Learning-based Vulnerability Detection in Binary Code

Amy Aumpansub
DePaul University
Chicago, Illinois, USA
amy.aump@gmail.com

Zhen Huang
DePaul University
Chicago, Illinois, USA
zhen.huang@depaul.edu

ABSTRACT

Cyberattacks typically exploit software vulnerabilities to compromise computers and smart devices. To address vulnerabilities, many approaches have been developed to detect vulnerabilities using deep learning. However, most learning-based approaches detect vulnerabilities in source code instead of binary code. In this paper, we present our approach on detecting vulnerabilities in binary code. Our approach uses binary code compiled from the SARD dataset to build deep learning models to detect vulnerabilities. It extracts features on the syntax information of the assembly instructions in binary code, and trains two deep learning models on the features for vulnerability detection. From our evaluation, we find that the BLSTM model has the best performance, which achieves an accuracy rate of 81% in detecting vulnerabilities. Particularly the F1-score, recall, and specificity of the BLSTM model are 75%, 95% and 75% respectively. This indicates that the model is balanced in detecting both vulnerable code and non-vulnerable code.

CCS CONCEPTS

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

KEYWORDS

software vulnerability, vulnerability detection, deep learning, neural network, machine learning

ACM Reference Format:
Amy Aumpansub and Zhen Huang. 2022. Learning-based Vulnerability Detection in Binary Code. In *2022 14th International Conference on Machine Learning and Computing (ICMLC) (ICMLC 2022), February 18–21, 2022, Guangzhou, China.* ACM, New York, NY, USA, Article 4, 6 pages. https://doi.org/10.1145/3529836.3529926

1 INTRODUCTION

The security and reliability of computer systems are essential to our daily lives. However, software vulnerabilities allowed many real-world cyberattacks to breach the security and reliability of computer systems and caused huge financial loss [1, 3, 17, 21]. For example, a recent data breach took advantage of a vulnerability to steal the private information of over 500 millions Facebook users and caused the loss of more than $5 billion U.S. dollars [1]. As a result, it is critical to detect and address vulnerabilities effectively and efficiently.

Particularly detecting vulnerabilities in off-the-shelve software in which only the binary code is available will allow users to rapidly mitigate the vulnerabilities without the need to wait for software vendors to release patches for the vulnerabilities [8]. This will considerably reduce the risks of cyberattacks because on average it takes over a month for software vendors to fix a vulnerability [6].

With the advances of machine learning and deep learning techniques, learning-based approaches have achieved success in many areas of software security and reliability [2, 4, 5, 7, 10–15, 22–25]. The vast majority of them focus on open-source projects and extract features from the program source code for model training, because the program source code contains a wealth of information about the programs, such as data types, variable names, function prototypes, and high-level program constructs. Unfortunately the binary code of off-the-shelve software lacks the aforementioned information provided by program source code. Thus it is challenging to apply machine learning on program binary code.

This paper presents our approach on detecting vulnerabilities in binary code using deep learning. It addresses two major challenges.

First, deep learning algorithms require features to classify vulnerable code and non-vulnerable code. What kind of features should we extract from binary code?

Second, we need to collect a dataset of binary code, choose the granularity of our vulnerability detection, and establish a ground-truth for our vulnerability detection. Do we identify vulnerabilities at the level of programs, functions, or basic blocks? How do we get the labels for vulnerable code and non-vulnerable code?

Unlike prior work that treats program code as a bag of words or tokens, our approach uses the syntax information on the assembly instructions as features to train machine learning and deep learning models. These features include instruction mnemonics, operand types, operand positions, and operand values.

Our approach creates the dataset of binary code by compiling the C/C++ programs from the SARD dataset [20], which is widely used as a test bed for detecting vulnerabilities in the source code. The SARD dataset comes with labels for vulnerable code and non-vulnerable code at the level of functions. We choose to detect vulnerabilities at the granularity of functions so that we can directly use the labels come with the dataset.

To choose the optimal hyperparameters, we use grid search to train a LSTM model with different values of hyperparameters on the dataset and compare the performance of the models. We use the values of hyperparameters that produce the best performance.

