Detecting Software Vulnerabilities Using Neural Networks

Amy Aumpansub
School of Computing
DePaul University
Chicago, IL, USA
aaumpans@depaul.edu

Zhen Huang
School of Computing
DePaul University
Chicago, IL, USA
zhen.huang@depaul.edu

ABSTRACT

As software vulnerabilities remain prevalent, automatically detecting software vulnerabilities is crucial for software security. Recently neural networks have been shown to be a promising tool in detecting software vulnerabilities. In this paper, we use neural networks trained with program slices, which extract the syntax and semantic characteristics of the source code of programs, to detect software vulnerabilities in C/C++ programs. To achieve a strong prediction model, we combine different types of program slices and optimize different types of neural networks. Our result shows that combining different types of characteristics of source code and using a balanced ratio of vulnerable program slices and non-vulnerable program slices a balanced accuracy in predicting both vulnerable code and non-vulnerable code. Among different neural networks, BGRU performs the best in detecting software vulnerabilities with an accuracy of 94.89%.

CCS CONCEPTS

• **Security and privacy → Software and application security**; • **Computing methodologies → Neural networks**.

KEYWORDS

vulnerability detection, software vulnerability, deep learning, neural network

ACM Reference Format:
Amy Aumpansub and Zhen Huang. 2021. Detecting Software Vulnerabilities Using Neural Networks. In *2021 13th International Conference on Machine Learning and Computing (ICMLC '21), February 26-March 1, 2021, Shenzhen, China.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3457682.3457707

1 INTRODUCTION

Software vulnerabilities is a severe threat to network and information security. As hackers and malware frequently exploit software vulnerabilities to compromise computer systems, popular software vendors offer as much as $1 million dollars reward to individuals who report software vulnerabilities [1–4].

For decades, a large number of studies have been contributed to detecting vulnerabilities at the source code level [6, 8, 12, 16, 23, 25]. Mainly they are based on code similarity detection or pattern matching. Unfortunately code similarity detection is not well-suited for detecting vulnerabilities not caused by code cloning, while pattern matching requires human experts to define patterns that represent vulnerabilities.

To address these limitations, neural networks have been used recently to detect vulnerabilities [7, 10, 18, 20, 21, 26]. The neural networks have been widely used in image processing and speech recognition as it can provide the high accuracy rate of prediction while insignificantly relying on human experts in feature extraction. As software vulnerabilities can be caused by a wide range of reasons, the neural networks can be served as a useful tool in detecting vulnerabilities. Unlike pattern-based methods, the neural network eliminates influence of human bias in feature extractions.

A main focus of this paper is to develop a strong predictive model using neural networks for detecting the vulnerability of C/C++ programs automatically. Unlike previous work that generates individual models from different types of characteristics extracted from the source code of the programs [13], we build the model on a dataset that combines different types of characteristics extracted from source code. We find that the model built from the combined dataset outperforms the individual models fitted with individual dataset.

To develop a strong predictive model, we optimize the neural networks with different hyperparameters such as optimizer, gating mechanism, and activation functions in our model development. Our results show that the BGRU model's accuracy rate is as high as 94.6% for a training set and 92.4% for a test set.

The major contributions of this paper is as follows:

- We show that the accuracy of the model built on the combined dataset of program slices surpasses the models built on individual dataset.
- By balancing the ratio of vulnerable program slices (class 1) and non-vulnerable program slices (class 0), the model performs well with a high balanced accuracy rate of 93% which is comparable to that of a training set. The high sensitivity and specificity imply the model has a good ability in explaining both vulnerability and non-vulnerability classes.
- We compare different types of neural networks and show that BGRU performs the best.
- The model built with BGRU achieves an accuracy rate of 94.89% by utilizing 10X more program slices.
- We have implemented a chain of tools for generating the model from program slices and open sourced the tools at https://gitlab.com/vulnerability_analysis/vulnerability_detection/.

