Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Bachelor Thesis

Adapting a C Deep Learning Vulnerability Detection System to Java

A Targeted Reproduction of VulCNN

Author: Aris Tsimidakis (2730979)

1st supervisor:	Fabio Massaci
daily supervisor:	Johannes Härtel
2nd reader:	Johannes Härtel

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

"I am the master of my fate, I am the captain of my soul"
from Invictus*, by William Ernest Henley*

Abstract

This study is a targeted replication of a Deep Learning-based source code vulnerability detection system named VulCNN. VulCNN was originally designed for classifying functions written in C, and this study extends its application to Java. There are three main objectives to this research: (a) to evaluate the performance of VulCNN in detecting vulnerabilities in C functions, (b) to assess the impact of different sentence embedding techniques on the results of the classification, and (c) to train a new VulCNN model for Java and determine if it achieves comparable accuracy when applied to different programming languages and datasets.

To achieve these goals, a series of experiments was conducted on both the C and Java implementations of VulCNN, focusing on the model's performance in classifying vulnerable functions. The experimental results indicate that both implementations achieve similar accuracy levels, suggesting that the approach is likely transferable to a range of programming languages with promising outcomes. However, the results also reveal significant limitations: the synthetic dataset used to train VulCNN does not provide sufficient diversity for the model to effectively generalize and distinguish different vulnerability patterns. This finding underscores the need for more comprehensive and varied training datasets, utilizing source code from real-world products.

Contents

List of Figures

List of Tables

1

Introduction

Over the last years, software vulnerabilities have become an increasingly large threat, with the number of new Common Vulnerabilities and Exposures discovered over the last 7 years having increased over 350% [2]. Additionally, the financial impact of cybercrime exceeded eight trillion U.S. dollars in 2023, and is expected to approach fourteen trillion U.S. dollars by 2027 [3]. These cyberattacks are a result of flaws in software or hardware called vulnerabilities. A vulnerability is defined as a weakness in an IT system that can be exploited by an attacker to deliver a successful attack. They can occur through flaws, features or user error, and attackers will look to exploit any of them, often combining one or more, to achieve their end goal [4]. As software products increase in complexity, identifying and addressing vulnerabilities manually becomes increasingly challenging. This is why we have turned to automated solutions for vulnerability detection, since they can provide more accuracy and are not prone to human errors. The rapid advancements in Deep Learning have laid ground for extensive research, aiming to utilize these models for vulnerability detection in software. Numerous approaches have already proven to be more accurate in detecting vulnerabilities compared to traditional methods such as static analysis [5].

In this Thesis, I aim to replicate a Deep Learning vulnerability detection system for C source code proposed in 2022, named VulCNN [1]. There are three main research questions behind this work. Firstly, I aim to evaluate the performance of VulCNN to determine whether the proposed approach is indeed a scalable and accurate method of classifying vulnerable source code, as claimed by the original researchers. Furthermore, I conduct several experiments on this model in order to test how its performance deviates based on the sentence embedding performed in the source code. This aims to understand the ability of the model to generalize vulnerability patterns across multiple programming languages without the need for training one model for each language. Lastly, I train a new version

of VulCNN focused on classifying Java vulnerabilities, in order to test the scalability and adaptability of the approach for different programming languages and datasets.

2

Original Study Design

This section provides a high-level overview of the original study that proposed VulCNN, conducted by Yueming Wu, Deqing Zou, Shihan Dou, Wei Yang, Duo Xu and Hai Jin [1].

2.1 Methodology description

2.1.1 Overview

VulCNN proposes a new graph-based method for extracting vulnerability features from source code by using its Program Dependence Graph (PDG) to create an image, which gets classified by a Convolutional Neural Network (CNN) as either vulnerable or not vulnerable. CNNs are a type of Artifitial Neural Network for processing inputs of a grid-like topology [6]. This makes them very effective in tasks such as image classification. The model is trained on data containing multiple vulnerable and safe functions written in the C programming language. Through this process, it learns specific patterns that make a function vulnerable and can then be deployed in order to classify new, unseen functions.

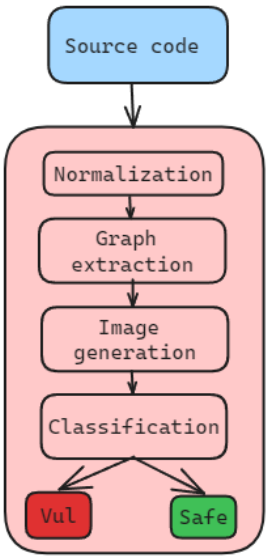

Figure 2.1: VulCNN's pipeline

2.1.2 Preprocessing steps

The process that VulCNN follows to turn a function's source code to an image consists of four main steps. First, the input source code is normalized by removing all comments and renaming user-defined variables and functions to standardized names.

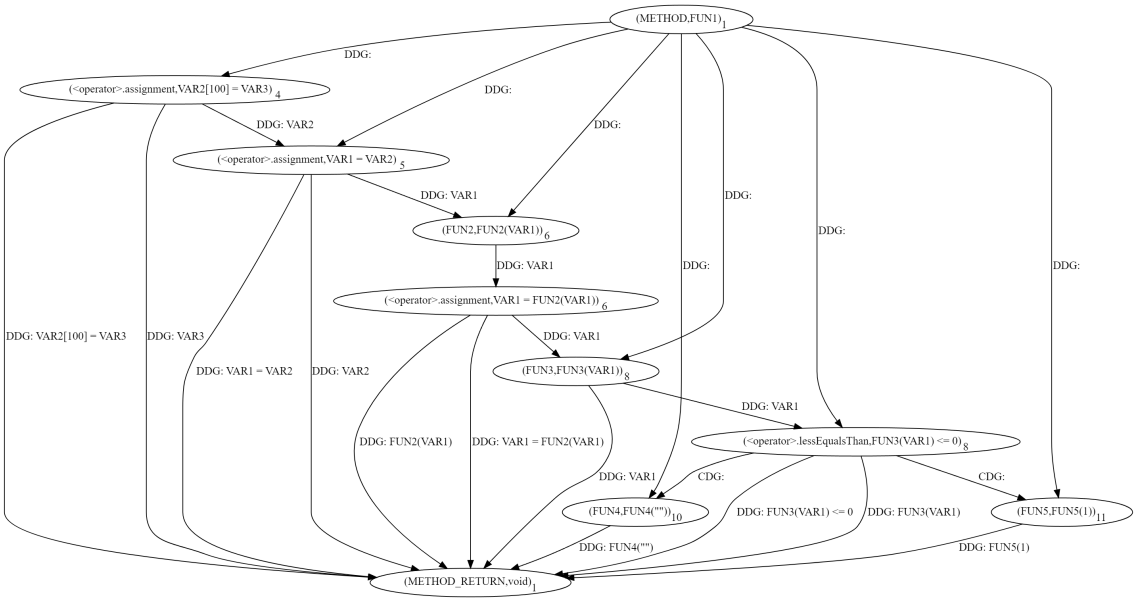

Figure 2.2: An example of a Program Dependency Graph

This step is crucial, as it has been shown that former DL-based techniques for vulnerability detection do not actually learn the cause of vulnerabilities, but rather classify based on specific artifacts from the code like variable/function names [7]. Normalizing the source code is an effective way of mitigating this risk. The next preprocessing step of VulCNN involves dependence graph extraction, where Joern [8], a static analysis tool, is used to extract the function's Program Dependence Graph (PDG). Generally, a PDG is an efficient way of representing a program's control flow and data flow at once. These are directed graphs, where each node represents a program statement and the edges represent either a control dependence or a data dependence between two nodes. An example of a PDG extracted from a function in the SARD dataset can be seen in figure 2.2.