Overall our work makes the following contributions:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICMLC 2022, February 18–21, 2022, Guangzhou, China
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9570-0/22/02…$15.00
https://doi.org/10.1145/3529836.3529926

- We use statistics of the syntax information of assembly instructions in program binary code as features for detecting vulnerabilities in program binary code.
- Using these features, we build two deep learning models, BLSTM and BGRU to detect vulnerabilities.
- Our evaluation shows that the syntax features extracted from assembly instructions are effective for detecting vulnerabilities. Our BLSTM model achieves an accuracy of 81.3%.

The paper is organized as follows. We present our methodology in Section 2. Section 3 describe the details on how we train our deep learning models, respectively. Particularly we present evaluation results on vulnerability detection in Section 3.6. Finally we discuss related work in Section 4 and conclude in Section 5.

2 METHODOLOGY

The program binary code is produced by compiling C/C++ programs in the SARD dataset. The CFG (Control Flow Graph) is extracted from each function in the binary code using `angr`, a python framework for analyzing binary code [19]. The assembly instructions and operands are extracted from the basic blocks and transformed to a vector using the CounterVectorizer for feature extraction. We train neural networks including Long Short-Term Memory (LSTM), Bidirectional LSTM (BLSTM), and Bidirectional GRU (BGRU).

2.1 Dataset

Our dataset consists of the binary programs generated by compiling the 13,443 programs written in C/C++ programming languages, which are from the Software Assurance Reference Dataset (SARD). The programs were compiled with gcc on Linux. They comprise 44,494 binary functions which include:

- 31,009 non-vulnerable functions (69.69 % of total functions)
- 13,485 vulnerable functions (30.31 % of total functions)

Dependent Variable. A binary variable that has a value of 0 for vulnerable function and 1 for non-vulnerable function.

Independent Variables. The numeric vectors of 105 features are extracted from each basic block in all functions using the CounterVectorizer Model. Our features focus on the syntax information of assembly instructions, i.e. the mnemonics and operands of assembly instructions, because it is straightforward to extract them. We plan to use control flow and data flow information between basic blocks, which can be extracted from binary code using more complicated program analysis [8], for our future work.

The sample observations of dataset are shown in Figure 1 in which each observation is a vector containing the total number of each of the 105 feature appeared in basic blocks of a binary function. The details of how the dataset was generated and feature extraction were discussed in the data preprocessing section.

2.2 Data Preprocessing

Instructions and Operands Extraction. The process to extract the set of instructions and operands from the binary programs is shown in Figure 2. The first step is to generate the CFG for each vulnerable function and non-vulnerable function of binary programs. The second step involves the machine instructions and operands extraction. Finally, the last step extracts unique features from the set of machine instructions and operands of all functions.

Independent Variables																	Dependent Variable
add	and	byte	call	cdq	cdqe	cmp	dword	endbr64	…	rep	ret	shl	shr	sub	test	xor	target
1	0	1	1	0	0	2	7	1	…	0	1	0	0	1	0	2	0
0	0	1	3	0	0	0	1	2	…	0	1	0	0	1	1	2	0
2	0	2	1	0	2	2	11	1	…	0	1	0	0	1	0	2	0
1	0	1	1	0	1	2	13	1	…	0	1	0	0	1	0	2	0
0	0	1	3	0	0	0	1	1	…	0	1	0	0	1	0	2	0

Figure 1: Sample Observations.

Step 1: The vulnerable functions and non-vulnerable functions were extracted from a binary program (object files). and the address of each function was parsed as an input for the next step.

Step 2: Each binary function was parsed using `angr` with a function address to construct Control Flow Graph which provides all basic blocks belonging to the function.

Step 3: For each function, the machine instructions and operands were extracted from the basic blocks as a set of tokens with an assigned test number. The output of this step is the set of instruction mnemonics and operands for all the instructions in each function. We use the following information on operands:

a) Data types including qword, dword, byte, etc.
b) Registers including both 32-bit and 64-bit registers
c) Memory addresses
d) Integer constants

Particularly we choose to include integer constants because prior work [16] shows that they have high feature weight in uniquely identifying program code. The operands are divided into 4 groups and labeled by number 1 and 2 which are operand positions for each assembly instruction.

Transforming Feature to Vectors. There are a total of 105 unique features extracted from the sets of instructions and operands for all functions, which includes 54 machine instructions and 51 operands comprising 7 data types, 40 registers, 2 groups of addresses, 2 groups of constants, as presented in Table 1.

Table 1: Features Extracted from All Functions.

