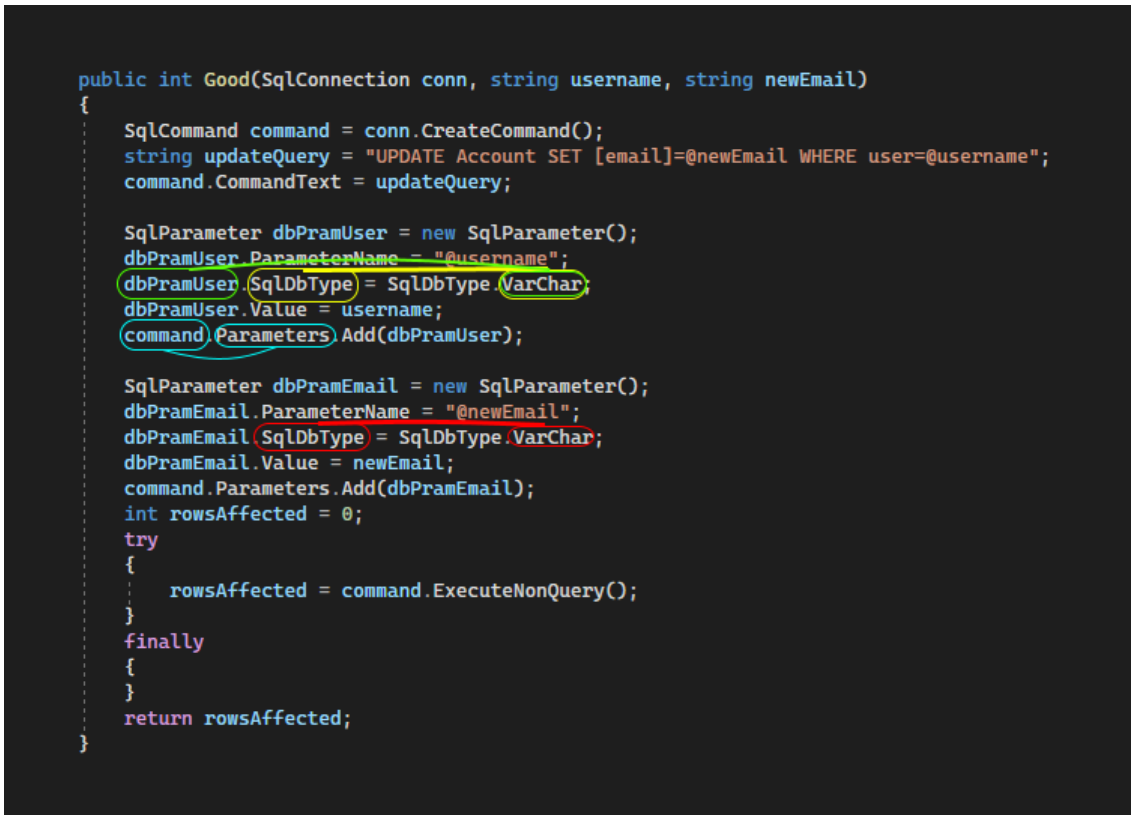
Additionally, Figure 5.8 shows a sample related to the non-vulnerable sample, which we have made vulnerable by removing the parameterised query and inserting the unsafe username- and newEmail variables directly into the query string.

The non-vulnerable example shows us that most attention is given to the SqlParameter variables, which could indicate the model detects that we use parameterized SQL variables and therefore predicts the sample is safe. In contrast to this, the vulnerable example shows us that most attention is given to the username variable. This could indicate that the model is able to detect that the username variable is not used safely and therefore the sample is vulnerable.

We noticed that each CWE89 classifier follows the same pattern: the path-contexts of the good sample, which are related to the parameterised SQL variables, get the most attention. Subsequently, the path-contexts of the bad sample, which are related to the parameter values get the most attention. However, despite the path-context attention being the same, the random- and semi-obfuscated classifiers were not able to correctly predict the vulnerable sample as vulnerable.

We performed the same predictions for the CWE22, CWE91, and combined CWE89 + CWE91 classifiers, which gave the following insights:

- All CWE22 classifiers provided the same top 4 path-contexts for our used test samples (Appendix A). No distinction was made between the bad or good samples and additionally, all samples were predicted as vulnerable.
- All CWE91 classifiers provided equivalent top 4 path-contexts for our used test samples (Appendix A). The top 4 path-contexts of the non-vulnerable samples were the path-contexts that were related to the LINQ "Where"-method, which is used to filter any malicious characters from the input. Additionally, all classifiers were able to correctly predict whether the used samples were vulnerable or non-vulnerable.
- For the CWE89 + CWE91 classifiers we utilised both the CWE89 and CWE91 samples (Appendix A). The attention given to the path-contexts was similar to the separate CWE89 and CWE91 classifiers; Most attention was given to the parts of the code which filtered input variables.

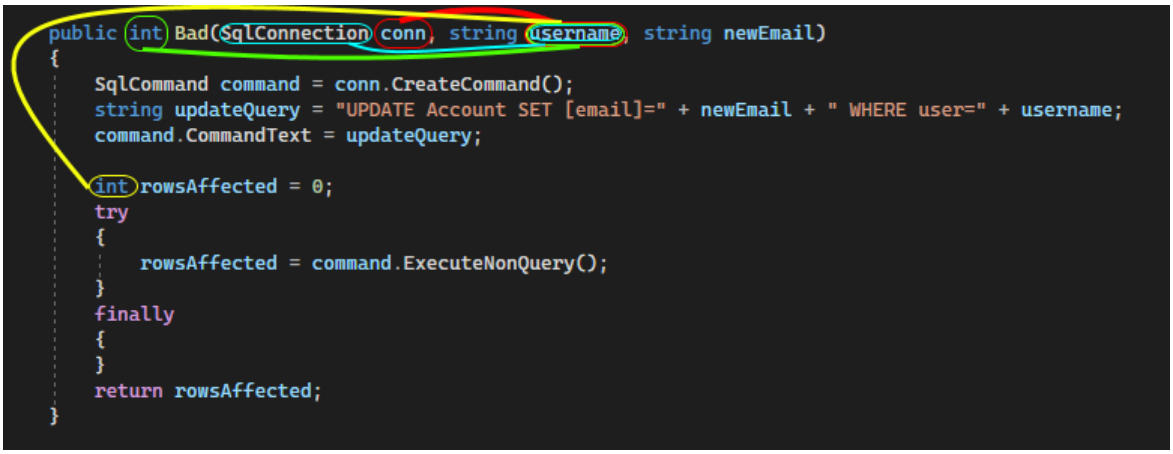


```
public int Good(SqlConnection conn, string username, string newEmail)
{
    SqlCommand command = conn.CreateCommand();
    string updateQuery = "UPDATE Account SET [email]=@newEmail WHERE user=@username";
    command.CommandText = updateQuery;

    SqlParameter dbPramUser = new SqlParameter();
    dbPramUser.ParameterName = "@username";
    dbPramUser.SqlDbType = SqlDbType.VarChar;
    dbPramUser.Value = username;
    command.Parameters.Add(dbPramUser);

    SqlParameter dbPramEmail = new SqlParameter();
    dbPramEmail.ParameterName = "@newEmail";
    dbPramEmail.SqlDbType = SqlDbType.VarChar;
    dbPramEmail.Value = newEmail;
    command.Parameters.Add(dbPramEmail);
    int rowsAffected = 0;
    try
    {
        rowsAffected = command.ExecuteNonQuery();
    }
    finally
    {
    }
    return rowsAffected;
}
```

Figure 5.7: Non-vulnerable Hasan test case example showing the top 4 paths that were given the most attention by the type obfuscated CWE89 model. The thickness of each colored line is related to the attention given to the path (red: 0.060740, yellow: 0.060740, green: 0.036044, blue: 0.033193).



```
public int Bad(SqlConnection conn, string username, string newEmail)
{
    SqlCommand command = conn.CreateCommand();
    string updateQuery = "UPDATE Account SET [email]=" + newEmail + " WHERE user=" + username;
    command.CommandText = updateQuery;

    int rowsAffected = 0;
    try
    {
        rowsAffected = command.ExecuteNonQuery();
    }
    finally
    {
    }
    return rowsAffected;
}
```

Figure 5.8: Hasan test case example which is modified to contain a vulnerability. The top 4 paths are shown that were given the most attention by the type obfuscated CWE89 model. The thickness of each colored line is related to the attention given to the path (red: 0.097439, yellow: 0.063245, green: 0.063245, blue: 0.046448).

6

Discussion

6.1. Discussion of the results

In this section, we discuss the results of our study and simultaneously answer our research questions.

RQ1: How does code2vec perform on a non-obfuscated data set when trained on detecting security vulnerabilities?