ICMLC '21, February 26-March 1, 2021, Shenzhen, China

ACM ISBN 978-1-4503-8931-0/21/02…$15.00
https://doi.org/10.1145/3457682.3457707

2 BACKGROUND

We use the dataset of C/C++ programs collected by Zhen Li, et al. [13]. It contains 1,592 programs from the National Vulnerability Database (NVD) and 14,000 programs from the Software Assurance Reference Dataset (SARD). The programs were pre-processed and transformed to 420,627 slices called semantic vulnerability candidates (SeVC) which include 56,395 vulnerable slices (13.5 % of total slices) and 364,232 non-vulnerable slices (86.5 % of total slices).

The slices of programs were divided into the following four main types regarding the vulnerability syntax analysis obtained from Checkmarx:

- **Library or API Function Call (FC).** This vulnerability type is associated with library or API functions calls in program which contain 811 C/C++ library/API function calls. This type represents 15.3% of total sample slices, comprising 13,603 vulnerable slices and 50,800 non-vulnerable slices.
- **Array Usage (AU).** This vulnerability is related to the use of arrays such as improper uses of array element access, array, and arithmetic, accounting for 10% of total slices which contains 10,926 vulnerable slices and 31,303 non-vulnerable slices.
- **Pointer Usage (PU).** This vulnerability type is correlated to inappropriate uses of pointer arithmetic and references which are a main type of sample slices comprising 69.4% of total slices which includes 28,391 vulnerable slices and 263,450 non-vulnerable slices.
- **Arithmetic Expression (AE).** This vulnerability type is related to improper arithmetic expressions such as integer overflow which represents 5.3% of total slices, comprising 3,475 vulnerable slices and 18,679 non-vulnerable slices.

2.1 Generating Program Slices

The program slices are generated in a two-phase process. First, Syntax-based Vulnerability Candidates (SyVCs) are extracted from Program Dependency Graph for each function of the C/C++ programs. Each SyVC embodies syntax characteristics of a vulnerability. Second, Semantics-based Vulnerability Candidates (SeVCs) are produced from SyVCs. Each SeVC extends a SyVC with data dependency and control dependency information. The process is illustrated in Figure 1. More details on program slice generation can be found in [13].

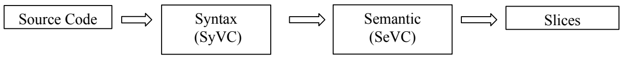

Source Code ⟹ Syntax (SyVC) ⟹ Semantic (SeVC) ⟹ Slices

Figure 1: Generating Program Slices fro Source Code.

2.2 Transforming Program Slices into Vectors

The program slices are then transformed into vectors to feed into neural networks. Each slice was transformed to an array of tokens in which all comments and white spaces were also removed before transformation and tokens were mapped with list of function names.

For each slice, the tokenized outputs were stored in pickle file and labeled with the unique sample ID (no duplicated ID). Each pickle file contains an array of 5 elements including a list of tokens, a target label (0/1), list of functions, vulnerability type, and sample ID with a main function name.

The tokens from each pickle file were converted to vectors using Word2Vector model built from Gensim package. The Word2Vector model converted tokens to vector based on cosine similarity distance which measures the angle between vectors in which the high similarity score indicates high similarity and a closer distance between tokens [15]. The cosine similarity is computed as follows:

$$sim(X,Y) = \frac{X \cdot Y}{\|X\| \times \|Y\|} = \frac{\sum_i (x_i \times y_i)}{\sqrt{\sum_i x_i^2} \times \sqrt{\sum_i y_i^2}}$$

The visualization of words in Word2Vector model is shown in Figure 2. As we can see, different program slice types have extremely different distributions of cosine similarities. This indicate that different program slice types convey different characteristics of vulnerabilities.

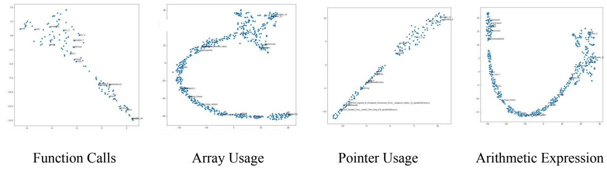

Function Calls
Array Usage
Pointer Usage
Arithmetic Expression

Figure 2: Visualized Words in W2V model for Each Vulnerability Type

3 MODEL OPTIMIZATION

3.1 Balancing Datasets

The training set should have a balanced number of non-vulnerable program slices and vulnerable program slices to ensure that the model can produce unbiased predictions. However, [13] used an imbalanced dataset in which non-vulnerable program slices only accounts for 15.6% of total program slices while vulnerable program slices accounts for 84.4% of total program slices.

