MACHINE LEARNING BASED CROSS-LANGUAGE VULNERABILITY DETECTION: HOW FAR ARE WE

by

Anki Chauhan

APPROVED BY SUPERVISORY COMMITTEE:

Wei Yang, Chair

Murat Kantarcioglu

Shuang Hao

This thesis is dedicated to my beloved parents,

for their everlasting support.

MACHINE LEARNING BASED CROSS-LANGUAGE VULNERABILITY DETECTION: HOW FAR ARE WE

by

ANKI CHAUHAN, B.Tech

THESIS

Presented to the Faculty of

The University of Texas at Dallas

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN

COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT DALLAS

May 2020

ACKNOWLEDGMENTS

My foremost gratitude goes to my advisor, Dr. Wei Yang, for his continuous support for my thesis and research work. His guidance has been helpful throughout my research time. My gratitude also extends to my thesis committee, Dr. Murat Kantarcioglu and Dr. Shuang Hao.

March 2020

MACHINE LEARNING BASED CROSS-LANGUAGE VULNERABILITY DETECTION: HOW FAR ARE WE

Anki Chauhan, MS
The University of Texas at Dallas, 2020

Supervising Professor: Wei Yang, Chair

This thesis concerns the study of Machine Learning based methods for detecting vulnerable code. Various Neural Network models have been trained to detect specific vulnerabilities on a programming language dataset. This work, entails an approach not targeting specific vulnerabilities. We also leverage the commonality among programming languages like `JAVA` and `C#` by training the model on both languages and detecting vulnerabilities.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Motivation

Many cyber attacks we see everyday are an exploitation of vulnerabilities in software. These vulnerabilities are not limited to a language or a product, but spread to a wider range.

The Common Vulnerabilities and Exposure[2] list maintained by the Mitre [12] organisation reports a total of 131,700 entries as of 2020. The entries made alone in 2020 are close to 10,210, where as total entries made in 2019 were around 20,974 . These entries span across many languages like JAVA, PHP, Python, Scala and many more.

The automated vulnerability detection programs developed so far fall into either of the two categories: `static` or `dynamic` analysis systems. The `static` analysis tools depend on analysing the source code to apply pattern based (Flawfinder[10], Checkmarx[1], FindBugs[9]), code-similarity based (VUDDY[25], VulPecker[26]) or deep learning algorithms for detecting vulnerabilities (VulDeePecker[27], VulSn[23]). The latter have performed better with lower false negative rates, but the issue with VulDeepPecker or VulSniper is their dataset only focuses on limited vulnerabilities, they focus on 2 CWEs[3], related to resource management and buffer memory only. The `dynamic` analysis tools examine the code executables for vulnerabilities.

In this thesis we focus on the static analysis tools. We propose a model that is trained with more vulnerabilities across 2 different languages, `JAVA` and `C#`. We use a list of 145 `CWEs`[1], to expand the coverage to more vulnerabilities and improve detection rate. We use 2 different datasets, gathered from SARD [18] for `JAVA` and `C#`. We make an attempt to utilize the language similarities between `JAVA` and `C#` and employ the same for detecting vulnerable

[1] "Common Weakness Enumeration is a community-developed list of common software and hardware security weaknesses. It serves as a common language, a measuring stick for security tools, and as a baseline for weakness identification, mitigation, and prevention efforts."[3] More details in section 2.1

code on one language dataset while the model is trained on a different language dataset. We also train a model on both language datasets and provide results for vulnerability detection.

1.2 Contribution

The main contribution of this thesis is in using the existing deep learning techniques over a wider range of CWEs. We train and test our models on a dataset that includes around 145 CWEs (section 2.1). The deep learning models used in this thesis are build using an approach similar to the existing work in VulDeePecker[27] and VulSn[23].

Another contribution is in developing a single model to detect vulnerable code in 2 languages (`JAVA` and `C#`).

The rest of the thesis proceeds as follows: Chapter 2 gives a brief overview of the algorithms used. Chapter 3 details the data collections, setup and experiment results. Chapter 4 concludes the thesis with mention of future work.

CHAPTER 2

BACKGROUND

In this section, we provide background information on CWEs[3] and common vulnerabilities across the 2 programming languages. Overview of the tool used for converting source code to feature vectors. Followed by a brief overview of the models used in the experiments: LSTMs and BiLSTMs. We briefly cover the technique used to balance the experiment dataset.

2.1 Common Vulnerabilities

In this work, we are going to focus on the CWEs[3] published on SARD[18]. Overall we have 145 CWEs in `JAVA` and `C#`, where 112 are tagged with `JAVA` and remaining 33 to `C#`. We analysed the CWEs further to find out the languages shared a total of `7 CWEs`[1] as shown in Table 2.1.

2.2 code2vec

We use code2vec[20] to convert the source code in `JAVA` and `C#` to feature vectors that can be used to train the models in Sections, 2.3 and 2.4.

code2vec is a neural network for learning distributed representations of code, it uses a path-based attention model for processing arbitrary-sized snippets of code into feature vectors [20]. It makes use of Abstract Syntax Tree (AST) to map the code snippet to corresponding feature vectors.

code2vec model produces a feature vector capturing the semantic properties of the code snippet. Their code embedding technique has shown a significant improvement over others as stated in their paper[20].

[1]The common CWEs are the ones which have specific cases present in both, Java and C# datasets.

Table 2.1. Common CWEs[3] from `JAVA` and `C#` datasets.