The results of our non-obfuscated classifiers show that the performance ranges from an F_1 score of 0.93 to 0.96, depending on the CWE category of the data set used to train the classifiers. These scores are relatively high, however, no real comparison can be made, as we could not find any studies which have used the same sample sources to train classifiers. Considering the code2vec study [15] we achieve higher metric results, however, the task of predicting method names is arguably a more complex task and therefore we cannot compare this with our results. Additionally, Coimbra et al. have used code2vec for detecting C security vulnerabilities and achieved an accuracy of 61.43, precision of 57.50, recall of 61.77, and F_1 score of 59.56. However, Coimbra et al. have used a C vulnerability data set, which contained various CWE vulnerability categories, as opposed to our data sets which contained samples from one or two different CWE categories. Finally, Baptista et al. [33] have used the code2seq [16] model to detect vulnerabilities and achieved an F_1 score of 0.93, precision of 0.90, recall of 0.97, and accuracy of 0.85. However, no specific information is given about the used data set and associated samples.

RQ2: What is the effect of using obfuscated variable names on the code2vec model when trained to detect broken access control or injection vulnerabilities?

Our results show that variable obfuscation does not seem to have a substantial effect on classifiers trained on the task of vulnerability detection.

The metric values of the classifiers trained on the CWE22 data set show that variable obfuscation has no impact on the performance at all. When we evaluated the CWE classifiers on the natural sample data set (extracted from the NVD), instead of using a synthetic sample data set, the classifier performance also showed no differences between non-

obfuscation and obfuscation. Furthermore, we found that the accuracy and precision were almost halved (by 49% and 47% respectively) and the F_1 score was diminished by 32%, compared to the synthetic CWE22 test sets. However, this decline in performance can be partly explained by the small number of cases extracted from the NVD compared to the number of synthetic samples.

We hypothesised that the complexity (cyclomatic complexity and number of paths) of the samples and the number of variables have an impact on the effect of variable obfuscation. We shrunk a copy of the CWE89 data set to the same size as the CWE22 data set and trained new classifiers. We observed that this reduced CWE89 data set resulted in (marginal) performance differences between the classifiers, although the performance of these classifiers was reduced compared to the CWE89 classifiers trained on the non-reduced CWE89 data set. Nonetheless, we conclude that the complexity of the samples does have an impact on the effect of variable obfuscation, as the CWE22 samples were less complex than the CWE89 samples and contained fewer variables compared to the CWE89 samples.

Regarding the CWE89 classifiers and CWE91 classifiers, we found that random- and semi-obfuscated classifiers showed lower performance values, although these differences were marginal. The non-obfuscated and type-obfuscated classifiers scored almost identical, although F_1 scores of the type-obfuscated classifiers were 0.0009 and 0.0005 higher compared to the non-obfuscated classifiers. We have argued in Section 5.3 that the significance of these differences should be statistically substantiated, however, we lack the required computing power to perform the computer-intensive randomisation test needed to prove a significant difference for F_1 [48].

Next, we examined code2vec's attention mechanism for our classifiers. Although examining the attention mechanism gives us an insight into the inner logic of the classifiers, the examination is only performed for a small number of samples and therefore is not representative of all samples. However, the top path-contexts used in the predictions show that there is no significant difference between non-obfuscation and the used obfuscation methods. These outcomes can be an indication of the results of our other experiments, which show us that variable obfuscation does not seem to affect the performance of our classifiers.

RQ3: How does variable obfuscation affect a classifier when trained to detect multiple CWE vulnerabilities instead of one CWE vulnerability?

To answer RQ3, we have trained classifiers using a combined CWE89 and CWE91 data set. The metrics show that the non-obfuscated classifier achieved the highest performance compared to the obfuscated classifiers. Although these differences were marginal. The CWE89 + CWE91 classifiers provided almost identical results to the separately trained CWE89 classifiers and CWE91 classifiers. As we have seen in the examination of the attention mechanism, the attention given to the top path-contexts was similar to the separate CWE89 classifiers and CWE91 classifiers. Therefore, we conclude that variable obfuscation does not have a different impact on classifiers trained on multiple CWE vulnerabilities instead of one CWE vulnerability. However, due to our limited source of vulnerability samples, we were not able to evaluate the impact of variable obfuscation on classifiers using a vulnerability set containing more vulnerability samples. Therefore, it could be interesting for further research to evaluate the influence of variable obfuscation on classifiers trained on a

data set containing samples of more varying CWE categories when one can acquire a more diverse data set.

Additional experiment results

To substantiate the answers to our research questions, we have performed some final experiments. First, we have made a comparison to the work of Compton et al. [19]. Compton et al. have trained classifiers on various tasks [19] and examined the influence of variable obfuscation for each task. We believe the task of detecting security vulnerabilities can best be compared to the bug detection task examined by Compton et al., as one can argue that a bug is represented in a comparable manner as a security vulnerability. In fact, a bug may introduce a security vulnerability.

Compton et al. have only used Cohen's Kappa (κ) to evaluate and compare the performance of their classifiers. Although Compton et al. claim the performance of classifiers is significantly increased by applying variable obfuscation [19], the improvements are marginal. Compton et al. state that overall the random-obfuscation method provides the most significant improvements, in comparison to type-obfuscation. However, as our results show in Table 5.4 (see Section 5.3), obfuscation does not show different results compared to non-obfuscation. Only the CWE91 classifiers seem to slightly benefit from using type-obfuscation according to the κ metric. The difference in results between Compton et al. and our experiments may be due to the problems listed above. Additionally, we have trained our classifiers using data sets of only one CWE (with a maximum of two for our CWE89 + CWE91 classifier), whereas the bug data set used by Compton et al. contains numerous different bug classifications. Finally, the following may also influence the differences:

- Compton et al. have based their results on class vectors, which are the result of aggregation methods performed on the method vectors. We have used method vectors because we believe method vectors may better capture the vulnerabilities. Compton et al. have not provided any findings which indicate whether or not class vectors better represent code snippets in comparison to method vectors.

- The bug data set used by Compton et al. is an imbalanced data set as most cases do not contain a bug, only a minority of the cases do contain a bug [49]. This may lead to high accuracy, as the classifier will classify most samples as 'bug-free', however, in general, the classifier may perform poorly.

- The sources used to create the bug data set used by Compton et al. are not validated by their creators [49] and therefore samples may be labelled incorrectly.

Finally, as explained in Section 5.6, we have cleaned the samples from the Juliet Test Suite from variable names like goodSqlCommand and badSqlCommand. However, to allow future studies to better compare their results with our results, we have trained CWE89 classifiers with a non-cleaned data set. Because the Juliet Test Suite provided only 729 of the 19872 samples for our CWE89 classifiers, we did not expect any differences in the results. Nevertheless, the results confirm that the non-cleaned data set provided classifiers with similar performance as the classifiers trained on the cleaned data set.

Additionally, one might wonder what the effect of variable obfuscation could have on vulnerability detection when the classifiers are trained on vulnerability samples extracted from large sources containing natural production source code. In this study, we could only

obtain and use synthetic vulnerability samples which were created by generators using a relatively small amount of templates. Therefore, the samples did not show remarkable differences in code complexity (cyclomatic complexity and number of paths), but also variable naming. Thus, our non-obfuscated samples are arguably already obfuscated, due to its synthetic nature. However, natural production source code could have variable names which indicate whether or not the programmer has thought carefully about the code implementation. For example, a programmer who is aware of buffer-overflow vulnerabilities could have named his buffer variables something like "boundedDataBuffer", indicating the buffer variable is not able to overflow. Additionally, vague variable names like "x", "foo", "bar", "someData" could indicate a less experienced programmer who does not (yet) comprehend how to program non-vulnerable code [50]. When we apply an obfuscation method to these variable names, this information is lost and therefore cannot be used by models trained on the task of vulnerability detection. However, we could categorise this potentially large data set with natural production vulnerability cases into two categories. The first category contains samples which contain badly named variable names and the last category contains samples with variable names which are carefully chosen. We hypothesise the first category could benefit from a variable obfuscation method such as type obfuscation because this would produce more describing variable names than vague or meaningless names. In contrast, variable obfuscation methods performed on the samples of the last category could have a negative impact on classifiers trained on the task of vulnerability detection, when trained using these obfuscated vulnerability samples. Nevertheless, how this manifests itself in practice should be validated by future research.

6.2. LIMITATIONS

In this section, we describe some limitations of our research.

6.2.1. MODEL INPUT

Because the code2vec model only accepts individual methods as input, vulnerabilities that span multiple parts in a program cannot easily be found using our approach. Some examples are vulnerabilities which span more than one method or vulnerabilities which are spread over multiple classes. Other studies have tried to mitigate this problem by using a technique called 'program slicing', such as the recent VulDeePecker [46], SySeVR [51] and GLICE [47] studies. However, this technique is out of scope for our project.

6.2.2. PROGRAMMING LANGUAGE

All samples used in this study are written in C#. Therefore, the results of this study could be different when the study is repeated using samples that are written in another programming language. However, the concept of variables is the same in every programming language. Consequently, we do not believe the used programming language has an impact on the results. However, this can only be proven by experiments.