After the program's PDG has been extracted, it is used in order to create an image. First, a sent2vec [9] model is used to perform sentence embedding on each of the nodes of the graph. This transforms them from natural language "sentences" to fixed-length vectors. Subsequently, centrality analysis is conducted on each node, calculating three different centralities: Closeness centrality, Katz centrality and degree centrality. Degree centrality is defined as the number of links incident upon a node (i.e., the number of ties that a node has) [10]. Closeness centrality of a node is the average length of the shortest paths between the node and all other nodes in the graph [11]. Katz centrality measures

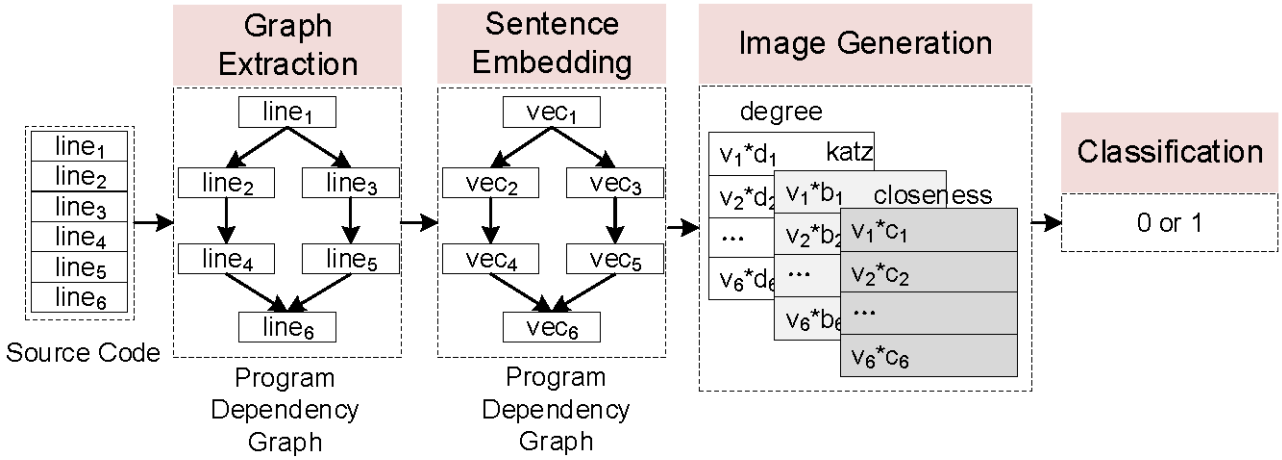

Figure 2.3: The process of turning a function's source code to an image (direct copy from [1])

the number of all nodes that can be connected through a path, with the connections to nodes being penalized [12]. This centrality analysis helps to realize the semantics of each line of code, as well as encapsulate the ways that it affects or is affected by the rest of the lines. After the centrality analysis, we have obtained three vectors, each containing one weight (centrality value) for each node of the graph. Each sentence embedding vector (line of code) is then multiplied with its corresponding weight in each centrality vector, effectively creating three channels. These channels are then arranged to create an image, in which the red, green and blue channels are represented by the degree, Katz and closeness channels respectively. Figure 2.3, which was taken from the original paper that introduced VulCNN, depicts the process of turning a function's source code to an image. After the preprocessing steps are completed, a Convolutional Neural Network (CNN) is trained in order to classify these images as either vulnerable or not vulnerable.

2.1.3 Model architecture

The model takes as an input an image of size $(num\_channels * hidden\_size * max\_len)$, where $num\_channels$ represents the number of channels in the image, $hidden\_size$ represents the dimensionality of the sentence embedding vectors and max_len represents the lines of code that are used to create each image. This results in $3 * 128 * 100$. The input images are first fed in a set of convolution layers, each with a different filter size, ranging from 1 to 10. Each of these 10 layers contains 32 filters, which convolve over the input to extract local features. After the convolution operation, ReLU activation and max-pooling

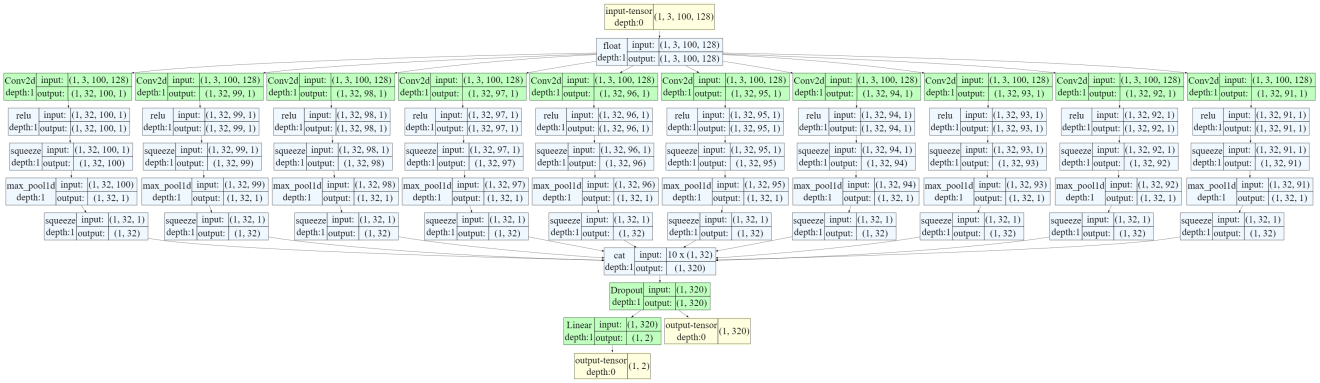

Figure 2.4: The architecture of VulCNN

is applied in order to extract the most important features from the image. The outputs of the convolutional layers are then concatenated and flattened in order to create the final feature vector. After the convolutional layers, the model utilizes a dropout layer with a dropout rate of 0.1. This layer is responsible for randomly setting a fraction (10%) of input units to zero, so that co-adaptation between neurons is avoided and the model does not overfit on the training data. Lastly, the feature vector goes in a fully connected dense layer, which outputs one positive class probability and one negative class probability for the given input. Figure 2.4 showcases the aforementioned architecture in detail.

2.2 Experiments

After completing the training of VulCNN, the researchers conduct four experiments in order to effectively test VulCNN's accuracy and scalability.

1. They perform a threshold experiment in order to determine the optimal number of lines of code for generating images, aiming for the highest accuracy while keeping the memory usage to a minimum. These experiments were conducted using a range of threshold values, ranging from 50 to 200.

2. In order to test the model's detection performance, they benchmark it against eight widely used vulnerability detection systems of varying kinds, including both static analysis and deep learning based tools.

3. To prove that the centrality analysis does indeed play an important role in identifying features that make a function vulnerable, the researchers also train VulCNN

without-centrality (VUlCNN-wc), which takes as input the sentence embedding vectors without multiplying them by the centrality values of each node.

4. Finally, to test the model's performance in real-world scenarios, they conduct a case study using three open-source projects: Libav [13], SeaMonkey [14] and Xen[15]. Using VulCNN, the researchers scan 600.233 functions from both recent and old versions of these projects, aiming to detect vulnerabilities within them.