CWE	Description	Java	C#
CWE-90	"Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection')"	Yes	Yes
CWE-89	"Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')"	Yes	Yes
CWE-78	"Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')"	Yes	Yes
CWE-476	"NULL Pointer Dereference"	Yes	Yes
CWE-209	"Generation of Error Message Containing Sensitive Information"	Yes	Yes
CWE-256	"Unprotected Storage of Credentials"	Yes	Yes
CWE-327	"Use of a Broken or Risky Cryptographic Algorithm"	Yes	Yes

We use 2 code2vec models, one for each language, `JAVA` and `C#`, to convert the source code into feature vectors of fixed length. The details on how we train the 2 models for feature vectors are described in Section 3.1.

2.3 LSTM

The first Neural Network we are going to use is a Long short-term memory(LSTM)[24] model, a Recurrent Neural Network with an adaptive gating mechanism. Since LSTMs do not suffer from Vanishing Gradient problem[21], they become a good choice over other RNNs.

2.4 BiLSTM

LSTMs are limited to forward flow, but in analysing program code many times the value of an argument is often influenced by the following as well as preceding lines of code. In such cases a feed forward network is not enough, hence we make use of Bidirectional LSTM (BiLSTM) model [27] for vulnerability detection.

A BiLSTM layer has a forward layer and a backward layer to capture these bidirectional dependencies.

2.5 SMOTE

As we go into details of the dataset in Section 3.1, we observe as shown in Table 3.1, the number of Vulnerable and Not-Vulnerable instances are highly imbalanced for both programming languages. To avoid biased results, we use oversampling method; Synthetic Minority Oversampling Technique (SMOTE)[22] to synthesise new instances for minority classes.

It is an over-sampling approach which creates "synthetic" examples of the minority class rather than simply using replacement, which duplicates only the original minority samples. SMOTE creates new synthetic examples by taking k minority class nearest neighbors. Value of k is determined by the amount of over-sampling required. If 200% of over-sampling is needed then only two nearest neighbors are chosen randomly and one sample is generated in the each direction. More implementation details can be found in the paper [22].

CHAPTER 3

STUDY METHODOLOGY

In this Section, we describe the initialization steps performed on the data, model configuration and present the experiment results.

3.1 Initialization

For performing the experiments we use 2 Test Suits from SARD[18]. Test Suite for `JAVA` is *Juliet Test Suite for JAVA version 1.3*[16] with 28,881 number of cases included in the suite. For `C#` we use *Test Suite ID 105* [16] with 32,003 cases.

The Test Suits from SARD[18] include both good and bad code for a CWE[3]. The bad code refers to a class/method with a CWE[3], while the good code refers to a class/method with a fix for the same CWE[3].

The `JAVA` test suite, *Juliet Test Suite for JAVA version 1.3*[16] has a couple of test cases for a single CWE[3]. We perform a pre-process data cleanup and extract only the concerned methods for a CWE[3] into individual files, labelling the methods as Vulnerable or not. A method is labelled as Vulnerable if it contains the CWE[3]. A method is labelled as Not-Vulnerable if the code contains a fix for the CWE[3]. All the other method calls like `main()` method call are labelled as *'NA'* and ignored in the training phase.

Figure 3.1 shows one of the vulnerable code samples from the collected instances for "*CWE-256 Unprotected Storage of Credentials*"[4]. The code is vulnerable if the password is stored without any encryption. If the password is encrypted using a encryption algorithm, the instance is secure and is marked as Not-Vulnerable.

A similar approach is applied to the test suit for `C#`. After this process, we can see the breakdown of the input test cases as in Table 3.1. For the time being, we write each case to a separate file in order to make the next step in processing easier.