6.2.3. VULNERABILITY TEST CASE SOURCES

The C# vulnerability samples we could use in our experiments were limited to the publicly available vulnerability test cases. Our results show that the data size does impact the results of variable obfuscation and therefore it would be beneficial to have larger data sets

available. Additionally, most CWEs are not present in the available vulnerability test case sources or even not available at all. Therefore, we could only use a small number of CWEs in our experiments. Other CWEs could potentially give different results which we could not investigate because of the limited vulnerability cases.

7

CONCLUSION AND RECOMMENDATIONS

7.1. CONCLUSION

Overall, our results show that variable obfuscation does not seem to have a substantial effect on classifiers trained on the task of vulnerability detection. This is because the resulting F_1 scores of the classifiers do not show significant differences. However, to be able to statistically substantiate this result, further research should perform a compute-intensive randomisation test [48], for which we lack computing power.

Nonetheless, we showed that variable obfuscation has none to minimal impact on the classifier performance, which we further substantiated by some additional experiments. First, we evaluated the performance of our CWE22 classifiers using the natural CWE22 C# vulnerability cases extracted from the NVD. The results of this experiment showed us that, based on our data set of natural samples, the performance of a non-obfuscated classifier was the same as the obfuscated classifiers.

Next, we evaluated whether or not the results of our previous experiment were caused by the complexity of our CWE22 samples, by training on a CWE89 data set for which the number of samples was reduced to the same size as our CWE22 data set. The results of this experiment depict that the reduced CWE89 data set does result in (marginal) performance differences between classifiers, although the performance of these classifiers was reduced compared to the CWE89 classifiers trained on the non-reduced data set. We conclude this is because our CWE22 samples are less complex and contain fewer variables, compared to the more complex CWE89 samples which contain more variable names.

Finally, we have examined the effect of variable obfuscation on code2vec's attention mechanism. Even though we have used a small number of samples to get an insight into the attention mechanism, we concluded that both the non-obfuscated and obfuscated classifiers had put the same attention on the same category of path-contexts to make a prediction. This further substantiates our conclusion that variable obfuscation does not have an impact on the classifier performance. If there is any difference, we noticed that variable obfuscation has the least impact on classifiers which are trained using samples which contain a small number of variables.

To end our conclusion, we give answer to our main question: **What influence does variable obfuscation have on the code2vec model when trained on the task of detecting software vulnerabilities?**

Although the study performed by Compton et al. shows that variable obfuscation could increase the performance of classifiers marginally, yet statistically significant, our results show that variable obfuscation does not have a significant influence on the detection of vulnerabilities.

7.2. Recommendations for future work

As described in Chapter 6, to statistically substantiate our results of variable obfuscation not improving the ability of classifiers to detect vulnerabilities, we should apply a compute-intensive randomisation test [48]. However, we were not able to perform this task during this study due to the lack of computing power and time. Therefore, further research could be performed to statistically substantiate our results.

In Section 2.6, we opted for not using class embeddings as described by Compton et al. [19], because this could result in information loss. However, during our research, we had to omit hundreds of test cases because they described vulnerabilities ranging over multiple classes or files (as stated in Section 4.1.1). Therefore, experimenting with various aggregation methods to create embeddings for code snippets which span various code parts, instead of only single methods, could be interesting for further research. These experiments do not only have to be aimed at detecting vulnerabilities in source code specifically but could also be beneficial for various other tasks, such as detecting (faultily implemented) design patterns in software or bug prediction.

An additional untested approach would be to modify the semi-obfuscation method to not use type obfuscation, but random obfuscation. Even a combined type and random obfuscation method is possible, to allow the classifier to rely both more on the variable structures and code structure respectively. Additionally, the percentage of variable obfuscation could be tweaked, where we only obfuscated 50% of the variables in our semi-obfuscation method. Due to the limited time of our research, we have only used one semi-obfuscation method approach in our experiments. However, because variable obfuscation does not seem to improve the performance of our classifiers as seen in our experiment results, we do not believe a modified semi-obfuscation method would provide groundbreaking differences in the results. Yet, as an additional approach, it would be interesting to examine whether or not completely different obfuscation methods would show different results. For example, by renaming variables names for variables which could potentially contain tainted data which is not sanitised. This way the model could rely on this additional information.

Furthermore, variable obfuscation could be applied to other classifier algorithms than code2vec in order to be able to further acknowledge our results that variable obfuscation does not improve the performance of classifiers trained on the task of vulnerability detection. For example, the VulDeePecker [46] or code2seq [16] models could be used. Additionally, although code2vec (and code2seq) do not require designing manual features [15; 16], variable obfuscation might have a bigger impact when the features are manually designed. The reason for this could be that a model has to rely more on these features as opposed to the more generic path-contexts generated for the code2vec (or code2seq) model. However, this has yet to be examined.

To continue on the idea of using different algorithms, we want to propose the idea of adding features related to security vulnerabilities to the code2vec vectors. Code2vec is able to generalise well [15] and together with additional information related to vulnerabilities,

the model might be able to better predict vulnerabilities within source code. This can even be performed for the code2seq model [16], however, this model uses more resources and therefore takes more time to train.

Finally, the test case sources we have used have been used in studies in the past, however, they could still contain errors as we have seen in the Hasan test cases (see Section 4.3.3). Therefore, to confirm the validity of our results the test case sources could be analyzed to detect if any errors (such as wrongly labelled cases) are present. However, we believe the majority of the test cases are correct and therefore some errors would not significantly have impacted our results.

A

DATA SET SAMPLE OVERVIEW

Table A.1: Overview of the OWASP Injection category CWEs that are present in our research and the number of extracted samples for each CWE (per data set).

CWE	Juliet Test Suite	Test Suite 105	Hasan	CVEfixes / NVD	**Total samples**
CWE 20 - Improper Input Validation	0	0	0	26	26
CWE 74 - Injection	0	0	0	2	2
CWE 78 - OS Command Injection	170	1740	3	0	1913
CWE 79 - Cross-site Scripting	0	0	0	14	14
CWE 80 - Basic XSS	306	0	0	0	306
CWE 83 - Improper Neutralization of Script in Attributes in a Web Page	153	0	0	0	153
CWE 89 - SQL Injection	729	19140	3	0	19872
CWE 90 - LDAP Injection	170	1740	0	0	1910
CWE 91 - XML Injection	0	5220	0	0	5220
CWE 94 - Code Injection	270	0	0	1	271
CWE 113 - HTTP Response Splitting	729	0	0	0	729
CWE 470 - Unsafe Reflection	153	0	0	0	153
CWE 643 - XPath Injection	270	0	0	0	270

Table A.2: Overview of the OWASP Broken Access Control category CWEs that are present in our research and the number of extracted samples for each CWE (per data set).

CWE	Juliet Test Suite	Test Suite 105	Hasan	CVEfixes / NVD	**Total samples**
CWE 22 - Path Traversal	0	2088	0	48	2136
CWE 23 - Relative Path Traversal	170	0	0	0	170
CWE-284 Improper Access Control	30	0	0	0	30
CWE-352 Cross-Site Request Forgery (CSRF)	0	0	0	0	0
CWE-566 Authorization Bypass Through User-Controlled SQL Primary Key	17	0	0	0	17
CWE-601 URL Redirection to Untrusted Site ('Open Redirect')	153	0	0	0	153
CWE-706 Use of Incorrectly-Resolved Name or Reference	0	0	0	0	0

B

CVEfixes C# CVE record results

As explained in Section 4.3.4, we encountered different results when running the query of Listing 4.3 on the Bhandari et al. database copy and our own database copy. The results are shown in Figure B.1. Compared to the Bhandari et al. database copy which contained 34 unique C# CVE records, our copy contained 50 unique C# CVE records. The difference of 15 new CVE records in our copy is due to the Bhandari et al. copy only containing records up to 9 June 2020. However, our database copy missed two C# CVE records and contained three additional CVE records, compared to the Bhandari et al. copy. Together, they provided 52 unique C# CVE records. An observant reader would note that CVE-2020-8416 describes a CWE-770 according to our database, but the Bhandari et al. database copy tells us different (CWE-400). However, the assigned CWE for this CVE has changed on 21 Juli 2021 in the NVD and therefore explains the difference between the two database copies [94].

Running the query of Listing 4.4 provided 49 out of 52 unique records of Listing 4.3, but also additional records have been found this way. Therefore, the query of Listing 4.4 provided the best results and has been used to create samples for our data set. The results of the query in Listing 4.4 are depicted in Figure B.2. This figure shows all results of the query in Listing 4.4, including the three CVE records which describe JavaScript or C fixes instead of C# fixes. These records have been struck through in Figure B.2. The interesting CVE records describing Broken Access Control and Injection category CWEs are in bold in Figure B.2.