2.3 Findings

This section presents the findings of the experiments conducted by Wu et al., as explained in section 2.2.

1. After the threshold experiment, the research conclude on a threshold of 100 lines of code as the optimal point for both high accuracy and minimized memory usage.

2. VulCNN managed to perform better than eight widely used vulnerability detection systems, achieving approximately a 90% true positive rate, 80% true negative rate and 83% accuracy.

3. The findings also show that the system has an impressive runtime overhead, with most of it coming from the process of extracting the program dependence graphs. Even so, the mean overhead for this step was 1.71 seconds, which can be reduced marginally by using different tools or optimization methods. The mean runtime overhead for sentence embedding, image generation, and classification were 489 microseconds, 0.26 seconds, and 41 microseconds respectively.

4. After scanning the aforementioned open-source projects, VulCNN managed to discover 73 vulnerabilities. Out of these vulnerabilities, 52 were still present in the products and 21 were found on older versions and had already been patched.

This are quite promising results that show that VulCNN offers both very good scalability, as it can be used efficiently to test large-scale software products, as well as good accuracy.

3

Replication Study Design

3.1 Introduction

This replication study for VulCNN consisted of three main phases, which will be discussed in detail under this section. These phases can be identified as follows: Original model testing, sentence embedding testing, and Java model testing. Each of these three experimental phases aims to answer one of the research questions of this study:

1. Is VulCNN indeed an accurate and scalable DL-based approach for detecting vulnerabilities? And if so, how does it perform on real-world data instead of synthetic functions?

2. How big of a role does sentence embedding play when it comes to feature extraction for vulnerability detection? Is it possible for a model that is trained on one programming language to generalize vulnerability patterns well enough to also detect them on other programming languages?

3. Does VulCNN perform equally well when retrained on a new dataset and programming language?

3.2 Original model testing

The first phase of this replication study involved training VulCNN using the SARD dataset, which was originally used by the researchers that introduce the model. Next, a series of tests was performed to evaluate the model's performance. The training dataset was obtained from VulCNN's GitHub repository. During this phase, no changes were made regarding the preprocessing of the data or the hyper-parameters of the model, to ensure

that the replication mirrored the original model as closely as possible. One important change that was made, however, was the use of stratified k-fold cross-validation instead of regular k-fold cross-validation. The choice to deviate from the original approach and use stratified k-fold instead was taken because it ensures that each fold is closely representing the original dataset, having a balanced amount of vulnerable and not vulnerable entries in each subset. In order to train and define the model, the pytorch library [16] was used, while the networkx [17] and pygraphviz [18] modules were used in order to parse .dot files and apply centrality analysis to the program dependency graphs. Since stratified k-fold cross-validation was used with $k = 5$, there were 5 different models trained by the end of the fitting process. Each model is trained for 100 epochs, and after each epoch of training, both the training and validation metrics and the architecture of the model for the specific epoch are saved. This process results in a set of 500 different models. In order to choose which of these models to use for the experiments of this study, I isolated 5 models, each performing the best at one of the following metrics during their validation phase:

- Lowest false negative rate (FNR)
- Lowest false positive rate (FPR)
- Lowest loss
- Highest accuracy
- Highest f1-score

Afterwards, I evaluated each of these model's true positive rate and true negative rate using the partition of the SARD dataset that was reserved for testing. The results of this process can be seen in table 3.1. The columns of the table signify the metric in which each of the models performed the best, and the rows signify the metric in which they were evaluated. As a result of this experiment, the model that had the highest accuracy was chosen for the experiments. This was because it had the highest as well as more balanced average in its results.

The testing of the selected model was conducted on two datasets: First, by utilizing entries from the SARD Dataset that were split from the training/validation data for this purpose. Second, in order to test its performance for real-world vulnerabilities, I utilized the MSR dataset, a dataset presented by Fan et al. [19], consisting of 1.121 vulnerable functions and 1.101 safe functions, sourced from C/C++ GitHub repositories.

	FPR	FNR	Loss	Accuracy	F1-Score
TPR	80.4%	66.4%	76.6%	84%	66.4%
TNR	67.5%	90.1%	87.5%	82.7%	90.1%

Table 3.1: Performance evaluation on the top-performing model for each metric

3.3 Sentence embedding testing

The second phase of experiments aims to evaluate the importance of sentence embedding in the model's performance. First, I collected a new dataset for Java vulnerabilities, which I obtained from Juliet test suite [20]. Juliet is a fully synthetic dataset provided by the U.S National Institute of Standards and Technology (NIST) [21]. The dataset that I extracted comprises 25.584 vulnerable functions and 59.507 non-vulnerable functions, organized by Common Weakness Enumeration (CWE). A detailed graphical representation of the entries in the dataset can be seen in figure 3.1. It is important to note that the functions in this suite are crafted solely for the purpose of creating the dataset and have not been obtained from real-world vulnerabilities. This poses a potential threat to the validity of this research, which will be further discussed in chapter 5. In order to apply sentence embedding to java source code, a sent2vec model [9] was trained from scratch using the source code in Juliet as its training data.

After collecting and processing the dataset, the following experiments were conducted:

1. Evaluating the original model's performance on C files with Java sentence embedding.
2. Evaluating the original model's performance on Java files with C sentence embedding.
3. Evaluating the original model's performance on Java files with Java sentence embedding.

These experiments were designed in order to test the significance of sentence embedding for the specific methodology, since there will be a comparative analysis for the model's performance on both C and Java files using different embeddings. This will provide an insight on whether the model is capable of generalizing vulnerability patterns across multiple languages.

3.4 Training and testing new model

The final phase of the experimental study involved training a new version of VulCNN, using the Java dataset and a Java sentence embedding model. This experiment aimed to