Instructions	Operands (position 1, position 2)			
	Data Type	Register	Address	Constant
mov, add, call	$byte_1$	eax_1	$address_1$	$constant_1$
lea, sub, push	$byte_2$	eax_2	$address_2$	$constant_2$
…	$dword_1$	rax_1		
	$dword_2$	rax_2		
	…	…		

Each unique feature represents 1 instruction or operand which is represented by a column of matrix, while the value of each cell is the count of the feature in that set of instructions and operands from each function. The row represents the vectors of one function with a target label 0 for non-vulnerability function and 1 for vulnerability function. In short, each vector represents the total number of each feature appeared in basic blocks of a binary function.

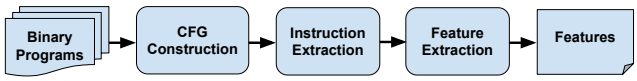

Figure 2: Extracting Features From Binary Programs.

The CounterVectorizer Model was utilized to transform the set of instructions and operands of each function into a vector on the basis of the frequency (count) of each feature.

Z-score Normalization. Support Vector Machine algorithm classifies the data by finding the hyperplane that maximizes the margin between the two classes. Thus it can be sensitive to feature scaling. To achieve an accurate model, we normalize the set of vectors before fitting the SVM model. Additionally, the normalized data were later used to build neural networks. The Z-score normalization transforms vectors of each feature by subtracting the mean and then scaling to unit variance. The normalized data have a normal distribution with a mean of 0 and a standard deviation of 1.

Dataset Splitting. The 44,494 observations were into 3 sets: Training, Validation, and Test sets using stratify sampling to ensure the proper ensure that each class within 3 sets has a proper representation of the entire population. There are a 69.63% of class 0 and 30.30% of class 1 in each set which represent the entire population of class 0 and class 1 in the original dataset, as shown in Table 2.

We selected 10% of the dataset randomly as the test set for an unbiased evaluation. The remaining data were split into a training set (70%) for fitting model and a validation set (30%) for model evaluation while tuning hyper-parameters.

Table 2: Number of Observations in Each Dataset.

	Train Set	Validation Set	Test Set
Class 0	19,535	8,373	3,101
Class 1	8,495	3,641	1,349

3 NEURAL NETWORKS

Each vector array discussed in Section 2 represents the number of feature counts in one binary function. The vector arrays were reshaped from two-dimensional vectors to three-dimensional vectors as input to deep learning models.

3.1 Hyperparameters

We use two different base models to search for the optimal hyperparameters for our deep learning models. Both base models use Long short-term memory (LSTM), which contains 256 neuron units and 2 hidden layers fitted with a batch size of 32 for 100 epochs. The "Sigmoid" function was applied to the output layer. The binary cross-entropy loss is selected as loss function as it can speed up the learning process and convergence. The models use the "Adam" optimizer and the learning rate is 0.001. For the hidden layer, base model 1 uses the "ReLu" activation function while base model 2 uses the "Tanh" activation function.

Base model 1 starts to be overfitted after Epoch 40 as the validation loss starts to increase while the training loss keeps decreasing, as shown in Figure 3. The highest validation accuracy is 81.10% with the loss rate of 0.34 at Epoch 40.

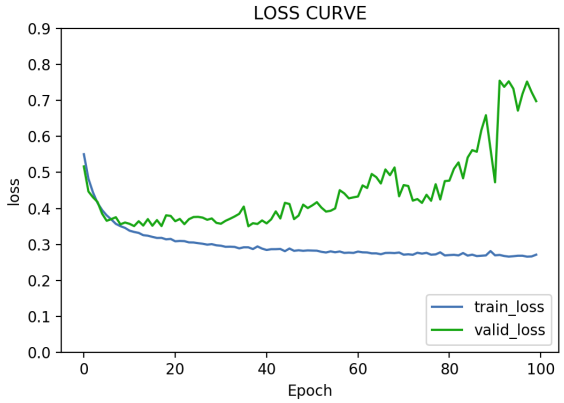

Figure 3: Base Model 1 Loss.

As shown in Figure 4, base Model 2 starts to be overfitted after Epoch 31 as the validation loss reaches the minimum and is constant while the training loss keeps decrease. Comparing to base model 1 ("ReLu"), base model 2 ("Tanh") performs slightly better as it has higher accuracy rate and lower loss rate. Thus, we choose to use the "Tanh" activation function to develop our neural networks.