To illustrate the issue, we compute the confusion matrix for the model fitted with imbalanced class sample (75% of class 0 and 25% of class 1). The result is presented in Table 1. The model has considerably more ability to predict class 0 as its specificity and negative prediction are significantly higher than its sensitivity and precision, respectively. Thus, the accuracy rate is biased towards class 0.

In order to solve an imbalanced class issue, a training set was resampled using down-sampling method to randomly extract samples from a majority class (label 0) from a training set. The new sample set has a balanced class label, comprising 50% vulnerable vector arrays and 50% non-vulnerable vector arrays.

Table 1: Confusion Matrix for Imbalanced Dataset

Predicted Class				
	Positive	Negative	Rate	
Positive	8.0	48.0	0.14285	Sensitivity
Negative	14.0	130.0	0.90277	Specificity
	0.36363	0.73033	0.60999	Accuracy
	Precision	Negprediction		

The process of down-sampling is shown in Figure 3. In the last step, all vector arrays for each slice program are adjusted to have a same length (same number of rows) as each array of vectors have different number of rows, but the neural networks required all vector input to have a same dimension (same number of columns and same number of rows). The mean of vector lengths for all slice programs was calculated and use as the threshold to adjust the vector lengths. If an array of vectors has a shorter length than the mean, it will embed with the zero array. For the array of vectors with longer length than the mean length, the vectors were truncated.

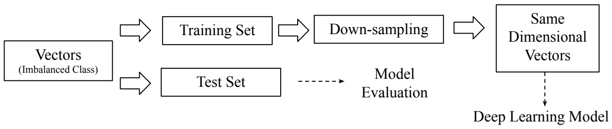


Figure 3: Down-sampling and Vector Adjustment

With the balanced dataset, we can see that the model has approximately same ability to predict class 0 and class 1, as shown in Table 2. This shows that balancing the dataset is crucial for the model to have balanced prediction power for both classes.

Table 2: Confusion Matrix for Balanced Dataset

Predicted Class				
	Positive	Negative	Rate	
Positive	1186.0	167.0	0.87916	Sensitivity
Negative	672.0	4018.0	0.86408	Specificity
	0.65236	0.96108	0.86747	Accuracy
	Precision	Negprediction		

3.2 Combining Datasets

In previous work [13], different models are built using different program slice types. From the visualization of Word2Vector models for each program slice type, presented in Figure 2, we note that different program slice types capture different characteristics of vulnerabilities, so we explore the use of a dataset combined from all different program slice types.

We perform a preliminary study using 1,000 randomly sampled program slices from each individual program slice types and 1,000 randomly sampled program slices among different program slice types, i.e. combined dataset. We compare the accuracy, sensitivity, and specificity for the models built using the sampled program slices from individual program slice types and the model built using the sampled program slices from the combined dataset. Our result is shown in Table 3.

Table 3: Comparison between individual datasets and combined dataset

Type	Accuracy	Sensitivity	Specificity
API	53%	69%	46%
AU	64%	79%	62%
PU	38%	83%	31%
AE	61%	61%	62%
COMBINED	61%	91%	53%

The model fitted with the combined data types outperform all models fitted with individual datasets in explaining the target class 1 (vulnerable) as the sensitivity is as high as 91%. Compared to API and PU models, the combined model performs better in detecting the target class 0 (non-vulnerable). The overall performance indicates that the combined model performs similarly to AU and AE models. The combined dataset is more appropriate for predicting vulnerabilities.

4 EVALUATION

4.1 Optimizers

There are several optimizers that can be applied to optimize neural network such as Stochastic Gradient Descent (SGD) by default, Adamax (SySe Paper), RMS Prop, and Adam.

Adam optimizer is a combination of RMSprop and SGD with momentum, which is an adaptive learning rate method and computationally efficient as it computes individual learning rates for different parameters.