	Our CVEfixes database copy		Bhandari et al. database copy	
Count	CVEID	CWE	CVEID	CWE
1	CVE-2011-0991	CWE-399	CVE-2011-0991	CWE-399
2			CVE-2012-3382	CWE-79
3	CVE-2013-6795	CWE-94	CVE-2013-6795	CWE-94
4	CVE-2014-4172	CWE-74	CVE-2014-4172	CWE-74
5	CVE-2015-2318	CWE-295	CVE-2015-2318	CWE-295
6	CVE-2015-2319	CWE-295	CVE-2015-2319	CWE-295
7	CVE-2015-2320	CWE-295	CVE-2015-2320	CWE-295
8	CVE-2015-8813	CWE-918	CVE-2015-8813	CWE-918
9	CVE-2015-8814	CWE-352	CVE-2015-8814	CWE-352
10	CVE-2017-0907	CWE-918	CVE-2017-0907	CWE-918
11	CVE-2017-0929	CWE-918	CVE-2017-0929	CWE-918
12	CVE-2017-1000457	CWE-79	CVE-2017-1000457	CWE-79
13	CVE-2017-15279	CWE-79	CVE-2017-15279	CWE-79
14			CVE-2017-15280	CWE-611
15	CVE-2017-16806	CWE-22	CVE-2017-16806	CWE-22
16	CVE-2018-7559	CWE-320	CVE-2018-7559	CWE-320
17	CVE-2018-8899	CWE-79	CVE-2018-8899	CWE-79
18	CVE-2019-1010199	CWE-79	CVE-2019-1010199	CWE-79
19	CVE-2019-12277	CWE-22	CVE-2019-12277	CWE-22
20	CVE-2019-18641	Unknown	CVE-2019-18641	Unknown
21	CVE-2019-19249	CWE-20	CVE-2019-19249	CWE-20
22	CVE-2019-20627	CWE-611	CVE-2019-20627	CWE-611
23	CVE-2020-26207	CWE-502	CVE-2020-26207	CWE-502
24	CVE-2020-26293	CWE-79	CVE-2020-26293	CWE-79
25	CVE-2020-27996	Unknown	CVE-2020-27996	Unknown
26	CVE-2020-28042	CWE-347	CVE-2020-28042	CWE-347
27	CVE-2020-36364	CWE-22	CVE-2020-36364	CWE-22
28	CVE-2020-5261	CWE-294	CVE-2020-5261	CWE-294
29	CVE-2020-5268	CWE-287	CVE-2020-5268	CWE-287
30	CVE-2020-7791	Unknown	CVE-2020-7791	Unknown
31	CVE-2020-8416	CWE-770	CVE-2020-8416	CWE-400
32	CVE-2020-8815	CWE-20	CVE-2020-8815	CWE-20
33	CVE-2021-21402	CWE-22	CVE-2021-21402	CWE-22
34	CVE-2021-23407	CWE-22		
35	CVE-2021-23415	CWE-22		
36	CVE-2021-23758	CWE-502		
37	CVE-2021-32054	CWE-706	CVE-2021-32054	CWE-706
38	CVE-2021-32795	CWE-20		
39	CVE-2021-32840	CWE-22		
40	CVE-2021-32841	CWE-22		
41	CVE-2021-33318	CWE-20		
42	CVE-2021-3646	CWE-79		
43	CVE-2021-3830	CWE-79		
44	CVE-2022-0159	CWE-79		
45	CVE-2022-0243	CWE-79		
46	CVE-2022-0274	CWE-79		
47	CVE-2022-0820	CWE-79		
48	CVE-2022-0821	CWE-863		
49	CVE-2022-0822	CWE-79		
50	CVE-2022-23627	CWE-863		
51	CVE-2022-28451	CWE-22		
52	CVE-2022-29245	CWE-338		

Figure B.1: The resulting C# CVE records of query 1 from Listing 4.3. Comparison between our generated database copy versus the database copy provided by Bhandari et al.