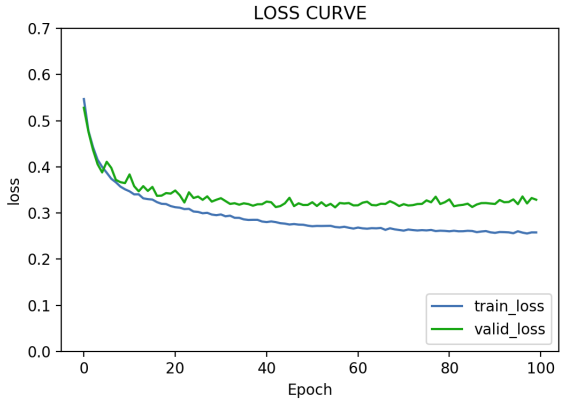

Figure 4: Base Model 2 Loss.

3.2 Bidirectional Recurrent Neural Networks

Bidirectional Recurrent Neural Networks promote the better understanding of context as it 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. 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. Additionally, 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 [18]. The regular RNN model preserves only past information. As a result, we choose to use BLSTM and BGRU.

3.3 BLSTM

The BLSTM model was fitted with the same architecture as the Base Model 2 except the Bidirectional LSTM was applied instead of regular LSTM. The performance metrics of LSTM and BLSTM models shown in Table 3 indicates that the bidirectional unit outperforms the regular LSTM holding other hyperparameters constant.

The BLSTM model slightly performs better than the LSTM Model. It has a lower loss rate of 0.31 compared to a loss rate of 0.32 from LSTM (Figure 5 and 6). The BLSTM also have an accuracy rate of 82%, which is higher than LSTM model's accuracy of 81%. The BLSTM starts to overfit after Epoch 33 similar to the LSTM model which starts to overfit after Epoch 31. The learning rate was adjusted in the next model to reduce the overfitting effect beyond epoch 30.

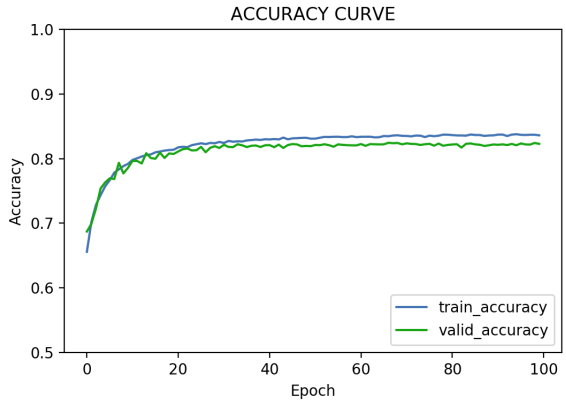

Figure 5: BLSTM Accuracy.

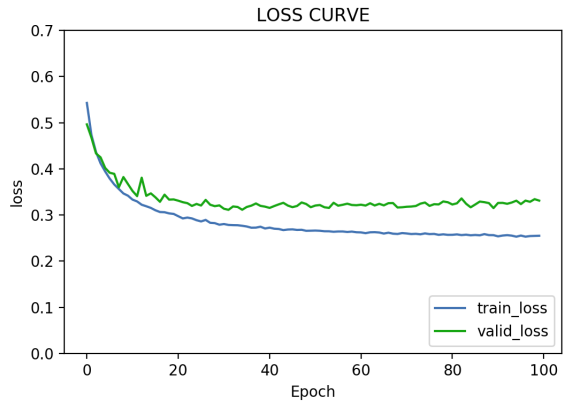

Figure 6: BLSTM Loss.

3.4 BGRU

In this phase, we built the model with Bidirectional Gated Recurrent Unit (BGRU). 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. Comparing to LSTM, the GRU has a simpler architecture which reduces the number of hyperparameters. The BGRU model performs slightly worse than the LSTM model ((Figure 7 and 8). Its accuracy rate (validation) is 81% which is slightly lower than that of the BLSTM model.