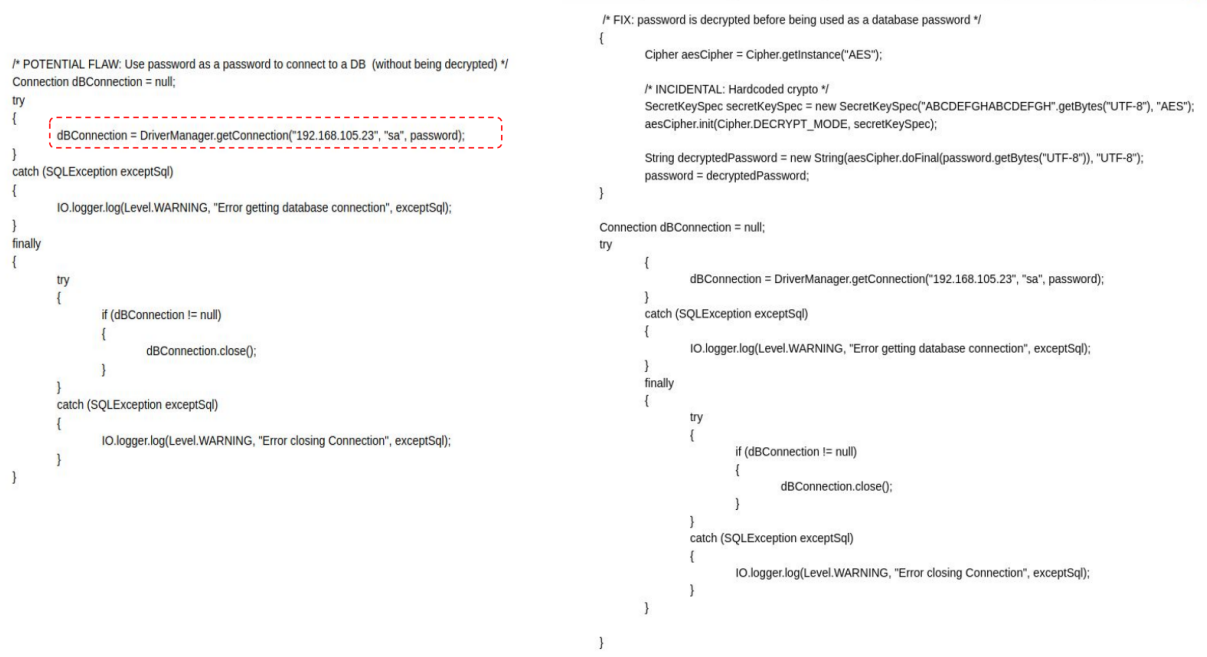

Figure 3.1. A glance at a Vulnerable/Not-Vulnerable code snippet for CWE-256

Now we have a labelled dataset with program code files. We need to convert the code to feature vectors which can be then used to train the neural network models described in Section 3.2. We use *code2vec*[20] tool to convert the `JAVA` and `C#` code to feature vectors. The process is outlined as follows:

For `JAVA` dataset, we use the trained code2vec model[19] for `JAVA` published by the code2vec team, which has been trained on 14 million `JAVA` files. The configuration for the code2vec model can be found here[11]. The code2vec model is run on each individual file in our labelled dataset producing a feature vector of length 384(as per the configuration) for each file. We then combine all the feature vectors with their labels, Vulnerable and Not-Vulnerable into a single *CSV* file.

For `C#` dataset, we train another code2vec model with similar configuration[11] on the dataset. We run multiple epochs and save the model with best F1 score. We then run use the saved model on each file in our labelled dataset, similar to the process used for `JAVA`.

Table 3.1. Distribution of code instances collected from SARD[18]

Programming Language	Total input instances	Vulnerable instances	Not Vulnerable instances
JAVA	123,080	32,734	90,346
C#	30,655	18,137	12,518

At the end of this step, we have 2 *CSV* files, one for `JAVA` and one for `C#` with feature vectors of size, 384.

3.2 Neural Network Configuration

In this Section we describe the neural network models and their configuration for the experiments.

First model we use for training is an LSTM (Section 2.3) model. The model is designed with `6` hidden layers with hidden input sizes of `256, 256, 128, 128, 64 and 64` respectively.

The corresponding layers of the LSTM model can be seen in Figure 3.2. We use *Dense* and *Flatten* layers intermittently through out the network to manage dimensions while passing features from one LSTM layer to another. The configuration summary of the model is seen in Figure 3.3.

Second model used for training is a BiLSTM(Section 2.4) model. The model is designed with `3` hidden layers with hidden input sizes of `64, 32 and 32` respectively.

The corresponding layers of the BiLSTM model can be seen in Figure 3.4. We use *Dense* and *Flatten* layers intermittently through out the network here as well, to manage dimensions while passing features from one BiLSTM layer to another. The configuration summary of the model is seen in Figure 3.5.

The hyerparameters can be glanced at in Table 3.2. We use `50` epochs for each model, with a batch size defined at `64`. The dropout for each layer is set at `0.2`. For BiLSTM layers