Count	cve_id	cwe_id	repo_url	hash/branch
1	CVE-2010-4159	Unknown	https://github.com/mono/mono	8e890a3bf80a4620e417814dc14886b1bbd17625
~~2~~	~~CVE-2010-4254~~	~~**CWE-20**~~	~~https://github.com/mono/mono~~	~~4905ef1130feb26c3150b28b97e4a96752e0d399~~
3	CVE-2011-0989	CWE-264	https://github.com/mono/mono	035c8587c0d8d307e45f1b7171a0d337bb451f1e
4	CVE-2011-0990	CWE-362	https://github.com/mono/mono	2f00e4bbb2137130845afb1b2a1e678552fc8e5c
5	CVE-2011-0991	CWE-399	https://github.com/mono/mono	3f8ee42b8c867d9a4c18c22657840d072cca5c3a
6	CVE-2011-0992	CWE-399	https://github.com/mono/mono	722f9890f09aadfc37ae479e7d946d5fc5ef7b91
7	CVE-2012-3382	**CWE-79**	https://github.com/mono/mono	d16d4623edb210635bec3ca3786481b82cde25a2
8	CVE-2013-6795	**CWE-94**	https://github.com/rackerlabs/openstack-guest-agents-windows-xenserver	ef16f88f20254b8083e361f11707da25f8482401
9	CVE-2014-10074	CWE-434	https://github.com/Umbraco/Umbraco-CMS	cad06502235acabf7fb7dca779d2f78f08547e39
10	CVE-2014-4172	**CWE-74**	https://github.com/Jasig/dotnet-cas-client	f0e030014fb7a39e5f38469f43199dc590fd0e8d
11	CVE-2015-2318	CWE-295	https://github.com/mono/mono	1509226c41d74194c146deb173e752b8d3cdeec4
12	CVE-2015-2319	CWE-295	https://github.com/mono/mono	9c38772f094168d8bfd5bc73bf8925cd04faad10
13	CVE-2015-2320	CWE-295	https://github.com/mono/mono	b371da6b2d68b4cdd0f21d6342af6c42794f998b
14	CVE-2015-8813	CWE-918	https://github.com/umbraco/Umbraco-CMS	924a016ffe7ae7ea6d516c07a7852f0095eddbce
15	CVE-2015-8814	**CWE-352**	https://github.com/umbraco/Umbraco-CMS	18c3345e47663a358a042652e697b988d6a380eb
16	CVE-2017-0907	CWE-918	https://github.com/recurly/recurly-client-net	9eef460c0084afd5c24d66220c8b7a381cf9a1f1
17	CVE-2017-0929	CWE-918	https://github.com/dnnsoftware/Dnn.Platform	d3953db85fee77bb5e6383747692c507ef8b94c3
18	CVE-2017-1000457	**CWE-79**	https://github.com/i7MEDIA/mojoportal	5ea8129f74c80cbf1f68b9083c745cc8a685485d
19	CVE-2017-15279	**CWE-79**	https://github.com/umbraco/Umbraco-CMS	fe2b86b681455ac975b294652064b2718d4e2ba2
20	CVE-2017-15280	CWE-611	https://github.com/umbraco/Umbraco-CMS	5dde2efe0d2b3a47d17439e03acabb7ea2befb64
21	CVE-2017-16806	**CWE-22**	https://github.com/Ulterius/server	770d1821de43cf1d0a93c79025995bdd812a76ee
22	CVE-2018-1002205	**CWE-22**	https://github.com/haf/DotNetZip.Semverd	55d2c13c0cc64654e18fcdd0038fdb3d7458e366
23	CVE-2018-1002206	**CWE-22**	https://github.com/adamhathcock/sharpcompress	42b1205fb435de523e6ef8ac5b7bafbe712997f6
24	CVE-2018-14485	CWE-611	https://github.com/rxtur/BlogEngine.NET	master
25	CVE-2018-7559	CWE-320	https://github.com/OPCFoundation/UA-.NETStandard	ebcf026a54dd0c9052cff009d96d827ac923d150
26	CVE-2018-8899	**CWE-79**	https://github.com/IdentityServer/IdentityServer4	21d0da227f50ac102de469a13bc5a15d2cc0f895
27	CVE-2019-1010199	**CWE-79**	https://github.com/ServiceStack/ServiceStack	a0e0d7de20f5d1712f1793f925496def4383c610
28	CVE-2019-10717	**CWE-22**	https://github.com/rxtur/BlogEngine.NET	master
29	CVE-2019-10721	**CWE-601**	https://github.com/rxtur/BlogEngine.NET	master
30	CVE-2019-12277	**CWE-22**	https://github.com/blogifierdotnet/Blogifier.git	3e2ae11f6be8aab82128f223c2916fab5a408be5
31	CVE-2019-18211	CWE-502	https://github.com/Orckestra/C1-CMS-Foundation	dev
32	CVE-2019-18641	Unknown	https://github.com/SparkDevNetwork/Rock	576f5ec22b1c43f123a377612981c68538167c61
33	CVE-2019-19249	**CWE-20**	https://github.com/d4software/QueryTree.git	57b700823f8eb1a42eb3bc0c706fbe5e5f5e766f
34	CVE-2019-19392	CWE-276	https://github.com/fordnn/usersexportimport	master
35	CVE-2019-20627	CWE-611	https://github.com/ravibpatel/AutoUpdater.NET	1dc25f2bea6ea522dbac1512b5563c4746d539c3
36	CVE-2020-26207	CWE-502	https://github.com/martinjw/dbschemareader	4c0ab7b1fd8c4e3140f9fd54d303f107a9c8d994
37	CVE-2020-26233	**CWE-706**	https://github.com/microsoft/Git-Credential-Manager-Core	61c0388e064babb3b4e60d3ec269e8a07ab3bc76
38	CVE-2020-26293	**CWE-79**	https://github.com/mganss/HtmlSanitizer	a3a7602a44d4155d51ec0fbbedc2a49e9c7e2eb8
39	CVE-2020-27996	Unknown	https://github.com/smartstore/SmartStoreNET	8702c6140f4fc91956ef35dba12d24492fb3f768
40	CVE-2020-28042	CWE-347	https://github.com/ServiceStack/ServiceStack	540d4060e877a03ae95343c1a8560a26768585ee
41	CVE-2020-36364	**CWE-22**	https://github.com/smartstore/SmartStoreNET	5ab1e37dc8d6415d04354e1a116f3d82e9555f5c
42	CVE-2020-5234	CWE-787	https://github.com/neuecc/MessagePack-CSharp	56fa86219d01d0a183babbbbcb34abbdea588a02
43	CVE-2020-5261	CWE-294	https://github.com/Sustainsys/Saml2	e58e0a1aff2b1ead6aca080b7cdced55ee6d5241
44	CVE-2020-5268	CWE-287	https://github.com/Sustainsys/Saml2	e58e0a1aff2b1ead6aca080b7cdced55ee6d5241
45	CVE-2020-7791	Unknown	https://github.com/turquoiseowl/i18n	c418e3345313dc896c1951d8c46ab0b9b12fcbd3
46	CVE-2020-8416	CWE-770	https://github.com/kolya5544/BearFTP	9965337f9d4c0325e4aab324dcd485e4cbb7b428
47	CVE-2020-8815	**CWE-20**	https://github.com/kolya5544/BearFTP	17a6ead72d4a25cbfcef5e27613aa0a5f88a4b26
48	CVE-2021-21402	**CWE-22**	https://github.com/jellyfin/jellyfin	0183ef8e89195f420c48d2600bc0b72f6d3a7fd7
49	CVE-2021-23407	**CWE-22**	https://github.com/trannamtrung1st/elFinder.Net.Core	5498c8a86b76ef089cfbd7ef8be014b61fa11c73
50	CVE-2021-23415	**CWE-22**	https://github.com/mguinness/elFinder.AspNet.git	675049b39284a9e84f0915c71d688da8ebc7d720
51	CVE-2021-23758	CWE-502	https://github.com/michaelschwarz/Ajax.NET-Professional	b0e63be5f0bb20dfce507cb8a1a9568f6e73de57
52	CVE-2021-25976	**CWE-352**	https://github.com/PiranhaCMS/piranha.core	e42abacdd0dd880ce9cf6607efcc24646ac82eda
~~53~~	~~CVE-2021-25977~~	~~**CWE-79**~~	~~https://github.com/PiranhaCMS/piranha.core~~	~~543bc53c7dbd28e793ec960b57fb0c716c6b18d7~~
54	CVE-2021-32054	**CWE-706**	https://github.com/FirelyTeam/spark	9c79320059f92d8aa4fbd6cc4fa8f9d5d6ba9941
55	CVE-2021-32607	Unknown	https://github.com/smartstore/SmartStoreNET	5b4e60ae7124df0898975cb8f994f9f23db1fae3
56	CVE-2021-32608	Unknown	https://github.com/smartstore/SmartStoreNET	ae03d45e23734555a2aef0b0c3d33c21e076c20f
57	CVE-2021-32795	**CWE-20**	https://github.com/JustArchiNET/ArchiSteamFarm	4cd581ec041912cf199c5512fe6d1dcaec0594c0
58	CVE-2021-32840	**CWE-22**	https://github.com/icsharpcode/SharpZipLib	a0e96de70b5264f4c919b09253b1522bc7a221cc
59	CVE-2021-32841	**CWE-22**	https://github.com/icsharpcode/SharpZipLib	5c3b293de5d65b108e7f2cd0ea8f81c1b8273f78
60	CVE-2021-33318	**CWE-20**	https://github.com/jchristn/IpMatcher	81d77c2f33aa912dbd032b34b9e184fc6e041d89
61	CVE-2021-3646	**CWE-79**	https://github.com/btcpayserver/btcpayserver	fc4e47cec608cc3dba24b19d0145ac69320b975e
62	CVE-2021-3830	**CWE-79**	https://github.com/btcpayserver/btcpayserver	fc4e47cec608cc3dba24b19d0145ac69320b975e
~~63~~	~~CVE-2021-43853~~	~~**CWE-79**~~	~~https://github.com/michaelschwarz/Ajax.NET-Professional~~	~~e89e39b9679feb8ab6644fe21ee7e652eb615e2b~~
64	CVE-2022-0159	**CWE-79**	https://github.com/orchardcms/orchardcore	4da927d39a49138527c30db09c962ff706f95202
65	CVE-2022-0243	**CWE-79**	https://github.com/orchardcms/orchardcore	218f25ddfadb66a54de7a82dffe3ab2e4ab7c4b4
66	CVE-2022-0274	**CWE-79**	https://github.com/orchardcms/orchardcore	218f25ddfadb66a54de7a82dffe3ab2e4ab7c4b4
67	CVE-2022-0820	**CWE-79**	https://github.com/orchardcms/orchardcore	b7096af1028d8f909f63dd076d1bbd573913a92d
68	CVE-2022-0821	CWE-863	https://github.com/orchardcms/orchardcore	b7096af1028d8f909f63dd076d1bbd573913a92d
69	CVE-2022-0822	**CWE-79**	https://github.com/orchardcms/orchardcore	b7096af1028d8f909f63dd076d1bbd573913a92d
70	CVE-2022-23627	CWE-863	https://github.com/JustArchiNET/ArchiSteamFarm	7a29d9282bdc3280db2a379c24f73916d786f9b4
71	CVE-2022-24774	**CWE-20 CWE-22 en CWE-35**	https://github.com/CycloneDX/cyclonedx-bom-repo-server	001a3278b5572e52c0ecac0bd1157bf2599502b7
72	CVE-2022-28451	**CWE-22**	https://github.com/nopSolutions/nopCommerce	47ff9a241243db9359f10216bcf401baaa36d0b4
73	CVE-2022-29245	CWE-338	https://github.com/sshnet/SSH.NET	03c6d60736b8f7b42e44d6989a53f9b644a091fb

Figure B.2: The resulting C# CVE records of query 2 from Listing 4.4 which have been used to create samples for our data set.

C

INTERPRETING ATTENTION SAMPLES

This appendix contains all sample code sources used during the interpretation of code2vec's attention mechanism, as described in Section 5.7.

Listing C.1: Good (non-vulnerable) CWE89 sample used to interpret code2vec's attention mechanism. Sample is taken from the Hasan test cases.

```
1 public int Good(SqlConnection conn, string username, string newEmail
      )
2 {
3   SqlCommand command = conn.CreateCommand();
4   string updateQuery = "UPDATE Account SET [email]=@newEmail WHERE
       user=@username";
5   command.CommandText = updateQuery;
6
7   SqlParameter dbPramUser = new SqlParameter();
8   dbPramUser.ParameterName = "@username";
9   dbPramUser.SqlDbType = SqlDbType.VarChar;
10  dbPramUser.Value = username;
11  command.Parameters.Add(dbPramUser);
12
13  SqlParameter dbPramEmail = new SqlParameter();
14  dbPramEmail.ParameterName = "@newEmail";
15  dbPramEmail.SqlDbType = SqlDbType.VarChar;
16  dbPramEmail.Value = newEmail;
17  command.Parameters.Add(dbPramEmail);
18  int rowsAffected = 0;
19  try
20  {
21    rowsAffected = command.ExecuteNonQuery();
22  }
23  finally
24  {
25  }
26  return rowsAffected;
27 }
```

Listing C.2: Bad (vulnerable) CWE89 sample used to interpret code2vec's attention mechanism. Sample is a modified version of the good sample taken from the Hasan test cases.

```
public int Bad(SqlConnection conn, string username, string newEmail)
{
    SqlCommand command = conn.CreateCommand();
    string updateQuery = "UPDATE Account SET [email]=" + newEmail +
        " WHERE user=" + username;
    command.CommandText = updateQuery;

    int rowsAffected = 0;
    try
    {
        rowsAffected = command.ExecuteNonQuery();
    }
    finally
    {
    }
    return rowsAffected;
}
```