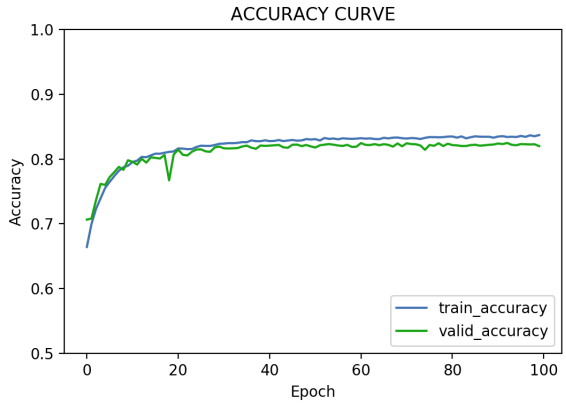

Figure 7: BGRU Accuracy.

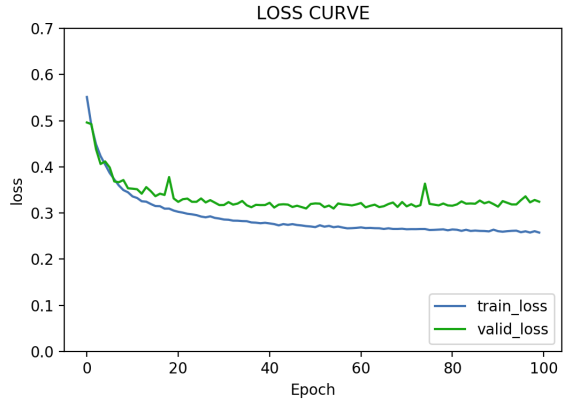

Figure 8: BGRU Loss.

Table 3: Deep Learning Model Comparison.

Model	Accuracy		Loss	
	Training	Validation	Training	Validation
LSTM	0.82	0.81	0.29	0.32
BLSTM	0.83	0.82	0.28	0.31
BGRU	0.82	0.81	0.30	0.32

3.5 Threshold Value for Binary Classification

The Sigmoid activation function was applied to all previous models to produce the Neural Network outputs in the last layer. The outputs are the decimal number ranges from 0 to 1, indicating the probability of the outputs to be classified as class 0 or 1 based on the chosen threshold. The previous models use the threshold of 0.5 which was applied to outputs for predicting the target class. For example, if the outputs are less than or equal to 0.5, they are classified as class 0 (Non-vulnerability functions). According to the performance of the previous models, the models suffer from some Type I and Type II errors, thus the threshold value needs to be tuned to minimize the cost of Type 1 Error and Type 2 Error.

Figure 9 shows the accuracy rates and F1 scores of validation sets across different threshold values. The optimal threshold is 0.45 which has the highest accuracy rate and F1 score.

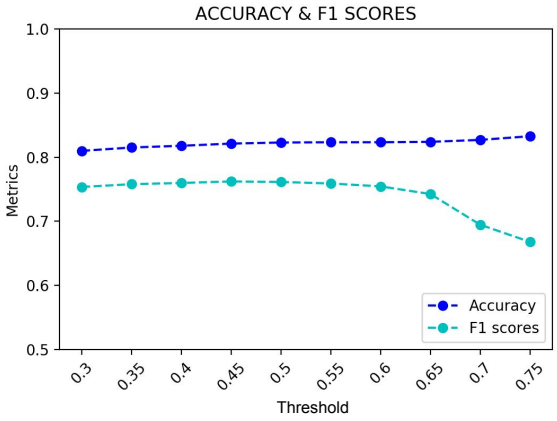

Figure 9: Accuracy rates and F1 scores across different threshold.

3.6 Vulnerability Detection

Our BLSTM model contains 256 neuron units and 2 hidden layers fitted with a batch size of 32 for 50 epochs. The accuracy curve of the model is presented in Figure 10. The model has a good performance with an accuracy rate of 81%.

As presented in Table 4, the BLSTM model has a F1 score of 75% and recall of 95% in detecting vulnerable code. Its specificity is 75%. The macro average results and weighted average results are similar. This indicates that the model performs well in classifying vulnerable code and non-vulnerable code.