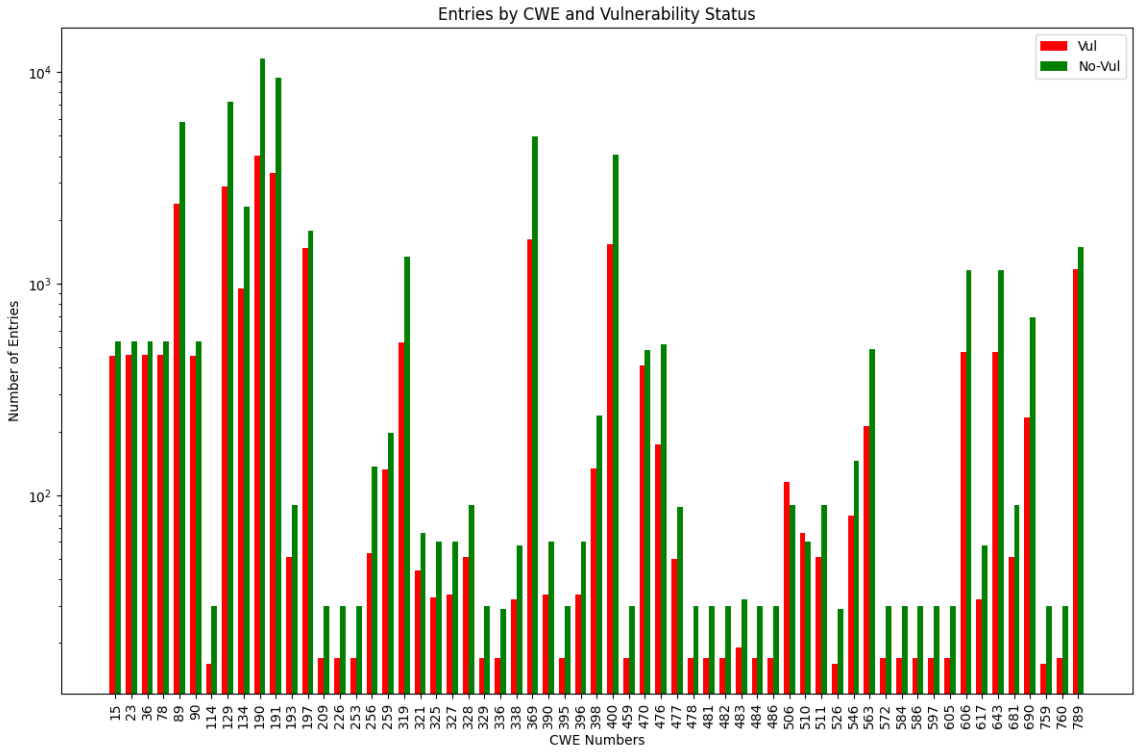

Figure 3.1: An analysis of the entries in the Java dataset

determine whether the proposed approach of VulCNN achieves comparable accuracy when transferred in different programming languages. In order to replicate the model as closely as possible, no hyper-parameter tuning was performed; instead, the original hyper-parameters and validation methods were used. These hyper-parameters are presented in table 3.4. The Juliet dataset was initially split in 80% training/validation and 20% testing data. This initial split ensures that a dedicated portion of the data is reserved for final evaluation, providing a measure of the model's performance on unseen data. The 80% reserved for training and validation of the model during fitting, was then split further using stratified k-fold cross-validation with $k = 5$, just as for the original version of VulCNN. This new model was evaluated on the reserved 20% of the Juliet dataset.

loss function	**batch size**	**learning rate**	**training epochs**
Cross Entropy Loss	32	0.001	100

Table 3.2: Hyper-parameter settings for the Java version of VulCNN

4

Results

4.1 Introduction

This chapter presents the results from the experiments conducted as part of this replication study, described in Chapter 3. The results are divided into three sections: original model testing, sentence embedding testing, and Java model testing. Each section provides insights into the performance of the model for each experiment. The model's performance was evaluated based on the achieved True Positive Rate (TPR), True Negative Rate (TNR), and Accuracy when classifying functions.

True Positive Rate (also known as recall) measures the proportion of actual positives (vulnerable functions) that were correctly classified by the model. It is calculated using the formula $TPR = \frac{TP}{TP+FN}$, where TP represents the number of correctly classified positives, and FN represents the number of actual positives that were incorrectly classified as negatives. True Negative Rate measures the proportion of actual negatives that were correctly classified by the model and is calculated by the formula $TNR = \frac{TN}{TN+FP}$. Finally, accuracy measures the proportion of all samples that were correctly predicted by the model and is calculated as follows: $Acc = \frac{TP+TN}{TP+TN+FP+FN}$.

For better readability and easier comparison, table 4.1 presents the results of all the experiments at once. The experiment IDs that are used in this table correspond to each of the three experimental phases of this study:

1. Original Model Testing
2. Sentence Embedding Testing
3. Java Model Testing

Each experiment and its results are discussed separately in detail in the following sections.

ID	Train Set	Test Set	Test Set Type	Embedding	TPR	TNR	ACC
1	SARD	SARD	Synthetic (C)	C	72.6%	89.75%	83.4%
1	SARD	MSR	Real-World (C)	C	45.3%	60.4%	52.8%
2	SARD	SARD	Synthetic (C)	Java	82.5%	17.4%	41.4%
2	SARD	MSR	Real-World (C)	Java	75.3%	36.8%	56.2%
2	SARD	Juliet	Synthetic (Java)	C	60.1%	10.7%	28.1%
2	SARD	Juliet	Synthetic (Java)	Java	91.9%	29.8%	43.3%
3	Juliet	Juliet	Synthetic (Java)	Java	78.4%	83.6%	82.1%

Table 4.1: Overview of the results of all the experiments conducted in this study.

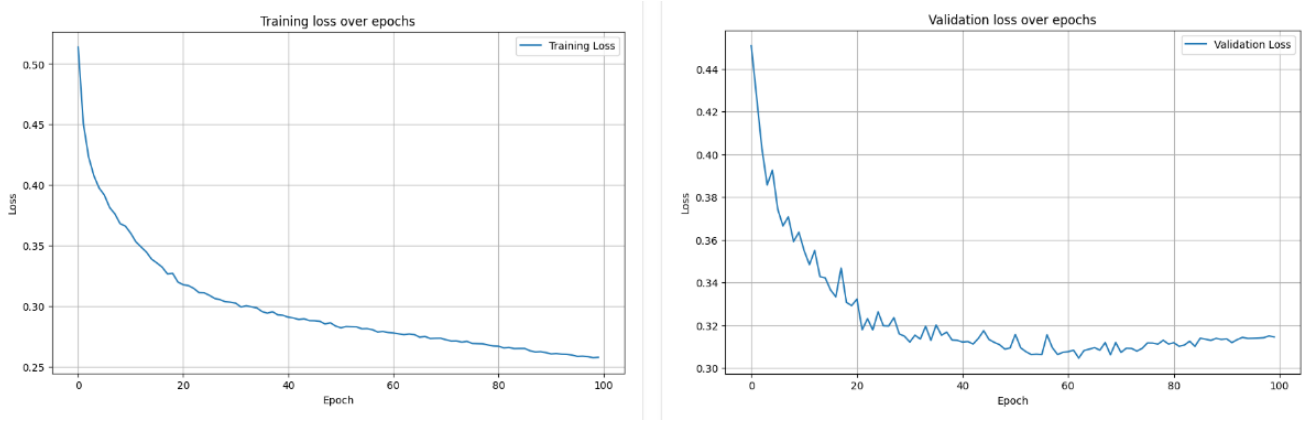

Figure 4.1: (a) Training loss of VulCNN over epochs, (b) Validation loss of VulCNN over epochs

4.2 Original Model Testing

As mentioned in section 3.2, the first step of this study was to replicate the original version of VulCNN as closely as possible and evaluate its performance. The model was tested on two grounds: first, by utilizing a portion of the training dataset that was reserved for testing, and second, by utilizing a real-world vulnerability dataset, MSR [19].

Figures 4.1 and 4.2 show the training/validation loss and accuracy of the model over the 100 epochs of training. Around epoch 65, the validation loss starts increasing while the training loss continues to decrease. Similarly, the validation accuracy starts decreasing while the training accuracy continues to increase. This indicates that the model has started over-fitting on the training set; however, this does not pose an issue due to the method that was used to choose the best model (Section 3.2).