Listing C.3: Good (non-vulnerable) CWE22 sample used to interpret code2vec's attention mechanism. Sample is taken from CVE-2021-32841.

```
private void Good(string destDir, TarEntry entry, bool
    allowParentTraversal)
{
  OnProgressMessageEvent(entry, null);
  string name = entry.Name;
  if (Path.IsPathRooted(name))
  {
    // NOTE:
    // for UNC names...  \\machine\share\zoom\beet.txt gives \zoom\
       beet.txt
    name = name.Substring(Path.GetPathRoot(name).Length);
  }
  name = name.Replace('/', Path.DirectorySeparatorChar);

  string destFile = Path.Combine(destDir, name);
  var destFileDir = Path.GetDirectoryName(Path.GetFullPath(destFile)
      ) ?? "";

  if (!allowParentTraversal && !destFileDir.StartsWith(destDir,
      StringComparison.InvariantCultureIgnoreCase))
  {
    throw new InvalidNameException("Parent traversal in paths is not
        allowed");
  }
  if (entry.IsDirectory)
  {
    EnsureDirectoryExists(destFile);
  }
  else
  {
    string parentDirectory = Path.GetDirectoryName(destFile);
    EnsureDirectoryExists(parentDirectory);
    bool process = true;
    var fileInfo = new FileInfo(destFile);
    if (fileInfo.Exists)
    {
      if (keepOldFiles)
      {
        OnProgressMessageEvent(entry, "Destination file already
            exists");
        process = false;
      }
      else if ((fileInfo.Attributes & FileAttributes.ReadOnly) != 0)
      {
        OnProgressMessageEvent(entry, "Destination file already
            exists, and is read-only");
        process = false;
      }
    }
    if (process)
    {
      using (var outputStream = File.Create(destFile))
      {
```

```
47          if (this.asciiTranslate)
48          {
49            // May need to translate the file.
50            ExtractAndTranslateEntry(destFile, outputStream);
51          }
52          else
53          {
54            // If translation is disabled, just copy the entry across
                directly.
55            tarIn.CopyEntryContents(outputStream);
56          }
57        }
58      }
59    }
60  }
```

Listing C.4: Bad (vulnerable) CWE22 sample used to interpret code2vec's attention mechanism. Sample is taken from CVE-2021-32841.

```
private void Bad(string destDir, TarEntry entry, bool
    allowParentTraversal)
{
  OnProgressMessageEvent(entry, null);
  string name = entry.Name;
  if (Path.IsPathRooted(name))
  {
    // NOTE:
    // for UNC names...  \\machine\share\zoom\beet.txt gives \zoom\
       beet.txt
    name = name.Substring(Path.GetPathRoot(name).Length);
  }
  name = name.Replace('/', Path.DirectorySeparatorChar);

  string destFile = Path.Combine(destDir, name);

  if (!allowParentTraversal && !Path.GetFullPath(destFile).
      StartsWith(destDir, StringComparison.InvariantCultureIgnoreCase
      ))
  {
    throw new InvalidNameException("Parent traversal in paths is not
        allowed");
  }
  if (entry.IsDirectory)
  {
    EnsureDirectoryExists(destFile);
  }
  else
  {
    string parentDirectory = Path.GetDirectoryName(destFile);
    EnsureDirectoryExists(parentDirectory);
    bool process = true;
    var fileInfo = new FileInfo(destFile);
    if (fileInfo.Exists)
    {
      if (keepOldFiles)
      {
        OnProgressMessageEvent(entry, "Destination file already
            exists");
        process = false;
      }
      else if ((fileInfo.Attributes & FileAttributes.ReadOnly) != 0)
      {
        OnProgressMessageEvent(entry, "Destination file already
            exists, and is read-only");
        process = false;
      }
    }
    if (process)
    {
      using (var outputStream = File.Create(destFile))
      {
        if (this.asciiTranslate)
```

```
47            {
48              // May need to translate the file.
49              ExtractAndTranslateEntry(destFile, outputStream);
50            }
51            else
52            {
53              // If translation is disabled, just copy the entry across
                    directly.
54              tarIn.CopyEntryContents(outputStream);
55            }
56          }
57        }
58      }
59  }
```

Listing C.5: Good (non-vulnerable) CWE91 sample used to interpret code2vec's attention mechanism. Sample is taken from Test Suite 105.

```
public static void Good(string[] args)
{
    string tainted_2 = null;
    string tainted_3 = null;
    tainted_2 = args[1];
    tainted_3 = tainted_2;
    if ((4 + 2 <= 42))
    {
        string pattern = @"/^[0-9]*$/";
        Regex r = new Regex(pattern);
        Match m = r.Match(tainted_2);
        if (!m.Success)
        {
            tainted_3 = "";
        }
        else
        {
            tainted_3 = tainted_2;
        }
    }

    string query = tainted_3;
    string filename = "file.xml";
    XDocument document = XDocument.Load(filename);
    XmlTextWriter writer = new XmlTextWriter(Console.Out);
    writer.Formatting = Formatting.Indented;
    var node = document.Root.Elements("foo").Where(x => (string)x.
       Element("bar") == query).SingleOrDefault();
    node.WriteTo(writer);
    writer.Close();
}
```

Listing C.6: Bad (vulnerable) CWE91 sample used to interpret code2vec's attention mechanism. Sample is taken from Test Suite 105.

```
public static void Bad(string[] args)
{
    string tainted_2 = null;
    string tainted_3 = null;
    tainted_2 = args[1];
    tainted_3 = tainted_2;
    if ((Math.Sqrt(42) <= 42))
    {
        string pattern = @"/^[0-9]*$/";
        Regex r = new Regex(pattern);
        Match m = r.Match(tainted_2);
        if (!m.Success)
        {
            tainted_3 = "";
        }
        else
        {
            tainted_3 = tainted_2;
```

```
19          }
20      }
21
22      //flaw
23      string query = "//user[@name='" + tainted_3 + "']";
24      string filename = "file.xml";
25      XmlDocument document = new XmlDocument();
26      document.Load(filename);
27      XmlTextWriter writer = new XmlTextWriter(Console.Out);
28      writer.Formatting = Formatting.Indented;
29      XmlNode node = document.SelectSingleNode(query);
30      node.WriteTo(writer);
31      writer.Close();
32  }
```

BIBLIOGRAPHY

[1] H. Stevenson and K. Alharbi, "Software Security," 2012. [Online]. Available: https://home.cs.colorado.edu/~kena/classes/5828/s12/presentation-materials/stevensonhunteralharbikhali.pdf 3

[2] Federal Bureau of Investigation, "2020 Internet Crime Report," 2020. [Online]. Available: https://www.ic3.gov/Media/PDF/AnnualReport/2020_IC3Report.pdf 3

[3] A. Edmundson, B. Holtkamp, E. Rivera, M. Finifter, A. Mettler, and D. Wagner, "An Empirical Study on the Effectiveness of Security Code Review," in *Engineering Secure Software and Systems*, ser. Lecture Notes in Computer Science, J. Jürjens, B. Livshits, and R. Scandariato, Eds. Berlin, Heidelberg: Springer, 2013, pp. 197–212. 3

[4] A. Meneely, A. C. R. Tejeda, B. Spates, S. Trudeau, D. Neuberger, K. Whitlock, C. Ketant, and K. Davis, "An empirical investigation of socio-technical code review metrics and security vulnerabilities," in *Proceedings of the 6th International Workshop on Social Software Engineering*, ser. SSE 2014. New York, NY, USA: Association for Computing Machinery, Nov. 2014, pp. 37–44. [Online]. Available: http://doi.org/10.1145/2661685.2661687 4

[5] H. G. Rice, "Classes of recursively enumerable sets and their decision problems," *Transactions of the American Mathematical Society*, vol. 74, no. 2, pp. 358–366, 1953. [Online]. Available: https://www.ams.org/tran/1953-074-02/S0002-9947-1953-0053041-6/ 4

[6] S. M. Ghaffarian and H. R. Shahriari, "Software Vulnerability Analysis and Discovery Using Machine-Learning and Data-Mining Techniques: A Survey," *ACM Computing Surveys*, vol. 50, no. 4, pp. 56:1–56:36, Aug. 2017. [Online]. Available: http://doi.org/10.1145/3092566 4

[7] Y. Shin and L. Williams, "Can traditional fault prediction models be used for vulnerability prediction?" *Empirical Software Engineering*, vol. 18, no. 1, pp. 25–59, Feb. 2013. [Online]. Available: https://doi.org/10.1007/s10664-011-9190-8 4

[8] G. Díaz and J. R. Bermejo, "Static analysis of source code security: Assessment of tools against SAMATE tests," *Information and Software Technology*, vol. 55, no. 8, pp. 1462–1476, Aug. 2013. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584913000384 4

[9] P. Zeng, G. Lin, L. Pan, Y. Tai, and J. Zhang, "Software Vulnerability Analysis and Discovery Using Deep Learning Techniques: A Survey," *IEEE Access*, vol. 8, pp. 197 158–197 172, Jan. 2020, publisher: IEEE. [Online]. Available: https://doaj.org/article/8c9bada603444693830b5ae8317a4d64 4

[10] H. Hanif, M. H. N. Md Nasir, M. F. Ab Razak, A. Firdaus, and N. B. Anuar, "The rise of software vulnerability: Taxonomy of software vulnerabilities detection and machine learning approaches," *Journal of Network and Computer Applications*, vol. 179, p. 103009, Apr. 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1084804521000369 4

[11] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed Representations of Words and Phrases and their Compositionality," *arXiv:1310.4546 [cs, stat]*, Oct. 2013, arXiv: 1310.4546. [Online]. Available: http://arxiv.org/abs/1310.4546 4, 9