Table 4: Performance of BLSTM on the Test Set

	Precision	Recall	F1-score	Support
Class 0	0.97	0.75	0.85	3,101
Class 1	0.63	0.95	0.75	1,349
Accuracy			0.81	4,450
Macro Ave.	0.80	0.85	0.80	4,450
Weighted Ave.	0.87	0.81	0.82	4,450

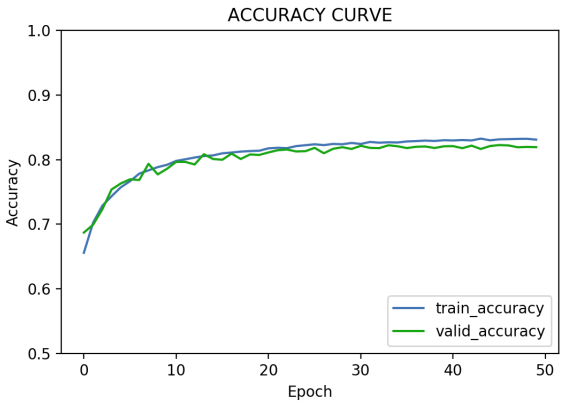

Figure 10: Vulnerability Detection Accuracy.

4 RELATED WORK

In this section, we discuss prior work on vulnerability detection. We categorize prior work into *dynamic analysis*, which execute target programs, and *static analysis*, which do not execute target programs. For static analysis, we focus on the work using machine learning and deep learning.

4.1 Machine Learning

VDiscover [5] learns from the sequences of API calls to predict vulnerabilities. It leverages three machine learning algorithms: logistic regression, multi-layer perceptron (MLP), and random forest. The dataset is composed of the sequences of API calls collected both statically from binary code and dynamically from program execution.It achieves an accuracy of 55% on vulnerable code and 83% on non-vulnerable code, with a false positive rate of 45%.

Yamaguchi et al. extract the information relevant to API function calls for all the functions of a target program, from the source code of the target program, then convert the extracted information into vectors, and use principal component analysis (PCA) on the vectors to identify the dominant usage pattern of API functions for each function in the target program [25]. By comparing the usage pattern of functions with that of a known vulnerable function, it predicts vulnerable functions.

VulPecker uses code similarity analysis to detect vulnerabilities [13]. It extracts features from vulnerability patches, and uses SVM to choose the best-performing code similarity algorithm for distinguishing patched code and unpatched code.

4.2 Deep Learning

Wang et al. use syntax information of Java programs to predict bugs. They collect tokens representing method calls, method and type declarations, and control flows from the abstract syntax tree (AST) of Java code, and build deep belief network (DBF) models on the tokens to generate semantic features to distinguish programs containing bugs and programs not containing bugs [22]. The evaluation shows that the semantic features considerably outperform the traditional manually-extracted features such as lines of code, number of operators, and number of methods.

To predict vulnerabilities, Wu et al. use the sequences of C library function calls as the dataset to build neural network models [24]. They treat each sequence of C library functions calls as a list of word tokens and convert them into vectors, then train three neural network models: CNN, LSTM, and CNN-LSTM with the vectors. Their evaluation shows that the neural network models remarkably perform better than the MLP used by VDiscover [5].

VulDeePecker [14] uses code gadgets, program statements that are data dependent or control dependent with each other, as the features to train neural networks to detect vulnerabilities. Focusing on library and API function calls, it extracts relevant program slices as code gadgets and converts them into vectors using word2vec. It then builds BLSTM models on the vectors for vulnerability detection. Comparing to prior work based on machine learning, such as VulPecker [13], VulDeePecker significantly reduces false positives.

Our prior work [2] detects vulnerabilities using LSTM, BLSTM, and BGRU on the syntax and semantic characteristics of program source code. Unlike this work, it extracts features from source code program slices relevant to four different types of characteristics: library or API function calls, array usage, pointer usage, and arithmetic expressions. The highest accuracy achieved by these models is 94%. While it achieves higher accuracy than this work, it requires access to program source code, which is not required by this work.

4.3 Dynamic Analysis

Typically static analysis can explore more program code and program state than dynamic analysis. But static analysis tends to have false positives as it has to approximate program state which may not be realized in actual program executions. Unlike static analysis, dynamic analysis [9, 10, 23] observes the exact program state that can be realized in program executions so it tends to have fewer false positives, although it suffers from lower code coverage.

5 CONCLUSION

This paper presents our approach to detecting vulnerabilities in binary code using deep learning. We use the binary code compiled from C/C++ programs in the SARD dataset and choose the syntax information on assembly instructions in the binary code as features to train deep learning models. Our evaluation shows that the BLSTM model outperforms the BGRU model. It achieves an accuracy of 81% in detecting vulnerabilities. The approximate equality of macro average results and weighted average results suggests that the models perform well in classifying both vulnerable code and non-vulnerable code.