According to Kingma et al. [9], ADAM method is appropriate for problems with large dataset and/or parameters, with non-stationary objectives, and for problems with very noisy and/or sparse gradients. ADAM optimizer will improve our neural network when a network has weak signals which is not sufficient to tune its weights effectively. Given the nature of software vulnerabilities which contain various causes, the ADAM method is an appropriate method to further examine and apply towards the model development in this paper.

We explore three different optimizers, ADAMAX, SGD, and ADAM, in order to find the best optimizer for our neural networks. The summary of models with ADAMAX and SGD optimizer is presented in Table 4.

We can see that the ADAM optimizer performs the best among the three optimizers for all program slice types. ADAM achieves an average accuracy rate of 90.0%. This is approximately 5% more than the accuracy rate of ADAMAX, which is used in previous work [13].

4.2 BGRU v.s. BLSTM

In this section, we compare the performance of different neural networks with a focus on BGRU and BLSTM.

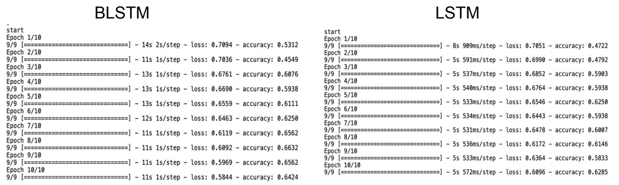
Comparing to LSTM, GRU has no explicit memory unit and no forget gate and update gate, hence it trains the model faster than LSTM, but may lead to a lower accuracy rate. GRU also has a simpler

Table 4: Accuracy Rate with Different Optimizers

Type	ADAMAX	SGD	ADAM
API	86.7%	63.1%	89.5%
AU	86.0%	58.6%	89.2%
PU	82.4%	62.3%	90.9%
AE	83.1%	67.1%	90.5%

architecture which reduces the number of hyperparameters. LSTM comprises both update gate and forget gate and remembers longer sequences than GRU. However, LSTM is found to be comparable to GRU on sequence modeling.

Bidirectional Recurrent Neural Networks provides the original input sequence to the first layer and a reversed copy of the input sequence to the second layer, so there are two layers side-by-side. Bidirectional RNNs are found to be more effective than regular RNNs. It has been widely used as it can overcome the limitations of a regular RNN [17]. The regular RNN model preserves only information of the past. Whereas, These Bidirectional networks have access to the past as well as the future information; therefore, the output is generated from both the past and future context and leads to a better prediction and classifying sequential patterns. The output from fitting LSTM and BLSTM models also indicates that the bidirectional unit outperforms the regular LSTM holding other hyperparameters constant, as shown in Figure 4.



BLSTM

```
start
Epoch 1/10
9/9 [==============================] - 14s 2s/step - loss: 0.7094 - accuracy: 0.5312
Epoch 2/10
9/9 [==============================] - 11s 1s/step - loss: 0.7036 - accuracy: 0.4549
Epoch 3/10
9/9 [==============================] - 13s 1s/step - loss: 0.6761 - accuracy: 0.6076
Epoch 4/10
9/9 [==============================] - 13s 1s/step - loss: 0.6690 - accuracy: 0.5938
Epoch 5/10
9/9 [==============================] - 13s 1s/step - loss: 0.6559 - accuracy: 0.6111
Epoch 6/10
9/9 [==============================] - 12s 1s/step - loss: 0.6463 - accuracy: 0.6250
Epoch 7/10
9/9 [==============================] - 11s 1s/step - loss: 0.6119 - accuracy: 0.6562
Epoch 8/10
9/9 [==============================] - 11s 1s/step - loss: 0.6092 - accuracy: 0.6632
Epoch 9/10
9/9 [==============================] - 11s 1s/step - loss: 0.5969 - accuracy: 0.6562
Epoch 10/10
9/9 [==============================] - 11s 1s/step - loss: 0.5844 - accuracy: 0.6424
```