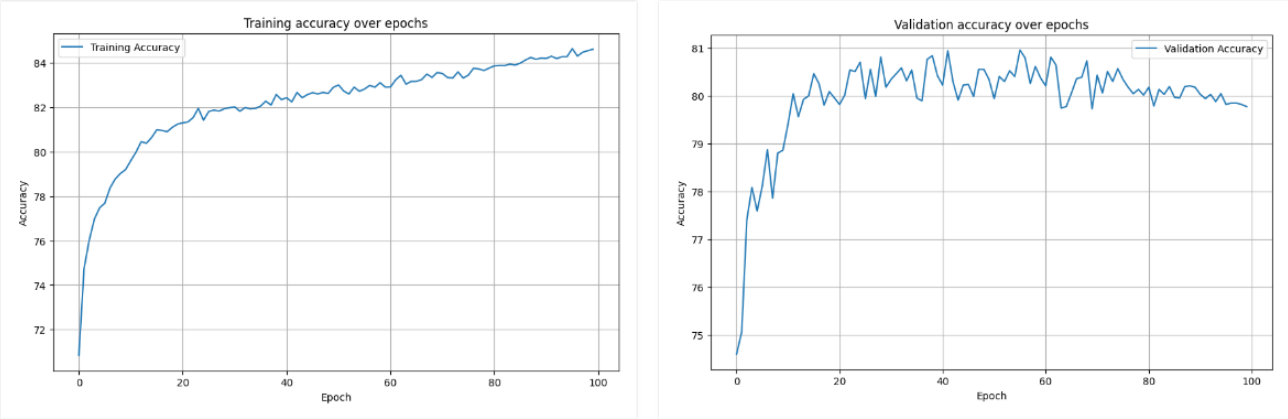

Figure 4.2: (a) Training accuracy of VulCNN over epochs, (b) Validation accuracy of VulCNN over epochs

4.2.1 Testing on the SARD Dataset

The performance metrics of VulCNN when classifying entries in the (synthetic) SARD dataset are presented in Table 4.2. The model achieved a satisfactory accuracy, but there is a notable imbalance between the TPR and TNR, with roughly a 17% performance increase in classifying negatives over positives.

TPR	TNR	Accuracy
72.6%	89.75%	83.4%

Table 4.2: Performance metrics of the original VulCNN model classifying functions from the SARD dataset

4.2.2 Testing on the MSR Dataset

The MSR dataset was proposed by Fan et al. [19], and contains vulnerability data from C and C++ GitHub repositories. The performance metrics of VulCNN classifying functions from this real-world dataset are presented in Table 4.3. The imbalance between the TPR and TNR is evident in this experiment as well. However, there is also a roughly 30% decrease in accuracy compared to the SARD dataset. This outcome is expected, since the model was trained exclusively on samples from the SARD dataset, which do not closely resemble real-world scenarios. Hence, the model faces more difficulty in identifying vulnerability patterns in more complex source code.

TPR	TNR	Accuracy
45.3%	60.4%	52.8%

Table 4.3: Performance metrics of the original VulCNN model classifying functions from the MSR dataset

4.3 Sentence Embedding Testing

This section presents the results of the second experimental phase of this study, aiming to evaluate the ability of VulCNN to generalize vulnerability patterns across different languages and examine its behavior when different sentence embedding techniques are applied to the source code.

4.3.1 Results on C Functions with Java sentence embedding

The performance metrics of VulCNN classifying C functions with C sentence embedding have already been presented in section 4.2. Here, I also present the performance of the model on C functions, but for the following experiments the sentence embedding on the source code has been conducted by a sent2vec model trained on the Java dataset. These results can be found in Table 4.4. Figure 4.3 shows the accuracy of the model for classifying C functions with both C and Java embedding, and for both the real-world and synthetic datasets.

Testing Dataset	TPR	TNR	ACC
SARD	82.5%	17.4%	41.4%
MSR	75.3%	36.8%	56.2%

Table 4.4: Performance metrics of VulCNN for classifying C functions from both datasets with Java sentence embedding

The results indicate that the model's performance significantly drops when Java sentence embedding is used for classifying C functions. This is because the model shows a preference of classifying functions as vulnerable, which is seen due to the big intervals between the achieved TPR and TNR. This is a clear indication of randomness in the results. However, we can not conclude that this randomness comes due to the change in sentence embedding. It also possible that the model has over-fitted on the very homogenous synthetic dataset that it was trained on, so this randomness may come due to the model's inability to generalize vulnerability patterns as a whole.

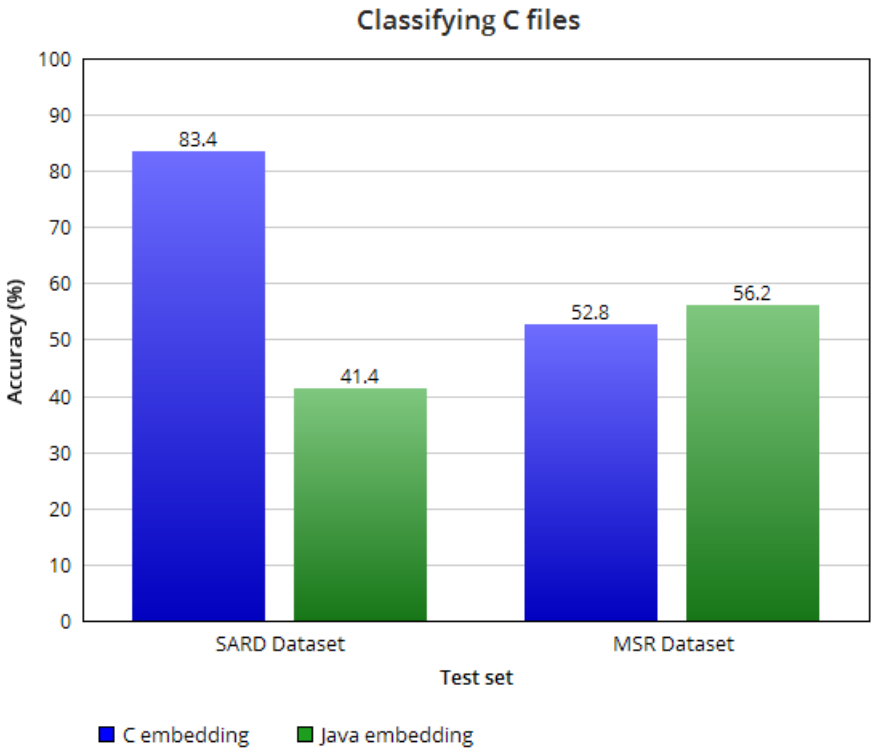

Figure 4.3: Achieved accuracy of VulCNN on C files with both C and Java sentence embedding.

4.3.2 Results on Java Functions

A major motivation for this study was to determine whether VulCNN can generalize well across different programming languages. To test this, I collected a set of Java functions from the Juliet Java test suite that tested the same CWEs as those in the SARD dataset. However, since many vulnerabilities in the SARD dataset are specific to C, the model was tested only on vulnerabilities that were common across the two datasets. The testing on these Java files was conducted with both C and Java sentence embedding. The classification results are presented in Tables 4.5 and 4.6 for C and Java respectively.

Vulnerability Description	TPR	TNR	Acc
Relative Path Traversal	39.8%	18.7%	28.4%
Absolute Path Traversal	39.8%	18.7%	28.4%
OS Command Injection	54.9%	1.7%	26.3%
Uncontrolled Format String	73%	14.2%	31.3%
NULL Dereference from Return	93%	0.0004%	26.2%

Table 4.5: Performance metrics of VulCNN for classifying Java functions with C sentence embedding

Vulnerability Description	TPR	TNR	Acc
Relative Path Traversal	95.4%	1.7%	45%
Absolute Path Traversal	95.4%	1.7%	45%
OS Command Injection	95.4%	1.7%	45%
Uncontrolled Format String	90.3%	7.8%	31.7%
NULL Dereference from Return	83%	16.9%	49.8%

Table 4.6: Performance metrics of VulCNN for classifying Java functions with Java sentence embedding

The results indicate a significant decrease in accuracy when classifying Java functions, regardless of the sentence embedding used. However, when Java sentence embedding is applied on the Java source code, the model shows a substantial preference in classifying the functions as vulnerable. This becomes evident due to the very high TPR and very low TNR respectively. On the other hand, when C sentence embedding is applied, the mean accuracy of the model is lower than the one achieved with java sentence embedding, but the imbalance between the TPR and TNR varies greatly depending on the vulnerability type. As with the results presented in section 4.3.1, we can not be sure that this seemingly

random way of classifying is a consequence of the changes in the input programming language or the sentence embedding that it is applied to it.