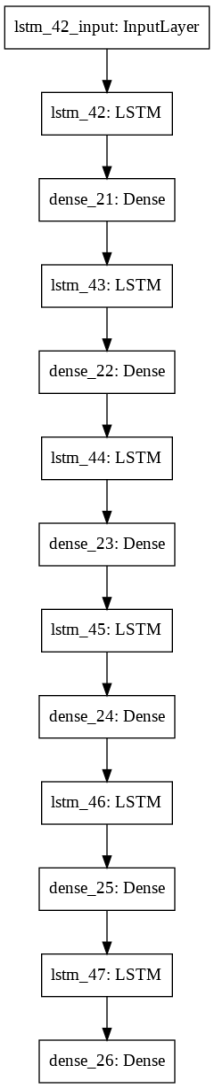

Figure 3.2. LSTM model

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_42 (LSTM)               (None, 1, 256)            656384
_________________________________________________________________
dense_21 (Dense)             (None, 1, 256)            65792
_________________________________________________________________
lstm_43 (LSTM)               (None, 1, 256)            525312
_________________________________________________________________
dense_22 (Dense)             (None, 1, 256)            65792
_________________________________________________________________
lstm_44 (LSTM)               (None, 1, 128)            197120
_________________________________________________________________
dense_23 (Dense)             (None, 1, 128)            16512
_________________________________________________________________
lstm_45 (LSTM)               (None, 1, 128)            131584
_________________________________________________________________
dense_24 (Dense)             (None, 1, 64)             8256
_________________________________________________________________
lstm_46 (LSTM)               (None, 1, 64)             33024
_________________________________________________________________
dense_25 (Dense)             (None, 1, 64)             4160
_________________________________________________________________
lstm_47 (LSTM)               (None, 1, 64)             33024
_________________________________________________________________
dense_26 (Dense)             (None, 1, 1)              65
=================================================================
Total params: 1,737,025
Trainable params: 1,737,025
Non-trainable params: 0
```

Figure 3.3. LSTM model with trainable parameters

Table 3.2. Hyperparameters for our models

Name	**Value**
Learning Rate	0.001
Number of Epochs	50
Batch size	64
Dropout	0.2
Recurrent Dropout	0.2

the activation function used is ReLU[15], while the final activation layer for both models uses Sigmoid function[17].

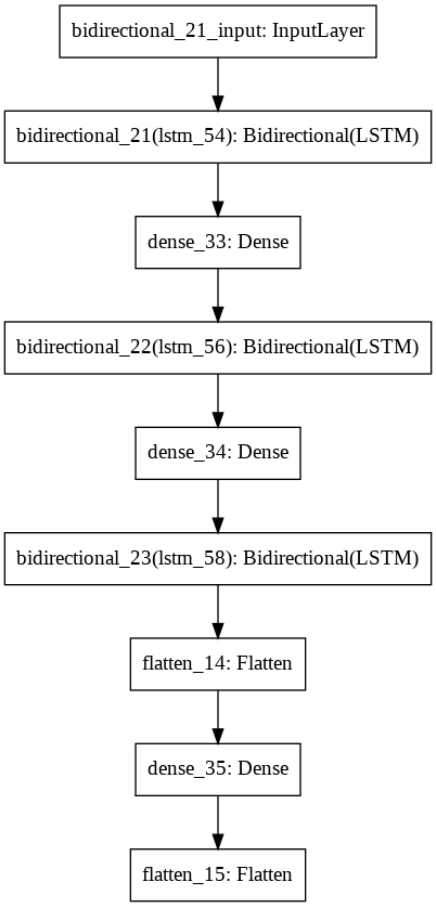

Figure 3.4. BiLSTM model

3.3 Results

All parts of the experiment were conducted using Google Collab using Intel(R) Xeon(R) CPU @ 2.30GHz with 25 GB of RAM. In these experiments, we split the data to training, validation and test sets randomly with the ratio of `54%, 13% and 33%` ,respectively.

We present the results in 3 Sections: Section 3.3.1 presents the results of experiments on the dataset for `JAVA` collected in section 3.1. Section 3.3.2 presents the results for the `C#` dataset. The last section 3.3.3 presents the results of the models using both `JAVA` and `c#` datasets.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
bidirectional_21 (Bidirectio (None, 1, 128)            229888
_________________________________________________________________
dense_33 (Dense)             (None, 1, 64)             8256
_________________________________________________________________
bidirectional_22 (Bidirectio (None, 1, 128)            66048
_________________________________________________________________
dense_34 (Dense)             (None, 1, 32)             4128
_________________________________________________________________
bidirectional_23 (Bidirectio (None, 1, 64)             16640
_________________________________________________________________
flatten_14 (Flatten)         (None, 64)                0
_________________________________________________________________
dense_35 (Dense)             (None, 1)                 65
_________________________________________________________________
flatten_15 (Flatten)         (None, 1)                 0
=================================================================
Total params: 325,025
Trainable params: 325,025
Non-trainable params: 0
_________________________________________________________________
```

Figure 3.5. BiLSTM model with trainable parameters

3.3.1 Results on JAVA dataset

We run the training instances first through SMOTE as described in Section 2.5 to balance the dataset distribution across the 2 categories. After resampling with SMOTE, we have `96,952` for training (65,970 before resampling), `16,493` for validation and `40,617` instances for testing.

The results for the LSTM and BiLSTM model as described in Section 3.2, are shared in Table 3.4. We can see the accuracy for both models is the same while the F1-score of the LSTM model is better by **.01**. We did further analysis to include the Area Under the Curve (AUC) for the Precision-Recall (PR) curve [13] and the Receiver operating characteristic (ROC) curve [14]. Since the dataset shows a balanced number of instances for both classes as shown in Table 3.3 after resampling with SMOTE, the ROC curve is better suited for our dataset. The Table 3.5 shows the BiLSTM model performs better than the LSTM model.

For further analysis, we examine the ROC curve in Figure 3.6, which plots the False Positive Rate against the True Positive Rate. We can see the True Positive Rate reaches

Table 3.3. Distribution of training instances after resampling.

Programming Language	**Total training instances**	**Vulnerable instances**	**Not Vulnerable instances**
JAVA	96,952	48,476	48,476
C#	19,410	9,705	9,705

Table 3.4. Results on `JAVA` dataset

Model	**Accuracy**	**Precision**	**Recall**	**F1 score**
LSTM	0.75	0.75	0.82	**0.74**
BiLSTM	0.75	0.74	0.81	**0.73**

Table 3.5. AUC Results on `JAVA` dataset

Model	**AUC for ROC Curve**	**AUC for PR curve**
LSTM	**0.85**	0.60
BiLSTM	**0.87**	0.68

Table 3.6. Results on `JAVA` dataset from FindBugs

Tool	**Accuracy**	**Total Classes**	**Classes identified with Vulnerability**
FindBugs	0.23	47319	11210

the highest with a lower False Positive Rate at 0.4, indicating the model is able to identify vulnerable code accurately without too many false positives.

We ran the `JAVA` dataset with a static code analysis tool, FindBugs[9]. The results of the run can be seen in Table 3.6. The FindBugs[9] tool was able to identify *11210* classes with bugs out of total, *47319*.