LSTM

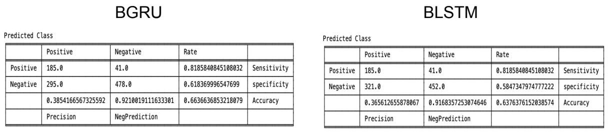
```
start
Epoch 1/10
9/9 [==============================] - 8s 909ms/step - loss: 0.7051 - accuracy: 0.4722
Epoch 2/10
9/9 [==============================] - 5s 591ms/step - loss: 0.6990 - accuracy: 0.4792
Epoch 3/10
9/9 [==============================] - 5s 537ms/step - loss: 0.6852 - accuracy: 0.5903
Epoch 4/10
9/9 [==============================] - 5s 540ms/step - loss: 0.6764 - accuracy: 0.5938
Epoch 5/10
9/9 [==============================] - 5s 533ms/step - loss: 0.6546 - accuracy: 0.6250
Epoch 6/10
9/9 [==============================] - 5s 534ms/step - loss: 0.6443 - accuracy: 0.5938
Epoch 7/10
9/9 [==============================] - 5s 531ms/step - loss: 0.6478 - accuracy: 0.6007
Epoch 8/10
9/9 [==============================] - 5s 536ms/step - loss: 0.6172 - accuracy: 0.6146
Epoch 9/10
9/9 [==============================] - 5s 533ms/step - loss: 0.6364 - accuracy: 0.5833
Epoch 10/10
9/9 [==============================] - 5s 572ms/step - loss: 0.6096 - accuracy: 0.6285
```

Figure 4: Model Fitting of BLSTM and LSTM

The BLSTM model has lower loss rate of 0.58 compared to a loss rate of 0.60 from LSTM. The BLSTM also have a higher accuracy rate of 64.2% than that of LSTM model of 62.8%. Note that both models were fit with same input parameter and dataset. The models were built with a small dataset of 1000 slices so the accuracy rate may not be high. The larger dataset would be used to fit the model in the later section.

The confusions matrix provides more details for model performance and validation of the models trained with 4,000 samples (80% of 5,000 samples) in Figure 5. The decision threshold is set to 0.5 for validation. The BGRU model outperforms the BLSTM in most metrics except the sensitivity. It has a higher accuracy, precision, and specificity which indicates the stronger ability of model to predict both vulnerability and non-vulnerability types.

For a larger dataset of 30,000, the BGRU also outperforms BLSTM in every metrics. The obvious improvement is the sensitivity which the BGRU has 90% of sensitivity which is 8% higher than that of BLSTM, indicating that the BGRU model can explain the vulnerability class better than the BLSTM, as shown in Figure 6. The model built from 100,000 datasets also shows the similar performance in which BGRU model performs better than BLSTM. Comparing to BLSTM, The BGRU network typically train, converge, and learn faster. Thus, the BLSTM models that fit with 10 epochs may not reach convergent which can explain why the models for BGRU perform better in 10 epochs.



BGRU

Predicted Class

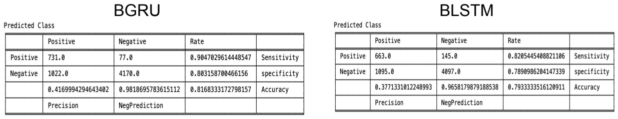
	Positive	Negative	Rate	
Positive	185.0	41.0	0.8185840845108032	Sensitivity
Negative	295.0	478.0	0.618369996547699	specificity
	0.3854166567325592	0.9210019111633301	0.6636636853218079	Accuracy
	Precision	NegPrediction		

BLSTM

Predicted Class

	Positive	Negative	Rate	
Positive	185.0	41.0	0.8185840845108032	Sensitivity
Negative	321.0	452.0	0.5847347974777222	specificity
	0.365612655878067	0.9168357253074646	0.6376376152038574	Accuracy
	Precision	NegPrediction		

Figure 5: Confusion Matrix for BGRU and BLSTM (fitted with 1,000 samples)



BGRU

Predicted Class

	Positive	Negative	Rate	
Positive	731.0	77.0	0.9047029614448547	Sensitivity
Negative	1022.0	4170.0	0.803158700466156	specificity
	0.4169994294643402	0.9818695783615112	0.8168333172798157	Accuracy
	Precision	NegPrediction		

BLSTM

Predicted Class

	Positive	Negative	Rate	
Positive	663.0	145.0	0.8205445408821106	Sensitivity
Negative	1095.0	4097.0	0.7890986204147339	specificity
	0.3771331012248993	0.9658179879188538	0.7933333516120911	Accuracy
	Precision	NegPrediction		