4.3.3 Threshold Testing

To thoroughly investigate the reasons behind the deviation in the model's performance when altering the input's sentence embedding or programming language, it is crucial to exclude the possibility of overfitting on the synthetic training dataset. This step is essential to determine whether the model can genuinely generalize vulnerability patterns. If overfitting is not the cause, it can be reasonably assumed that the model's performance degradation is due to the loss of information necessary for identifying vulnerability patterns caused by the differences in sentence embedding or programming language.

To address this, I conducted all previously described experiments and plotted the Receiver Operating Characteristic (ROC) curve of the classifier for each experiment. The ROC curve is a graphical representation of a classifier's True Positive Rate (TPR) against its False Positive Rate (FPR) at various threshold settings. By calculating the Area Under the ROC Curve (AUC), we obtain a metric that indicates the classifier's ability to distinguish between classes. An AUC score of 1 denotes a perfect classifier, whereas an AUC of 0.5 indicates a classifier with performance no better than random chance.

Figure 4.4 displays the ROC curves for each of the six experiments discussed previously. From these plots, it becomes evident that the randomness in classification results is not attributable to variations in language or sentence embedding. While the AUC for classifying synthetic C functions with C sentence embedding appears exceptional, the model performs as a random classifier in all other scenarios, including when classifying real-world C functions with C sentence embedding. This finding indicates that VulCNN is significantly overfitting on its synthetic training dataset and does not genuinely recognize and differentiate vulnerability patterns.

The cause of this overfitting can be traced back to the nature of the functions within the SARD dataset. These synthetic functions are not only highly similar and repetitive but also extremely simplistic, with an average file length of just 25 lines. Consequently, it is logical that the model struggles to generalize vulnerability patterns beyond these synthetic functions.

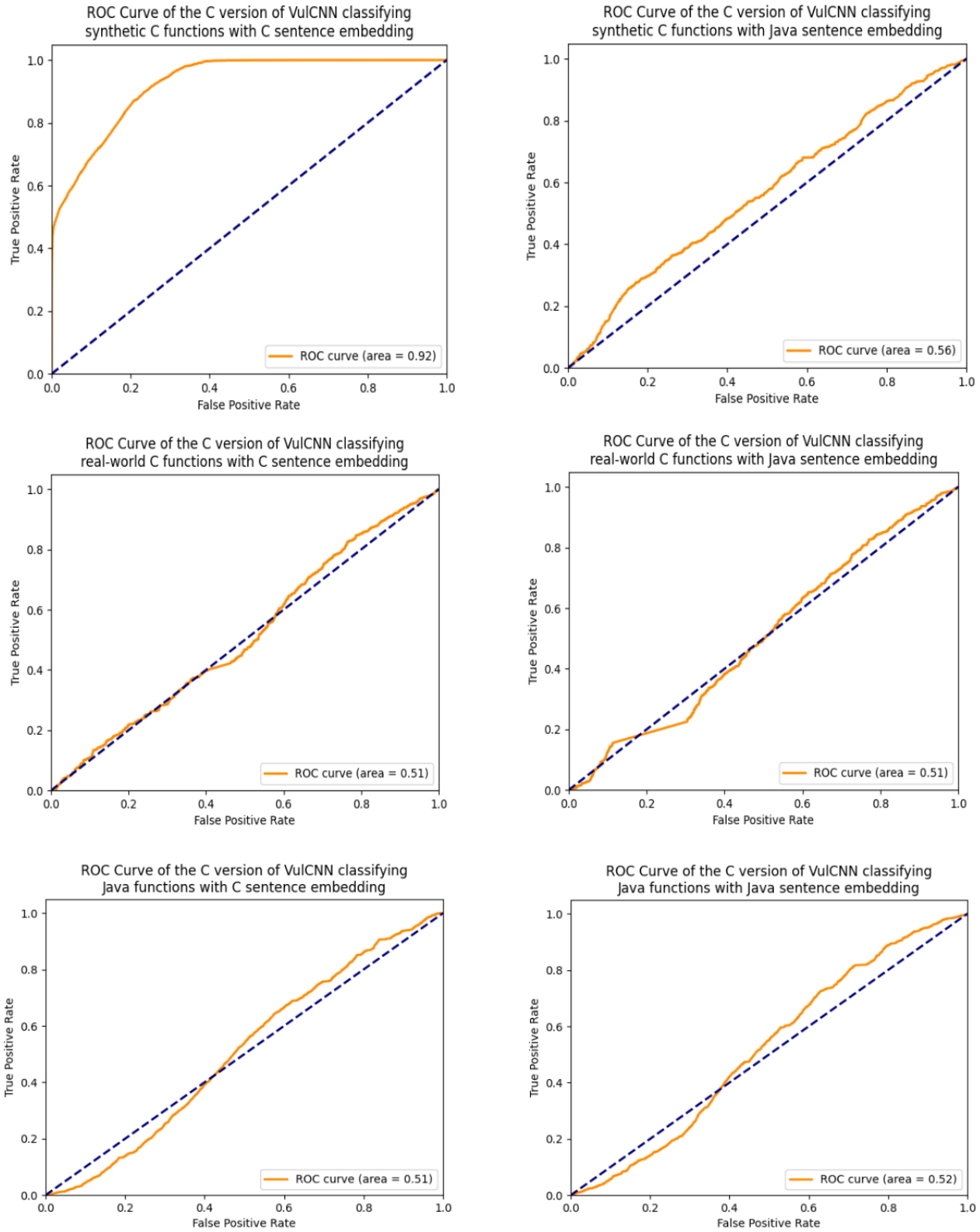

Figure 4.4: ROC curves of all of the aforementioned experiments.

4.4 Training VulCNN for Java

The Java version of VulCNN achieved results very similar to the original version of the model that was made for C. The model was tested on a set of 15.938 functions that were split from the Juliet dataset as a test set. The performance metrics can be seen in table 4.7. Specifically, we see a 5.8% increase in True Positive Rate, a 6.15% decrease in False Positive rate and a 1.3% decrease in accuracy. These results show that the approach of VulCNN can be successfully transferred to different programming languages. Figure 4.5 shows the training and validation loss of the model during the 100 epochs of fitting, while figure 4.2 shows its training and validation accuracy. In both cases, we can see big fluctuations from epoch to epoch in the validation stage, which results to quite anomalous graphs. This is attributed to the fact that the validation data are shuffled differently before each validation phase. Hence, different subsets of the data are used to test the model each epoch, which means that the variability in the performance metrics just represents the performance of the model on these different subsets.

Despite the high performance metrics that the model achieved, it is likely that this model is also unable to generalize vulnerability patterns beyond the synthetic functions that it was trained on. The Juliet dataset is also a part of SARD for Java and is thus similarly structured, with very simplistic and repetitive functions as its entries. It should be noted that other Java datasets like ProjectKB [22] and Vul4J [23] were considered as training datasets for this model but were ultimately excluded due to a very low number of data and lack of structure.