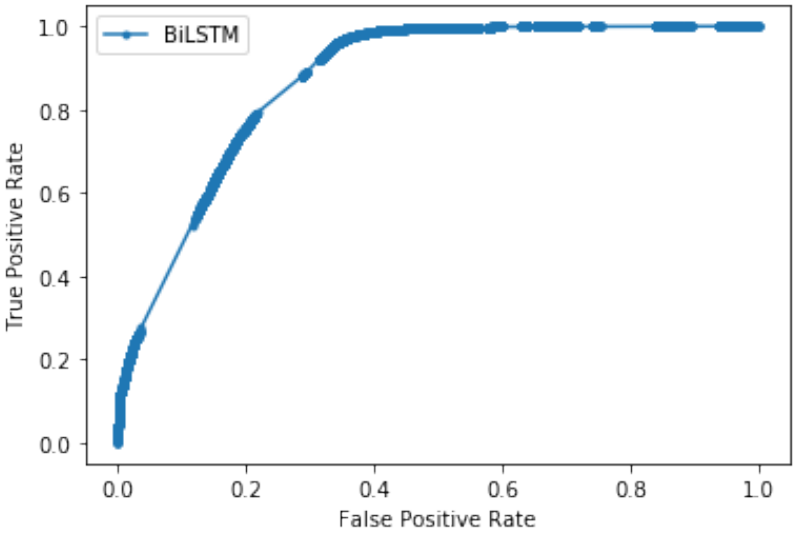

Figure 3.6. ROC curve for BiLSTM model

Table 3.7. Results on `C#` dataset

Model	Accuracy	Precision	Recall	F1 score
LSTM	0.54	0.62	0.59	**0.53**
BiLSTM	0.7	0.7	0.71	**0.69**

3.3.2 Results on C# dataset

We run the training instances first through SMOTE again. After SMOTE, we have `19,410` for training (16,430 before), `4,108` for validation and `10,117` instances for testing.

The results for the LSTM and BiLSTM model as described in Section 3.2, are shared in Table 3.7. We can see the BiLSTM model outperforms the LSTM model as shown in Table 3.7. Since the dataset shows a balanced number of instances for both classes for `C#` as shown in Table 3.3 after resampling with SMOTE, the ROC curve is better suited for our dataset. The Table 3.8 shows the BiLSTM model performs better than the LSTM model.

For further analysis, we examine the ROC curve in Figure 3.7, which plots the False Positive Rate against the True Positive Rate. We can see the True Positive Rate reaches

Table 3.8. AUC Results on C# dataset

Model	AUC for ROC Curve	AUC for PR curve
LSTM	**0.57**	0.7
BiLSTM	**0.81**	0.83

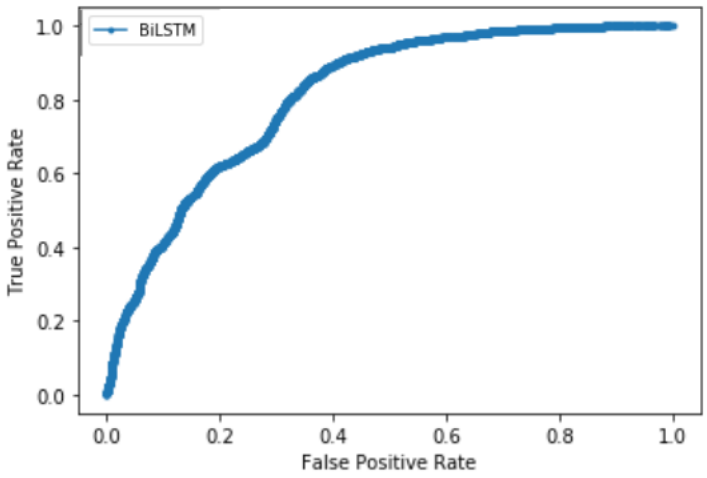

Figure 3.7. ROC curve for BiLSTM model

the highest with a lower False Positive Rate at 0.5, indicating the model is able to identify vulnerable code accurately without too many false positives.

3.3.3 Results on JAVA and C# datasets

We further analysed the trained BiLSTM model from Section 3.3.2 by testing it on the JAVA test dataset with 40,617 instances as explained in Section 3.3.1.

The Table 3.9 shows the results of the run, we see the model trained on C# train instances is able to to reach an accuracy of **66%**.

We take a deeper look at the CWEs[3] the model was able to identify. Tables 3.10 and 3.11 shows the detailed listing of the CWEs[3] identified by the model. The model trained on C# instances is able to detect 71 CWEs out of 112 in JAVA discussed in Section 2.1.

Table 3.9. Results for C# model on JAVA dataset

Model	Accuracy	Precision	Recall	F1 score
BiLSTM	**0.66**	0.54	0.54	0.55

Table 3.10. Overview of the detected common CWEs on JAVA dataset

CWE	Description	Java	C#
CWE-90	"Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection')"	Yes	Yes
CWE-89	"Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')"	Yes	Yes
CWE-78	"Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')"	Yes	Yes
CWE-476	"NULL Pointer Dereference"	Yes	Yes
CWE-209	"Generation of Error Message Containing Sensitive Information"	Yes	Yes
CWE-256	"Unprotected Storage of Credentials"	Yes	Yes

Table 3.10 lists the 6 common CWEs[3] the model was able to detect when run on test instances from JAVA out of the 7 listed in Table 2.1.

Table 3.11 lists the 65 CWEs[3] which were not present in the C# training set but the model was able to detect them on the JAVA test instances. On further analysis, we found out that few of the CWEs in Table 3.11 can be traced back to common CWEs in Table 3.10. For example, the "*CWE-549 Missing Password Field Masking*"[7], can be linked to "*CWE-256 Unprotected Storage of Credentials*"[4], via an inter-dependency relationship which tags both CWEs under "*CWE-522: Insufficiently Protected Credentials*"[6]. The "*CWE-690: Unchecked Return Value to NULL Pointer Dereference*"[8] is also linked to the common "*CWE-476: NULL Pointer Dereference*[5]"".