Figure 6: Confusion Matrix for BGRU and BLSTM (fitted with 30,000 samples)

4.3 Combined Datasets

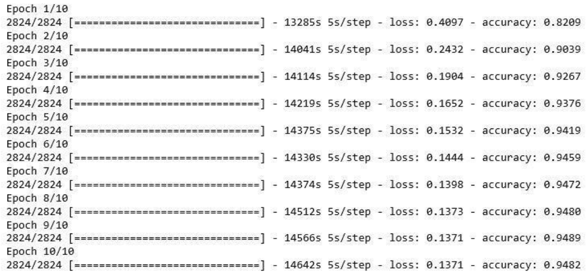
We combine the total 420,067 programs slices into one dataset, comprising 64,403, 42,229, 291,281, and 22,154 from API, AU, PU, and AE types, respectively. The combined dataset was spitted into a training set and a test set with 80/20 ratios. Then, the training set is down-sampling to ensure that the target classes (vulnerable and non-vulnerable) in dataset are balanced.

The model is built with ADAM optimizer. The hyperparameters of include Bidirectional gated recurrent unit (BGRU) of 256 neuron units with 2 hidden layers. Tanh function was applied to produce the outputs of 2 hidden layers and sigmoid function was applied to compute activation outputs in the last layer. The learning rate is 0.1 with batch size of 32, vector inputs of shape [mean vector lengths x 30]. The cross-entropy loss was chosen as it can speed up the convergence of loss.

We present the learning process in Figure 7, the learning process is faster in the beginning as the loss rates significantly decrease in epoch 1 to 3. The accuracy rates increase for as the training process goes from epoch 1 to 10. The model has the highest accuracy rate of 94.89% in epoch 9 and starts to decrease in epoch 10 as the error rate has no longer minimized. The network outputs range between 0 and 1 as a sigmoid function is applied the output layer.

Table 5 shows the confusion matrix on the test set. We can see that the model performs well to explain both target classes as the sensitivity and specificity are over 90%. However, the F1 score was further computed in the next section to blend the precision and sensitivity.

As presented in Figure 8, the F1 scores increase while balanced accuracy decreases as the threshold increases. The peak point of



```
Epoch 1/10
2824/2824 [==============================] - 13285s 5s/step - loss: 0.4097 - accuracy: 0.8209
Epoch 2/10
2824/2824 [==============================] - 14041s 5s/step - loss: 0.2432 - accuracy: 0.9039
Epoch 3/10
2824/2824 [==============================] - 14114s 5s/step - loss: 0.1904 - accuracy: 0.9267
Epoch 4/10
2824/2824 [==============================] - 14219s 5s/step - loss: 0.1652 - accuracy: 0.9376
Epoch 5/10
2824/2824 [==============================] - 14375s 5s/step - loss: 0.1532 - accuracy: 0.9419
Epoch 6/10
2824/2824 [==============================] - 14330s 5s/step - loss: 0.1444 - accuracy: 0.9459
Epoch 7/10
2824/2824 [==============================] - 14374s 5s/step - loss: 0.1398 - accuracy: 0.9472
Epoch 8/10
2824/2824 [==============================] - 14512s 5s/step - loss: 0.1373 - accuracy: 0.9480
Epoch 9/10
2824/2824 [==============================] - 14566s 5s/step - loss: 0.1371 - accuracy: 0.9489
Epoch 10/10
2824/2824 [==============================] - 14642s 5s/step - loss: 0.1371 - accuracy: 0.9482
```

Figure 7: Model Fitting with a Training Set

Table 5: Confusion Matrix for Test Set

Predicted Class				
	Positive	Negative	Rate	
Positive	10768.0	439.0	0.96082	Sensitivity
Negative	5898.0	67019.0	0.91911	Specificity
	0.64610	0.99349	0.92467	Accuracy
	Precision	Negprediction		

balanced accuracy is at 0.5. F1 is utilized to blend precision and specificity for model evaluation. The accuracy rate keeps increasing, hence balanced accuracy is more accurate as accuracy is biased with the increasing threshold.