TPR	**TNR**	**Acc**
78.4%	83.6%	82.1%

Table 4.7: Performance metrics of the Java version of VulCNN on functions from Juliet

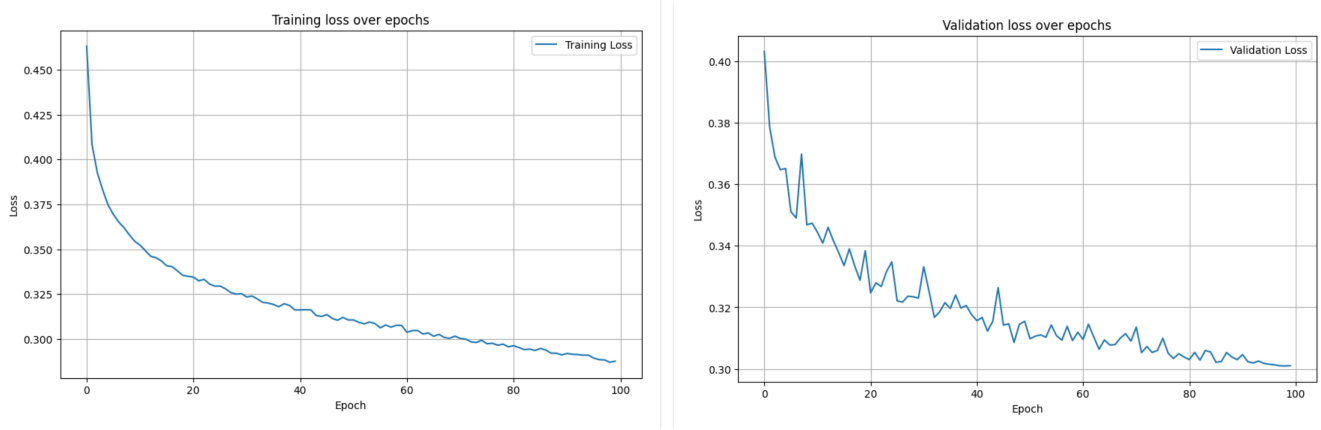

Figure 4.5: (a) Training loss of Java VulCNN over epochs, (b) Validation loss of Java VulCNN over epochs

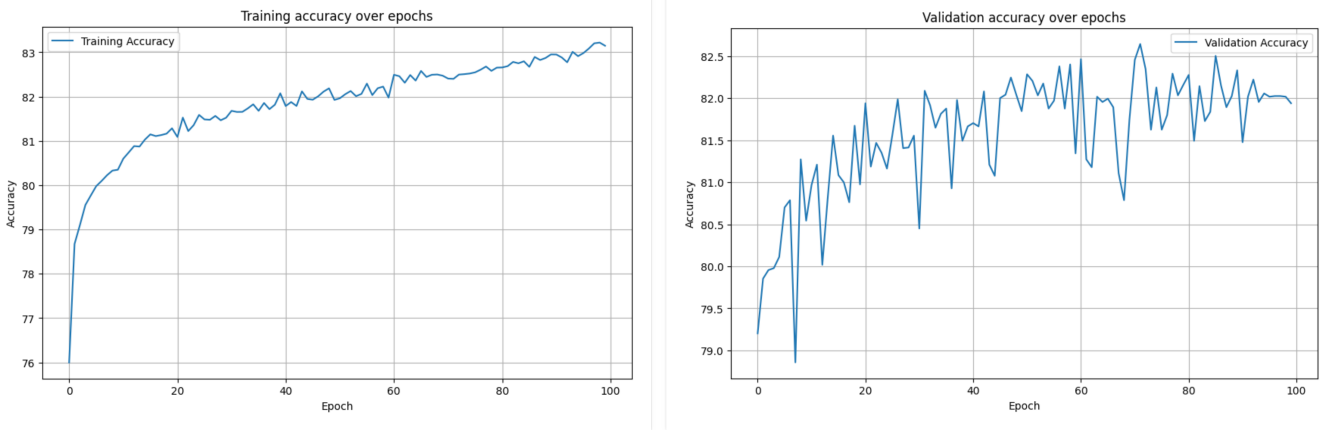

Figure 4.6: (a) Training accuracy of Java VulCNN over epochs, (b) Validation accuracy of Java VulCNN over epochs

5

Threats To Validity

There are four main threats to the validity of the results obtained by the experiments conducted in this paper.

1. Both the C and the Java versions of VulCNN in this paper were trained on synthetic datasets that do not resemble real-world vulnerabilities. Therefore, more promising results on real-world data could be achieved with the use of a more diverse dataset.

2. When normalizing Java source code, a set of Java keywords such as common library/API calls was chosen to be excluded from being normalized. This set of keywords was chosen based on the most common API calls that were found in the Juliet test suite. Because of this, many important keywords that play a crucial role in different vulnerabilities and might be found in real-world software might be redacted, so the vulnerability semantics will be lost during preprocessing.

3. The sentence embedding model that was used to embed java source code was a sent2vec model that was trained by me, using the entries in the Juliet dataset as its training set. The use of a more sophisticated embedding model trained on a larger and more diverse dataset could yield better results.

4. The experiments that were conducted for this research were only run once each, without repetition tests. However, it is likely that repetitions of the same experiments would produce slightly different results due to differences when splitting the dataset to training, validation and testing sets, as well due to small differences in the internal architecture of the models when being retrained.

6

Related Work

Over the years, there has been a lot of research conducted on Deep Learning-based vulnerability detection. The approaches taken on this issue vary widely, both on the utilized DL models and on the feature extraction methods.

In [24], Hussain et al. use a method very similar to VulCNN's, although they enhance it greatly. The researchers utilize a Quantum Convolutional Neural network (QCNN) with self-attentive pooling to classify vulnerable Java functions as well as identify the vulnerability type. To achieve that, they make use of both graph- and token-based techniques. First, the Code Property Graph is parsed and node embedding is applied to it. Afterwards, a hybrid graph neural network is used to accurately extract features from the graph. Additionally, separately from the graph extraction process, codeBERT is used in order to tokenize the source code and extract vulnerability features. The features extracted from the graph- and token-based methods are then classified by the QCNN. This method then differs with VulCNN both because of the more advanced QCNN but also because of the progressive feature extraction methods. It significantly improves past attempts for vulnerability detection with CNN classifiers, as it achieved an impressive accuracy of 99.32%.

Another study conducted by Guo et al. introduces VulExplore, a method for classifying vulnerable functions based on their Code Metrics (CMs) [25]. After the CMs have been generated, they are fed in a CNN + LSTM Neural Network. The CNN layer is responsible for extracting relevant features from the CMs, while the LSTM takes the feature vectors provided by the CNN and creates a deep representation of them that maps their features to vulnerabilities. This approach differs from VulCNN as it combines Convolutional and LSTM layers and also uses Code Metrics instead of PDg analysis to extract relevant features.

Li et al. propose an LSTM apporach for vulnerability detetion, VulDeePecker [26]. VulDeePecker specializes in identifying vulnerabilities that are a result of improper use of library/API calls. After function calls have been extracted from a function, a code gadget is created using the lines of code that contain statements relevant to the arguments of the call. Word2vec embedding is then applied to the code gadget, and a Biderectional LSTM (Bi-LSTM) Neural Network is trained to classify them as vulnerable or not. The same researchers later published μVulDeePecker, a more advanced version of the original approach [27]. It is used to classify whether a function is vulnerable as well as identify the vulnerability type. It uses System Dependence Graph analysis to extract the statements related to the function call and create the code gadgets. From these gadgets, code attentions are generated according to syntax patterns of vulnerabilities, aiding in the categorization of vulnerability type. The gadgets and code attention are turned into vectors, and A Neural network is trained that utilizes three Bi-LSTM models. One is used to learn global features, one to learn local features, and one to merge these features together and make the classification.