[12] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient Estimation of Word Representations in Vector Space," *arXiv:1301.3781 [cs]*, Sep. 2013, arXiv: 1301.3781. [Online]. Available: http://arxiv.org/abs/1301.3781 4, 9

[13] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to Sequence Learning with Neural Networks," in *Advances in Neural Information Processing Systems*, vol. 27. Curran Associates, Inc., 2014. [Online]. Available: https://proceedings.neurips.cc/paper/2014/hash/a14ac55a4f27472c5d894ec1c3c743d2-Abstract.html 4, 9

[14] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "A general path-based representation for predicting program properties," *ACM SIGPLAN Notices*, vol. 53, no. 4, pp. 404–419, Jun. 2018. [Online]. Available: http://doi.org/10.1145/3296979.3192412 4, 9, 12, 13

[15] ——, "Code2vec: Learning Distributed Representations of Code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 40:1–40:29, Jan. 2019. [Online]. Available: http://doi.org/10.1145/3290353 4, 10, 12, 13, 19, 20, 23, 49, 50, 52, 53, 54, 62, 68

[16] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating Sequences from Structured Representations of Code," 2019. [Online]. Available: https://openreview.net/forum?id=H1gKYo09tX 4, 49, 53, 54, 62, 68, 69

[17] M. Fowler and M. Foemmel, "Continuous integration," 2006. [Online]. Available: https://moodle2019-20.ua.es/moodle/pluginfile.php/2228/mod_resource/content/2/martin-fowler-continuous-integration.pdf 5

[18] D. Coimbra, S. Reis, R. Abreu, C. Păsăreanu, and H. Erdogmus, "On using distributed representations of source code for the detection of C security vulnerabilities," *arXiv:2106.01367 [cs]*, Jun. 2021, arXiv: 2106.01367. [Online]. Available: http://arxiv.org/abs/2106.01367 5, 19, 23, 49

[19] R. Compton, E. Frank, P. Patros, and A. Koay, "Embedding Java Classes with code2vec: Improvements from Variable Obfuscation," in *Proceedings of the 17th International Conference on Mining Software Repositories*. New York, NY, USA: Association for Computing Machinery, Jun. 2020, pp. 243–253. [Online]. Available: http://doi.org/10.1145/3379597.3387445 5, 11, 19, 20, 21, 22, 23, 49, 54, 64, 68

[20] P. Morrison, K. Herzig, B. Murphy, and L. Williams, "Challenges with applying vulnerability prediction models," in *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security*, ser. HotSoS '15. New York, NY, USA:

Association for Computing Machinery, Apr. 2015, pp. 1–9. [Online]. Available: http://doi.org/10.1145/2746194.2746198 5, 23

[21] F. Chollet, *Deep learning with Python*. Shelter Island, New York: Manning Publications Co, 2018, oCLC: ocn982650571. 7

[22] Y. Bengio, R. Ducharme, P. Vincent, and C. Janvin, "A neural probabilistic language model," *The Journal of Machine Learning Research*, vol. 3, no. null, pp. 1137–1155, 2003. 8

[23] J. Pennington, R. Socher, and C. D. Manning, "GloVe: Global Vectors for Word Representation," in *Empirical Methods in Natural Language Processing (EMNLP)*, 2014, pp. 1532–1543. [Online]. Available: http://www.aclweb.org/anthology/D14-1162 9

[24] Y. Fang, S. Han, C. Huang, and R. Wu, "TAP: A static analysis model for PHP vulnerabilities based on token and deep learning technology," *PLOS ONE*, vol. 14, no. 11, p. e0225196, Nov. 2019, publisher: Public Library of Science. [Online]. Available: https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0225196 9

[25] G. Grieco, G. Grinblat, L. Uzal, S. Rawat, J. Feist, and L. Mounier, "Toward Large-Scale Vulnerability Discovery using Machine Learning," 2016. 9

[26] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola, "Dive into Deep Learning," *arXiv:2106.11342 [cs]*, Jun. 2021, arXiv: 2106.11342 version: 1. [Online]. Available: http://arxiv.org/abs/2106.11342 9

[27] A. Ben-David, "Comparison of classification accuracy using Cohen's Weighted Kappa," *Expert Systems with Applications*, vol. 34, no. 2, pp. 825–832, Feb. 2008. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957417406003435 11

[28] D. Chicco, M. J. Warrens, and G. Jurman, "The Matthews Correlation Coefficient (MCC) is More Informative Than Cohen's Kappa and Brier Score in Binary Classification Assessment," *IEEE Access*, vol. 9, pp. 78 368–78 381, 2021, conference Name: IEEE Access. 11

[29] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to Represent Programs with Graphs," *arXiv:1711.00740 [cs]*, May 2018, arXiv: 1711.00740. [Online]. Available: http://arxiv.org/abs/1711.00740 12

[30] S. M. Radack, "The Common Vulnerability Scoring System (CVSS)," Oct. 2007, last Modified: 2020-01-27T16:24-05:00. [Online]. Available: https://www.nist.gov/publications/common-vulnerability-scoring-system-cvss 14

[31] S. Rahaman, Y. Xiao, S. Afrose, F. Shaon, K. Tian, M. Frantz, M. Kantarcioglu, and D. D. Yao, "CryptoGuard: High Precision Detection of Cryptographic Vulnerabilities in Massive-sized Java Projects," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. New York, NY, USA: Association for Computing Machinery, Nov. 2019, pp. 2455–2472. [Online]. Available: http://doi.org/10.1145/3319535.3345659 16

[32] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks," Sep. 2019, arXiv:1909.03496 [cs, stat] version: 1. [Online]. Available: http://arxiv.org/abs/1909.03496 19

[33] T. Baptista, N. Oliveira, and P. R. Henriques, "Using Machine Learning for Vulnerability Detection and Classification," in *10th Symposium on Languages, Applications and Technologies (SLATE 2021)*, ser. Open Access Series in Informatics (OASIcs), R. Queirós, M. Pinto, A. Simões, F. Portela, and M. J. Pereira, Eds., vol. 94. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, pp. 14:1–14:14, iSSN: 2190-6807. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2021/14431 19, 62

[34] H. J. Kang, T. F. Bissyandé, and D. Lo, "Assessing the generalizability of code2vec token embeddings," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, Nov. 2019, pp. 1–12. [Online]. Available: http://doi.org/10.1109/ASE.2019.00011 19

[35] B. Elema, A. Hommersom, and H. Vranken, "Finding Chinks in the Armour Software Vulnerability Prediction Using Deep Learning on Graph Representations of Source Code," Master's thesis, Open University of the Netherlands, Heerlen, Mar. 2020. 20, 53, 54

[36] P. Black, "SARD: Thousands of Reference Programs for Software Assurance," *Journal of Cyber Security and Information Systems*, vol. 5, pp. 6–13, Oct. 2017. [Online]. Available: https://www.nist.gov/publications/sard-thousands-reference-programs-software-assurance 27, 28, 36

[37] Center for Assured Software National Security Agency, "Juliet Test Suite v1.2 for Java," Dec. 2012. [Online]. Available: https://samate.nist.gov/SARD/downloads/documents/Juliet_Test_Suite_v1.2_for_Java_-_User_Guide.pdf 29, 34

[38] B. Stivalet and E. Fong, "Large Scale Generation of Complex and Faulty PHP Test Cases," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. Chicago, IL: IEEE, Apr. 2016, pp. 409–415. [Online]. Available: https://ieeexplore.ieee.org/document/7515499/ 33

[39] P. E. Black, "Juliet 1.3 Test Suite: Changes From 1.2," Jun. 2018, last Modified: 2021-05-04T09:23-04:00. [Online]. Available: https://www.nist.gov/publications/juliet-13-test-suite-changes-12 34

[40] A. Wagner and J. Sametinger, "Using the Juliet Test Suite to Compare Static Security Scanners," Aug. 2014. 34

[41] J. Kronjee, A. Hommersom, and H. Vranken, "Discovering Software Vulnerabilities Using Data-flow Analysis and Machine Learning: 13th International Conference on Availability, Reliability and Security," *Proceedings of the 13th International Conference on Availability, Reliability and Security*, 2018, place: New York, NY, USA Publisher: acm. 38, 53, 54

[42] G. Bhandari, A. Naseer, and L. Moonen, "CVEfixes: automated collection of vulnerabilities and their fixes from open-source software," in *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*. New York, NY, USA: Association for Computing Machinery, Aug. 2021, pp. 30–39. [Online]. Available: http://doi.org/10.1145/3475960.3475985 38

[43] D. R. Wallace, A. H. Watson, and T. J. McCabe, "Structured testing :: a testing methodology using the cyclomatic complexity metric," National Institute of Standards and Technology, Gaithersburg, MD, Tech. Rep. NIST SP 500-235, 1996, edition: 0. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication500-235.pdf 41

[44] P. Norvig and S. Russel, "Artificial Intelligence: A Modern Approach," in *Artificial Intelligence: A Modern Approach*, 3rd ed. Pearson, Dec. 2009, p. 709. 49