Taking inspiration from the findings of the last experiment, we trained a model with both JAVA and C# instances. Followed by testing the model on all of JAVA and C# original instances, the results can be seen in Table 3.12

Table 3.11. Overview of the detected CWEs on `JAVA` dataset not reported in `C#`

CWE	Description	Java	C#
CWE-129	"Improper Validation of Array Index"	Yes	No
CWE-190	"Integer Overflow or Wraparound"	Yes	No
CWE-369	"Divide By Zero"	Yes	No
CWE-191	"Integer Underflow (Wrap or Wraparound)"	Yes	No
CWE-606	"Unchecked Input for Loop Condition"	Yes	No
CWE-396	"Declaration of Catch for Generic Exception"	Yes	No
CWE-789	"Uncontrolled Memory Allocation"	Yes	No
CWE-400	"Uncontrolled Resource Consumption"	Yes	No
CWE-197	"Numeric Truncation Error"	Yes	No
CWE-23	"Relative Path Traversal"	Yes	No
CWE-113	"Improper Neutralization of CRLF Sequences in HTTP Headers ('HTTP Response Splitting')"	Yes	No
CWE-643	"Improper Neutralization of Data within XPath Expressions ('XPath Injection')"	Yes	No
CWE-319	"Cleartext Transmission of Sensitive Informatio"n	Yes	No
CWE-470	"Use of Externally-Controlled Input to Select Classes or Code ('Unsafe Reflection')"	Yes	No
CWE-398	"Code Quality"	Yes	No
CWE-83	"Improper Neutralization of Script in Attributes in a Web Page"	Yes	No
CWE-134	"Use of Externally-Controlled Format String"	Yes	No
CWE-36	"Absolute Path Traversal"	Yes	No
CWE-533	"Information Exposure Through Server Log File"s	Yes	No
CWE-80	"Improper Neutralization of Script-Related HTML Tags in a Web Page (Basic XSS)"	Yes	No
CWE-563	"Assignment to Variable without Use"	Yes	No
CWE-259	"Use of Hard-coded Password"	Yes	No
CWE-835	"Loop with Unreachable Exit Condition ('Infinite Loop')"	Yes	No
CWE-690	'Unchecked Return Value to NULL Pointer Dereference"	Yes	No
CWE-325	"Missing Required Cryptographic Step"	Yes	No
CWE-81	"Improper Neutralization of Script in an Error Message Web Page"	Yes	No
CWE-506	"Embedded Malicious Code"	Yes	No
CWE-549	"Missing Password Field Masking"	Yes	No
CWE-601	"URL Redirection to Untrusted Site ('Open Redirect')"	Yes	No
CWE-597	"Use of Wrong Operator in String Comparison"	Yes	No
CWE-15	"External Control of System or Configuration Setting"	Yes	No

Table 3.11 continued

CWE	Description	Java	C#
CWE-600	"Uncaught Exception in Servlet"	Yes	No
CWE-681	"Incorrect Conversion between Numeric Types"	Yes	No
CWE-523	"Unprotected Transport of Credentials"	Yes	No
CWE-510	"Trapdoor"	Yes	No
CWE-477	"Use of Obsolete Function"	Yes	No
CWE-328	"Reversible One-Way Hash"	Yes	No
CWE-378	"Creation of Temporary File With Insecure Permissions"	Yes	No
CWE-613	"Insufficient Session Expiration"	Yes	No
CWE-511	"Logic/Time Bomb"	Yes	No
CWE-605	"Multiple Binds to the Same Port"	Yes	No
CWE-226	"Sensitive Information Uncleared in Resource Before Release for Reuse"	Yes	No
CWE-193	"Off-by-one Error"	Yes	No
CWE-535	"Exposure of Information Through Shell Error Message"	Yes	No
CWE-586	"Explicit Call to Finalize()"	Yes	No
CWE-459	"Incomplete Cleanup"	Yes	No
CWE-546	"Suspicious Comment"	Yes	No
CWE-483	"Incorrect Block Delimitation"	Yes	No
CWE-481	"Assigning instead of Comparing"	Yes	No
CWE-534	"Information Exposure Through Debug Log Files"	Yes	No
CWE-252	"Unchecked Return Value"	Yes	No
CWE-390	"Detection of Error Condition Without Action"	Yes	No
CWE-486	"Comparison of Classes by Name"	Yes	No
CWE-478	"Missing Default Case in Switch Statement"	Yes	No
CWE-338	"Use of Cryptographically Weak Pseudo-Random Number Generator (PRNG)"	Yes	No
CWE-598	"Use of GET Request Method With Sensitive Query Strings"	Yes	No
CWE-617	"Reachable Assertion"	Yes	No
CWE-397	"Declaration of Throws for Generic Exception"	Yes	No
CWE-315	"Cleartext Storage of Sensitive Information in a Cookie"	Yes	No
CWE-253	"Incorrect Check of Function Return Value"	Yes	No
CWE-609	"Double-Checked Locking"	Yes	No
CWE-321	"Use of Hard-coded Cryptographic Key"	Yes	No
CWE-566	"Authorization Bypass Through User-Controlled SQL Primary Key"	Yes	No
CWE-114	"Process Control"	Yes	No
CWE-759	"Use of a One-Way Hash without a Salt"	Yes	No

Table 3.12. Results for model trained on `C# and JAVA` dataset

Model	Language	Accuracy	Precision	Recall	F1 score
LSTM	JAVA	**0.79**	0.75	0.8	0.76
LSTM	C#	**0.59**	0.3	0.5	0.37
BiLSTM	JAVA	**0.74**	0.74	0.81	0.72
BiLSTM	C#	**0.59**	0.3	0.5	0.37

Table 3.13. AUC Results on `C# and JAVA` dataset

Model	Language	AUC for ROC Curve	AUC for PR curve
LSTM	JAVA	0.89	0.72
LSTM	C#	0.69	0.74
BiLSTM	JAVA	0.88	0.7
BiLSTM	C#	0.69	0.74

We further look at the PR curves for both languages for this model, since we did not use SMOTE for oversampling and balancing the dataset, while performing the tests, we will go with the Precision-Recall curve for a better analysis of the model performance. We observe the common model performs significantly better on the `JAVA` instances than `C#`, which is also evident in the AUC curves in Figure 3.8 and Figure 3.9.