Some approaches have deviated from trying to detect vulnerabilities based on source code, and have turned into byte code instead. VulHunter [28] is an approach to DL-based vulnerability detection that first uses control- and data flow analysis to extract relevant code slices, and then turns them to bytecode. The bytecode slices are turned to vectors which are fed in a NN consisting of two Bi-LSTM layers, one merge layer that concatenates their outputs, and one dense layer to classify the slice as vulnerable or not.

In [29], the authors take a similar approach to VulHunter[28], since both methods use bytecode representation to classify vulnerabilities. However, n contrast to VulHunter, this model utilizes an embedding layer for dimension reduction in the opcodes, which are represented as matrices. This embedding projection is then passed to a convolutional layer with pooling, and finally to two fully connected layers to make the classification.

7

Conclusion

This study aimed to evaluate the accuracy of VulCNN and its capacity to generalize vulnerability patterns. It also sought to determine whether a single language-agnostic model can effectively classify functions written in various programming languages while preserving the syntactical semantics of the source code using sentence embedding techniques. Additionally, a new version of VulCNN was trained using the Juliet test suite for Java, achieving accuracy comparable to that of VulCNN. This outcome demonstrates that the approach can be successfully applied across different programming languages, provided their control and data flow can be graphically represented using Program Dependence Graphs.

However, the experiments conducted on the generalizability of sentence embeddings revealed significant limitations in the model's ability to accurately distinguish vulnerability patterns beyond the training data. Because of this, we can not judge certainly whether different sentence embedding techniques can be correctly perceived by the model.In light of these findings, I recommend future research to focus on developing more diverse datasets, encompassing both synthetic and real-world vulnerable source code. It is also essential to invest collective efforts in creating language-agnostic models trained on extensive datasets that include multiple programming languages.

Further research should also explore advanced techniques for extracting features from source code, such as those proposed by Hussain et al [24]. The integration of natural language processing and sequential models such as GRU, LSTM, and Transformers for extracting syntactical features holds promise for significantly improving the accuracy of future models. Lastly, it is crucial to investigate the development of robust cross-language sentence embedding techniques that can preserve semantic information accross multiple languages.

References

[1] Yueming Wu, Deqing Zou, Shihan Dou, Wei Yang, Duo Xu, and Hai Jin. **VulCNN: An Image-inspired Scalable Vulnerability Detection System**. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 2365–2376, 2022. iii, 1, 3, 5

[2] Statista Search Department Ani Petrosyan. **Number of common IT security vulnerabilities and exposures (CVEs) worldwide from 2009 to 2024 YTD**, 2024. 1

[3] Statista Search Department Anna Fleck. **Cybercrime expected to skyrocket: Estimated annual cost of cybercrime worldwide (in trillion U.S. dollars)**, 2023. 1

[4] U.K. National Cyber Security Center. **Vulnerability management**. 1

[5] Fang Wu, Jigang Wang, Jiqiang Liu, and Wei Wang. **Vulnerability detection with deep learning**. In *2017 3rd IEEE International Conference on Computer and Communications (ICCC)*, pages 1298–1302, 2017. 1

[6] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT Press, 11 2016. 3

[7] Baishakhi Ray, Yangruibo Ding, Rahul Krishna, and Saikat Chakraborty. **Deep Learning Based Vulnerability Detection: Are We There yet?** *IEEE Transactions on Software Engineering*, **48**(9):3280–3296, 2022. 4

[8] joern.io. **Joern: The Bug Hunter's Workbench**, January 2024. 4

[9] Matteo Pagliardini, Prakhar Gupta, and Martin Jaggi. **Unsupervised Learning of Sentence Embeddings using Compositional n-Gram Features**.

In *NAACL 2018 - Conference of the North American Chapter of the Association for Computational Linguistics*, 2018. 4, 10

[10] Wikipedia contributors. **Degree centrality: Wikipedia**, 5 2024. 4

[11] Wikipedia contributors. **Closeness centrality: Wikipedia**, 5 2024. 4

[12] Wikipedia contributors. **Katz centrality: Wikipedia**, 5 2024. 5

[13] **libav**, 2024. 7

[14] **The SeaMonkey Project**, 2024. 7

[15] **Xen Project**, 2024. 7

[16] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. **PyTorch: An Imperative Style, High-Performance Deep Learning Library**. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. 9

[17] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. **Exploring network structure, dynamics, and function using NetworkX**. In Gäel Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference (SciPy2008)*, pages 11–15, Pasadena, CA USA, Aug 2008. 9

[18] **PyGraphviz — A Python interface to the Graphviz graph layout and visualization package**. 9

[19] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. **A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries**. In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR '20)*, page 5, New York, NY, USA, May 25–26 2020. ACM. 9, 14, 15

[20] **Juliet Java 1.3 - NIST Software Assurance Reference Dataset**, 2017. 10

[21] **National Institute of Standards and Technology, U.S. Department of Commerce**, 5 2024. 10

[22] Serena E. Ponta, Henrik Plate, Antonino Sabetta, Michele Bezzi, and C´edric Dangremont. **A Manually-Curated Dataset of Fixes to Vulnerabilities of Open-Source Software**. In *Proceedings of the 16th International Conference on Mining Software Repositories*, May 2019. 21

[23] Quang-Cuong Bui, Riccardo Scandariato, and Nicolás E. Díaz Ferreyra. **Vul4J: A Dataset of Reproducible Java Vulnerabilities Geared Towards the Study of Program Repair Techniques**. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, pages 464–468, 2022. 21

[24] Shumaila Hussain, Muhammad Nadeem, Junaid Baber, Mohammed Hamdi, Adel Rajab, Mana Saleh Al Reshan, and Asadullah Shaikh. **Vulnerability Detection in Java Source Code Using a Quantum Convolutional Neural Network with Self-Attentive Pooling, Deep Sequence, and Graph-Based Hybrid Feature Extraction**. *Scientific Reports*, **14**(1):1–17, 2024. 24, 26

[25] J. Guo, Z. Wang, H. Li, and et al. **Detecting Vulnerability in Source Code using CNN and LSTM Network**. *Soft Computing*, **27**:1131–1141, 2023. 24

[26] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. **VulDeePecker: A Deep Learning-Based System for Vulnerability Detection**. Manuscript, 2018. 25

[27] Deqing Zou, Sujuan Wang, Shouhuai Xu, Zhen Li, and Hai Jin. **μVulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection**. *IEEE Transactions on Dependable and Secure Computing*, **18**(5):2224–2236, 2021. 25

[28] Ning Guo, Xiaoyong Li, Hui Yin, and Yali Gao. **VulHunter: An Automated Vulnerability Detection System Based on Deep Learning and Bytecode**. In Jianying Zhou, Xiapu Luo, Qingni Shen, and Zhen Xu, editors, *Information and Communications Security*, pages 199–218, Cham, 2020. Springer International Publishing. 25

REFERENCES

[29] Kaixi Yang, Paul Miller, and Jesus Martinez-Del-Rincon. **Convolutional Neural Network for Software Vulnerability Detection**. In *2022 Cyber Research Conference - Ireland (Cyber-RCI)*, pages 1–4, April 25 2022. 25