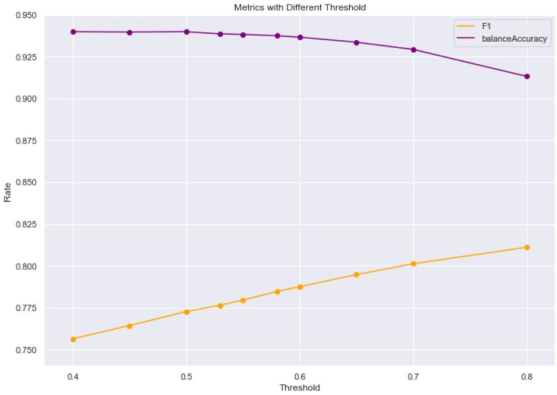


Figure 8: F1 v.s. Accuracy Rate for Different Thresholds

Overall, the model fitted with full dataset performs well with a high balanced accuracy rate of 93%. The high sensitivity and specificity imply a good ability of model in explaining both vulnerability and non-vulnerability classes. The model performs better in predicting non-vulnerability class as it has 99% in negative prediction. However, the predictive power for vulnerability class is still moderately strong as the F1 ranges between 75% to 80% across different thresholds.

5 RELATED WORK

Many techniques have been proposed to detect vulnerabilities in the source code using various human-defined features such as source code text features [5], complexity, code churn, and developer activity metrics [19], abstract syntax trees [24], function imports and function calls [16]. The main drawback of these techniques is that they requires considerable human effort to define these features.

Recent techniques use deep learning on the source code of programs to detect vulnerabilities so that no human experts is needed to define features [11, 13, 14, 27]. They either rely on one type of training data or use imbalanced training data. Our work differs from them by using a balanced dataset combined from different types of training data.

Rather than using static information collected from the source code, some techniques apply machine learning on dynamic information collected from the sequences of function calls to detect vulnerabilities [6, 22]. Particularly deep learning models have been shown to have better accuracy than traditional machine learning models [22].

6 CONCLUSION

We present a study on using neural networks for detecting software vulnerabilities in this paper. The neural networks are trained with program slices extracted from the source code of 14,000 C/C++ programs. We compare different types of training data and different types of neural networks. Our result shows that the model combining different types of characteristics of source code surpasses models based on individual type of characteristics of source code. Using a balanced number of vulnerable program slices and non-vulnerable program slices ensures a balanced accuracy in predicting both vulnerable code and non-vulnerable code. We find that BGRU performs the best among other neural networks. Its accuracy reaches 94.89% with a sensitivity of 96% and a specificity of 91%.

REFERENCES

[1] 2020. Apple Security Bounty. https://developer.apple.com/security-bounty/.

[2] 2020. Facebook. https://www.facebook.com/whitehat/.

[3] 2020. Intel Bug Bounty Program. https://www.intel.com/content/www/us/en/security-center/bug-bounty-program.html.

[4] 2020. Microsoft Bug Bounty Program. https://www.microsoft.com/en-us/msrc/bounty?rtc=1.

[5] Boris Chernis and Rakesh Verma. 2018. Machine Learning Methods for Software Vulnerability Detection. In *Proceedings of the Fourth ACM International Workshop on Security and Privacy Analytics* (Tempe, AZ, USA) *(IWSPA '18)*. Association for Computing Machinery, New York, NY, USA, 31–39. https://doi.org/10.1145/3180445.3180453

[6] Gustavo Grieco, Guillermo Luis Grinblat, Lucas Uzal, Sanjay Rawat, Josselin Feist, and Laurent Mounier. 2016. Toward Large-Scale Vulnerability Discovery Using Machine Learning. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy* (New Orleans, Louisiana, USA) *(CODASPY '16)*. Association for Computing Machinery, New York, NY, USA, 85–96. https://doi.org/10.1145/2857705.2857720

[7] J. Guo, J. Cheng, and J. Cleland-Huang. 2017. Semantically Enhanced Software Traceability Using Deep Learning Techniques. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 3–14. https://doi.org/10.1109/ICSE.2017.9

[8] S. Kim, S. Woo, H. Lee, and H. Oh. 2017. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*. 595–614. https://doi.org/10.1109/SP.2017.62

[9] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *The 3rd International Conference for Learning Representations* (San Diego, California).