REFERENCES

[1] 533 million Facebook users' phone numbers and personal data have been leaked online 2021. https://www.cshub.com/attacks/articles/iotw-facebook-data-leak-impacts-533-million-users.

[2] Amy Aumpansub and Zhen Huang. 2021. Detecting Software Vulnerabilities Using Neural Networks. In *Proceedings of ICMLC 2021: 13th International Conference on Machine Learning and Computing, Shenzhen China, 26 February, 2021- 1 March, 2021*. ACM, 166–171. https://doi.org/10.1145/3457682.3457707

[3] CISA orders agencies to quickly patch critical Netlogon bug 2020. https://www.cyberscoop.com/cisa-netlogon-microsoft-vulnerability-emergency/.

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

[5] 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 (CODASPY '16)*. Association for Computing Machinery, New York, NY, USA, 85–96. https://doi.org/10.1145/2857705.2857720

[6] Zhen Huang, Mariana DAngelo, Dhaval Miyani, and David Lie. 2016. Talos: Neutralizing Vulnerabilities with Security Workarounds for Rapid Response. In *Proceedings of 2016 IEEE Symposium on Security and Privacy (S&P 2016)*. 618–635. https://doi.org/10.1109/SP.2016.43

[7] Zhen Huang and David Lie. 2014. Ocasta: Clustering Configuration Settings for Error Recovery. In *Proceedings of 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '14)*. 479–490. https://doi.org/10.1109/DSN.2014.51

[8] Zhen Huang and Gang Tan. 2019. Rapid Vulnerability Mitigation with Security Workarounds. In *Proceedings of the 2nd NDSS Workshop on Binary Analysis Research (BAR '19)*.

[9] Zhen Huang and Xiaowei Yu. 2021. Integer Overflow Detection with Delayed Runtime Test. In *Proceedings of ARES 2021: The 16th International Conference on Availability, Reliability and Security, Vienna, Austria, August 17-20, 2021*, Delphine Reinhardt and Tilo Müller (Eds.). ACM, 28:1–28:6. https://doi.org/10.1145/3465481.3465771

[10] Bo Jiang, Ye Liu, and WK Chan. 2018. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 259–269.

[11] Tue Le, Tuan Nguyen, Trung Le, Dinh Phung, Paul Montague, Olivier De Vel, and Lizhen Qu. 2018. Maximal divergence sequential autoencoder for binary software vulnerability detection. In *International Conference on Learning Representations*.

[12] Zhen Li, Deqing Zou, Jing Tang, Zhihao Zhang, Mingqian Sun, and Hai 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

[13] 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 (ACSAC '16)*. Association for Computing Machinery, New York, NY, USA, 201–213. https://doi.org/10.1145/2991079.2991102

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

[16] Dhaval Miyani, Zhen Huang, and David Lie. 2017. BinPro: A Tool for Binary Source Code Provenance. arXiv:1711.00830.

[17] Russian government hackers are behind a broad espionage campaign that has compromised U.S. agencies, including Treasury and Commerce 2020. https://www.washingtonpost.com/national-security/russian-government-spies-are-behind-a-broad-hacking-campaign-that-has-breached-us-agencies-and-a-top-cyber-firm/2020/12/13/d5a53b88-3d7d-11eb-9453-fc36ba051781_story.html.

[18] Mike Schuster and Kuldip K Paliwal. 1997. Bidirectional recurrent neural networks. *IEEE transactions on Signal Processing* 45, 11 (1997), 2673–2681.

[19] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*.

[20] Software Assurance Reference Dataset (SARD) 2021. https://samate.nist.gov/SARD/.

[21] VMware Flaw a Vector in SolarWinds Breach? 2020. https://krebsonsecurity.com/2020/12/vmware-flaw-a-vector-in-solarwinds-breach/.

[22] 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 (ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 297–308. https://doi.org/10.1145/2884781.2884804

[23] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. 2010. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *2010 IEEE Symposium on Security and Privacy*. IEEE, 497–512.

[24] 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

[25] Fabian Yamaguchi, Felix Lindner, and Konrad Rieck. 2011. Vulnerability Extrapolation: Assisted Discovery of Vulnerabilities Using Machine Learning. In *Proceedings of the 5th USENIX Conference on Offensive Technologies (WOOT'11)*. USENIX Association, USA, 13.