[45] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu, "VulPecker: an automated vulnerability detection system based on code similarity analysis," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, ser. ACSAC '16. New York, NY, USA: Association for Computing Machinery, Dec. 2016, pp. 201–213. [Online]. Available: https://doi.org/10.1145/2991079.2991102 53, 54

[46] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "VulDeePecker: A Deep Learning-Based System for Vulnerability Detection," *Proceedings 2018 Network and Distributed System Security Symposium*, 2018, arXiv: 1801.01681. [Online]. Available: http://arxiv.org/abs/1801.01681 65, 68

[47] W. de Kraker, H. Vranken, and A. Hommersom, "Combining program slicing and graph neural networks to detect software vulnerabilities," Master's thesis, Open University of the Netherlands, Heerlen, Jul. 2022. 53, 54, 65

[48] A. Yeh, "More accurate tests for the statistical significance of result differences," in *COLING 2000 Volume 2: The 18th International Conference on Computational Linguistics*, 2000. [Online]. Available: https://aclanthology.org/C00-2137 53, 63, 67, 68

[49] R. Ferenc, Z. Tóth, G. Ladányi, I. Siket, and T. Gyimóthy, "A public unified bug dataset for java and its assessment regarding metrics and bug prediction," *Software Quality Journal*, vol. 28, no. 4, pp. 1447–1506, Dec. 2020. [Online]. Available: https://doi.org/10.1007/s11219-020-09515-0 64

[50] S. L. Abebe, S. Haiduc, P. Tonella, and A. Marcus, "The Effect of Lexicon Bad Smells on Concept Location in Source Code." IEEE Computer Society, Sep. 2011, pp. 125–134. [Online]. Available: http://www.computer.org/csdl/proceedings-article/scam/2011/06065171/12OmNzBOi1B 65

[51] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities," *IEEE Transactions on Dependable and Secure Computing*, pp. 1–1, 2021, arXiv: 1807.06756. [Online]. Available: http://arxiv.org/abs/1807.06756 65

WEB LINKS

[52] "Heartbleed Bug." [Online]. Available: https://heartbleed.com/

[53] "Log4j – Apache Log4j Security Vulnerabilities." [Online]. Available: https://logging.apache.org/log4j/2.x/security.html

[54] NIST, "NVD - Home." [Online]. Available: https://nvd.nist.gov/

[55] MITRE, "CWE - Common Weakness Enumeration." [Online]. Available: https://cwe.mitre.org/

[56] Lee, Adriana, "How Codenomicon Found The Heartbleed Bug Now Plaguing The Internet," Apr. 2014. [Online]. Available: https://readwrite.com/heartbleed-security-codenomicon-discovery/

[57] https://www.facebook.com/ryanwneal, "Heartbleed Bug: CEO David Chartier Explains How Codenomicon Found The Massive Internet Security Breach," Apr. 2014, section: Internet. [Online]. Available: https://www.ibtimes.com/heartbleed-bug-ceo-david-chartier-explains-how-codenomicon-found-massive-internet-15697

[58] L. Eadicicco, "How A Group Of Engineers Uncovered The Biggest Bug The Internet Has Seen In Years." [Online]. Available: https://www.businessinsider.com/heartbleed-bug-codenomicon-2014-4

[59] Google, "linear-relationships: word embedding vector space." [Online]. Available: https://developers.google.com/machine-learning/crash-course/images/linear-relationships.svg

[60] R. Řehůřek and P. Sojka, "Software Framework for Topic Modelling with Large Corpora," Valetta, MT, May 2010, pages: 45–50 Series: Proceedings of LREC 2010 workshop New Challenges for NLP Frameworks original-date: 2011-02-10T07:43:04Z. [Online]. Available: http://is.muni.cz/publication/884893/en

[61] "Code2vec," Oct. 2022, original-date: 2018-07-24T03:40:20Z. [Online]. Available: https://github.com/tech-srl/code2vec

[62] "astminer," Oct. 2022, original-date: 2018-12-14T16:37:33Z. [Online]. Available: https://github.com/JetBrains-Research/astminer

[63] "id2vec," Jun. 2022, original-date: 2019-12-29T20:47:39Z. [Online]. Available: https://github.com/tech-srl/id2vec

[64] NIST, "CVE - CVE." [Online]. Available: https://cve.mitre.org/

[65] ——, “CVE - CVE and NVD Relationship.” [Online]. Available: https://cve.mitre.org/about/cve_and_nvd_relationship.html

[66] “NVD - CVSS Severity Distribution Over Time.” [Online]. Available: https://nvd.nist.gov/general/visualizations/vulnerability-visualizations/cvss-severity-distribution-over-time#CVSSSeverityOverTime

[67] “OWASP ZAP | OWASP Foundation.” [Online]. Available: https://owasp.org/www-project-zap/

[68] “OWASP WebGoat | OWASP Foundation.” [Online]. Available: https://owasp.org/www-project-webgoat/

[69] OWASP Foundation, “OWASP Top Ten Web Application Security Risks | OWASP,” Oct. 2021. [Online]. Available: https://owasp.org/www-project-top-ten/

[70] “CWE - CWE-327: Use of a Broken or Risky Cryptographic Algorithm (4.9).” [Online]. Available: https://cwe.mitre.org/data/definitions/327.html

[71] MITRE, “CWE - 2021 CWE Top 25 Most Dangerous Software Weaknesses,” Jul. 2021. [Online]. Available: https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html

[72] OWASP Foundation, “OWASP Top 10 - 2010,” 2010. [Online]. Available: https://owasp.org/www-pdf-archive/OWASP_Top_10_-_2010.pdf

[73] ——, “OWASP Top 10 - 2013,” 2013. [Online]. Available: https://owasp.org/www-pdf-archive/OWASP_Top_10_-_2013.pdf

[74] ——, “OWASP Top Ten 2017 | 2017 Top 10 | OWASP Foundation,” 2017. [Online]. Available: https://owasp.org/www-project-top-ten/2017/Top_10.html

[75] “Hyperparameter Optimization with Weights & Biases.” [Online]. Available: https://wandb.ai/site/sweeps,http://wandb.ai/site/sweeps

[76] “A03 Injection - OWASP Top 10:2021.” [Online]. Available: https://owasp.org/Top10/A03_2021-Injection/

[77] D. Mathijssen, “Detecting software vulnerabilities in source code and the influence of variable naming,” Oct. 2022, original-date: 2022-10-31T09:56:38Z. [Online]. Available: https://github.com/daveymathijssen/DataSetPreparation-thesis-vulnerability-detection

[78] “Test suites.” [Online]. Available: https://samate.nist.gov/SARD

[79] “The .NET Compiler Platform,” Aug. 2022, original-date: 2015-01-11T02:39:03Z. [Online]. Available: https://github.com/dotnet/roslyn

[80] “ANTLR.” [Online]. Available: https://www.antlr.org/

[81] thelma.allen@nist.gov, "SARD Acknowledgments and Test Case Descriptions," Mar. 2021, last Modified: 2021-05-17T11:59-04:00. [Online]. Available: https://www.nist.gov/itl/ssd/software-quality-group/sard-acknowledgments-and-test-case-descriptions

[82] "C# Vulnerability Test Suite 105." [Online]. Available: https://samate.nist.gov/SARD

[83] B. Stivalet, "C# Vulnerability test suite," Jan. 2016, original-date: 2015-10-21T15:36:57Z. [Online]. Available: https://github.com/stivalet/C-Sharp-Vuln-test-suite-gen

[84] "Juliet C# 1.3." [Online]. Available: https://samate.nist.gov/SARD

[85] H. Hasan, "Test Case 61774," Dec. 2011. [Online]. Available: https://samate.nist.gov/SARD/test-cases/61774/versions/1.0.0

[86] ——, "Test Case 61774 new version," Aug. 2022. [Online]. Available: https://samate.nist.gov/SARD/test-cases/61774/versions/1.1.0

[87] Y. Somda, "Guesslang documentation — Guesslang 2.2.2 documentation," Sep. 2021. [Online]. Available: https://guesslang.readthedocs.io/en/latest/contents.html#how-does-guesslang-guess

[88] P. Bengtsson, L. Coursen, R. Sese, R. Sewell, M. Pollard, G. Park, and S. Guntrip, "Github: About repository languages," Dec. 2021. [Online]. Available: https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/about-repository-languages

[89] D. Mathijssen and U. Alon, "Preprocessor step disposing numbers in (variable) names · Issue #164 · tech-srl/code2vec," Oct. 2022. [Online]. Available: https://github.com/tech-srl/code2vec/issues/164

[90] "Install TensorFlow with pip." [Online]. Available: https://www.tensorflow.org/install/pip

[91] "Google Colaboratory." [Online]. Available: https://colab.research.google.com/

[92] "Google Colab faq." [Online]. Available: https://research.google.com/colaboratory/faq.html#resource-limits

[93] D. Mathijssen, "Code2vec," Oct. 2022, original-date: 2022-10-09T14:47:18Z. [Online]. Available: https://github.com/daveymathijssen/code2vec

[94] "NVD - CVE-2020-8416." [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2020-8416#VulnChangeHistorySection