[10] J. Li, P. He, J. Zhu, and M. R. Lyu. 2017. Software Defect Prediction via Convolutional Neural Network. In *2017 IEEE International Conference on Software Quality,*

Reliability and Security (QRS). 318–328. https://doi.org/10.1109/QRS.2017.42

[11] Z. Li, D. Zou, J. Tang, Z. Zhang, M. Sun, and H. Jin. 2019. A Comparative Study of Deep Learning-Based Vulnerability Detection System. *IEEE Access* 7 (2019), 103184–103197. https://doi.org/10.1109/ACCESS.2019.2930578

[12] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. 2016. VulPecker: An Automated Vulnerability Detection System Based on Code Similarity Analysis. In *Proceedings of the 32nd Annual Conference on Computer Security Applications* (Los Angeles, California, USA) *(ACSAC '16)*. Association for Computing Machinery, New York, NY, USA, 201–213. https://doi.org/10.1145/2991079.2991102

[13] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2018. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. arXiv:1807.06756 [cs.LG]

[14] Z. Li, D. Zou, Shouhuai Xu, Xinyu Ou, H. Jin, S. Wang, Zhijun Deng, and Y. Zhong. 2018. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium*, Vol. abs/1801.01681.

[15] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. arXiv:1301.3781 [cs.CL]

[16] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. 2007. Predicting Vulnerable Software Components. In *Proceedings of the 14th ACM Conference on Computer and Communications Security* (Alexandria, Virginia, USA) *(CCS '07)*. Association for Computing Machinery, New York, NY, USA, 529–540. https://doi.org/10.1145/1315245.1315311

[17] M. Schuster and K. K. Paliwal. 1997. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing* 45, 11 (1997), 2673–2681. https://doi.org/10.1109/78.650093

[18] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. 2015. Recognizing Functions in Binaries with Neural Networks. In *Proceedings of the 24th USENIX Conference on Security Symposium* (Washington, D.C.) *(SEC'15)*. USENIX Association, USA, 611–626.

[19] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne. 2011. Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities. *IEEE Transactions on Software Engineering* 37, 6 (2011), 772–787. https://doi.org/10.1109/TSE.2010.81

[20] Song Wang, Taiyue Liu, and Lin Tan. 2016. Automatically Learning Semantic Features for Defect Prediction. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 297–308. https://doi.org/10.1145/2884781.2884804

[21] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 87–98.

[22] F. Wu, J. Wang, J. Liu, and W. Wang. 2017. Vulnerability detection with deep learning. In *2017 3rd IEEE International Conference on Computer and Communications (ICCC)*. 1298–1302. https://doi.org/10.1109/CompComm.2017.8322752

[23] Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. 2012. Generalized Vulnerability Extrapolation Using Abstract Syntax Trees. In *Proceedings of the 28th Annual Computer Security Applications Conference* (Orlando, Florida, USA) *(ACSAC '12)*. Association for Computing Machinery, New York, NY, USA, 359–368. https://doi.org/10.1145/2420950.2421003

[24] Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. 2012. Generalized Vulnerability Extrapolation Using Abstract Syntax Trees. In *Proceedings of the 28th Annual Computer Security Applications Conference* (Orlando, Florida, USA) *(ACSAC '12)*. Association for Computing Machinery, New York, NY, USA, 359–368. https://doi.org/10.1145/2420950.2421003

[25] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. 2013. Chucky: Exposing Missing Checks in Source Code for Vulnerability Discovery. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security* (Berlin, Germany) *(CCS '13)*. Association for Computing Machinery, New York, NY, USA, 499–510. https://doi.org/10.1145/2508859.2516665

[26] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun. 2015. Deep Learning for Just-in-Time Defect Prediction. In *2015 IEEE International Conference on Software Quality, Reliability and Security*. 17–26. https://doi.org/10.1109/QRS.2015.14

[27] M. Zagane, M. K. Abdi, and M. Alenezi. 2020. Deep Learning for Software Vulnerabilities Detection Using Code Metrics. *IEEE Access* 8 (2020), 74562–74570. https://doi.org/10.1109/ACCESS.2020.2988557