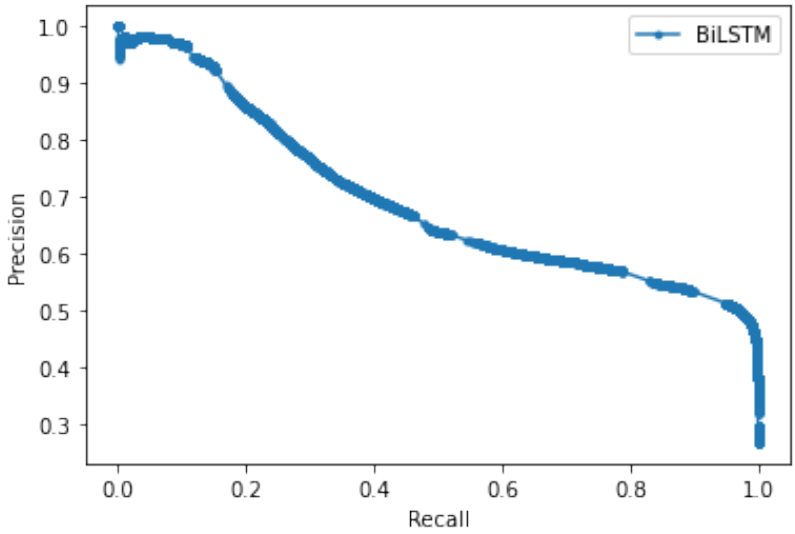

Figure 3.8. PR curve for common BiLSTM model on `JAVA` instances

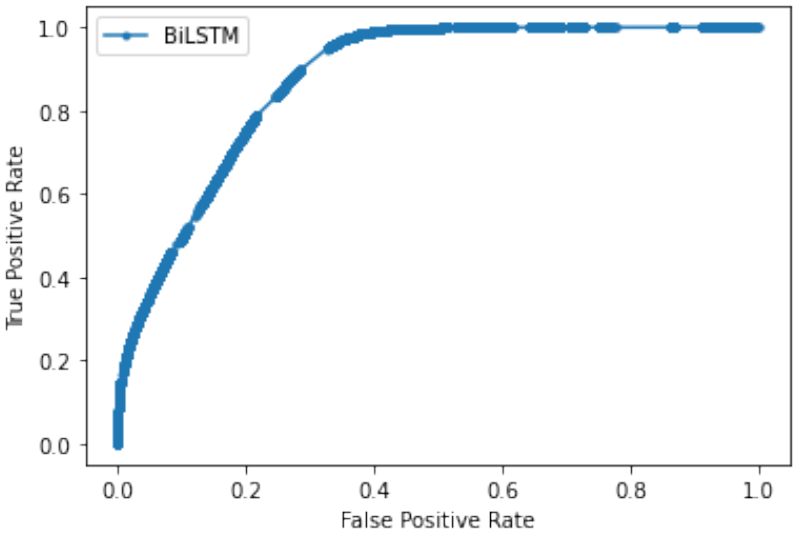

Figure 3.9. ROC curve for common BiLSTM model on `JAVA` instances

CHAPTER 4

CONCLUSION AND FUTURE WORK

This work shows how a single model can be used to detect vulnerabilities in programming languages covering more CWEs. We also show how we can make use of programming language commonalities to detect vulnerabilities in code and build a single model for `JAVA` and `C#` . To our knowledge this is the first attempt of this kind, in exploring vulnerabilities across programming languages.

In future, we plan to improve the models for better performance and include more languages like C,C++,Scala, Python for vulnerability detection.

REFERENCES

[1] *Checkmarx*. https://www.checkmarx.com/.

[2] *Common Vulnerabilities and Exposures*. https://cve.mitre.org/.

[3] Common weakness enumeration. https://cwe.mitre.org/.

[4] *CWE-256: Unprotected Storage of Credentials)*. https://cwe.mitre.org/data/definitions/256.html.

[5] *CWE-476: NULL Pointer Dereference*. https://cwe.mitre.org/data/definitions/476.html.

[6] *CWE-522: Insufficiently Protected Credentials*. https://cwe.mitre.org/data/definitions/522.html.

[7] *CWE-549: Missing Password Field Masking*. https://cwe.mitre.org/data/definitions/549.html.

[8] *CWE-690: Unchecked Return Value to NULL Pointer Dereference*. https://cwe.mitre.org/data/definitions/690.html.

[9] *FindBugs*. http://findbugs.sourceforge.net/factSheet.html.

[10] *Flawfinder*. https://dwheeler.com/flawfinder/.

[11] *Github page for code2vec*. https://github.com/tech-srl/code2vec.

[12] *The MITRE Corporation*. https://www.mitre.org/.

[13] *Precision and recall*. https://en.wikipedia.org/wiki/Precision_and_recall.

[14] *Receiver operating characteristic*. https://en.wikipedia.org/wiki/Receiver_operating_characteristic.

[15] *Rectifier (neural networks)*. https://en.wikipedia.org/wiki/Rectifier_(neural_networks).

[16] *SARD Resources*. https://samate.nist.gov/SARD/testsuite.php.

[17] *Sigmoid function*. https://en.wikipedia.org/wiki/Sigmoid_function.

[18] *Software Assurance Reference Dataset Project*. https://samate.nist.gov/SARD/.

[19] *Trained model for Java shared by code2vec team.* `https://s3.amazonaws.com/code2vec/model/java14m_model.tar.gz`.

[20] Alon, U., M. Zilberstein, O. Levy, and E. Yahav (2019, January). Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang. 3*(POPL), 40:1–40:29.

[21] Bengio, Y. (2009, January). Learning deep architectures for ai. *Found. Trends Mach. Learn. 2*(1), 1–127.

[22] Bowyer, K. W., N. V. Chawla, L. O. Hall, and W. P. Kegelmeyer (2011). SMOTE: synthetic minority over-sampling technique. *CoRR abs/1106.1813*.

[23] Duan, X., J. Wu, S. Ji, Z. Rui, T. Luo, M. Yang, and Y. Wu (2019, 7). Vulsniper: Focus your attention to shoot fine-grained vulnerabilities. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, pp. 4665–4671. International Joint Conferences on Artificial Intelligence Organization.

[24] Hochreiter, S. and J. Schmidhuber (1997, 12). Long short-term memory. *Neural computation 9*, 1735–80.

[25] Kim, S., S. Woo, H. Lee, and H. Oh (2017, May). Vuddy: A scalable approach for vulnerable code clone discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*, pp. 595–614.

[26] Li, Z., D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu (2016). Vulpecker: An automated vulnerability detection system based on code similarity analysis. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, ACSAC '16, New York, NY, USA, pp. 201–213. Association for Computing Machinery.

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

BIOGRAPHICAL SKETCH

Anki Chauhan began learning about cyber security with machine learning in Spring 2019 under Dr. Wei Yang. She has been involved with several projects concerning malware detection, vulnerability detection and attacks on adversarial neural networks with Dr. Wei Yang and his lab.

CURRICULUM VITAE

ANKI CHAUHAN
Dallas, TX 75252
http://www.linkedin.com/in/ankiC | http://ankichauhan.com
axc170043@utdallas.edu , anki54@gmail.com | +1 972 834 1399

Education

THE UNIVERSITY OF TEXAS AT DALLAS, Richardson, Texas — Expected
Master of Science in Computer Science,**GPA 3.66/4** — May 2020
Related Courses: Machine Learning, Cyber Security(ML), BigData Analytics, CNNs, NLP, Design and Analysis of Algorithms

Skills

Languages	: Java 1.8, Python 3, Scala, UNIX Shell scripts, SQL, Prolog, Lisp,Solidity
Frameworks/Tools	: Spring Boot 2.1.x, Spring 4.x, Hibernate, JOOQ, REST, Maven, Splunk, Jenkins, Groovy, BigQuery, Octave/Matlab, PyTorch, TensorFlow, Hadoop, Spark, Kafka
Cloud	: Cloud Foundry, Heroku , Google Cloud Platform
Web	: HTML, CSS, JavaScript, AngularJS, JQuery, Ext JS
Databases	: PostgreSQL, Oracle, mssql

Publications

Mirazul Haque, Anki Chauhan, Cong Liu, Wei Yang. ILFO: Adversarial Attack on Adaptive Neural Networks. In *CVPR*, 2020.
Paper on attacking energy consumption of Deep Neural Networks using Adversarial Learning.

Achievements

Ranked among the top 500 in Google Hashcode 2020 Online Qualification Round in the United States.

Certifications

Neural Networks and Deep Learning by deeplearning.ai

Academic Projects

Common Vulnerability detection using Deep Learning, Master's Thesis — May 2019 - Apr 2020
Building a Deep Learning model using BiLSTMs on SARD dataset for Java and C# to automatically detect vulnerable code.

Work Experience

PeopleSoft Security Analyst, University of Texas at Dallas — Jan 2019 - Jul 2019
Dallas, USA

- Analyze and perform PeopleSoft account access administration using tools like SQL and PSQuery.

Associate of Corporate and Investment Banking, JP Morgan and Chase — Sep 2017 - Jun 2018
Bangalore, India

- On boarded Spring 3.x applications for Collateral Triparty Management to an in house Application Monitoring on the cloud (Cloud Foundry). Designed a quick build system with Jenkins that saved time during releases.

Senior Associate Platform Level 1, Sapient Consulting Private Limited — Nov 2014 - Sep 2017
Bangalore, India

- Build an API using Spring Boot 2.x and REST APIs that made it a golden source of data across the firm.
- Built strong integration test suits to prevent potential applications breaks using Mockito, H2, Java and Python.

System Analyst, Nihilent Technologies — Mar 2013 - Nov 2014
Pune, India

- Implemented payment module for healthcare project My Medical Records using Intuit API.

Software Developer, CGI Information System and Management Consulting Pvt. LTD. — July 2010 - Mar 2013
Mumbai, India

- Designed and developed integration modules for interfaces based on EIP with Spring integration.

Activities

Young Women in Science and Engineering Investigators, College Mentor — Aug 2018 - Apr 2020