Published by :
http://www.ijert.org

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 11 Issue 02, February-2022

Deep Learning to Detect Software Vulnerabilities

Gourav Bansal
Kurukshetra University

***Abstract*- The importance of automated vulnerability analysis techniques is growing as more software is developed. In this research, we present a deep learning-based method for learning assembly code in order to detect software flaws. Unlike previous research that relied on API function call sequences, our method begins by storing the assembly code in an immutable vector before using deep learning to learn the assembly language. When it comes to modeling assembly code, we choose Instruction2vec, which is efficient in vectorizing the code. We classify if the new functions have software weaknesses or not after learning the assembly code of the current functions using the vector provided by Instruction2vec. Many ways to detecting vulnerabilities using deep learning have been developed to solve vulnerabilities. Most learning-based approaches, on the other hand, discover vulnerabilities in source code rather than binary code. We present our method for detecting vulnerabilities in binary code in this paper. Our method builds deep learning models to discover vulnerabilities using binary code produced from the SARD dataset.**

Keywords- Vulnerability, Binary Code, Vulnerability detection, security, SARD dataset, Deep Learning, symmetric cryptographic algorithms, API function calls.

1. INTRODUCTION

Detecting vulnerabilities in software systems before they are distributed to consumers is one of the most successful approaches to correcting issues. A wide number of ways to detecting vulnerabilities have been proposed over the years. Fuzzing [6], symbolic execution [2], taint analysis and machine learning [9] are among the approaches they employ. To corrupt computer systems, many real-world cyberattacks [1,7] leveraged software vulnerabilities. A vulnerability was to blame for the recent data breach that exposed the private information of 500 million Facebook users [1]. As a result, resolving vulnerabilities effectively and efficiently is crucial for cybersecurity.

Learning-based approaches have been proven to have the potential for reliable vulnerability detection with recent advances in machine learning, particularly deep learning techniques [8]. 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, the vast majority of them focus on open-source projects and extract features from the program source code for model training.

2. APPROACHES ON DETECTING VULNERABILITIES

First, we must collect a binary code dataset, determine the granularity of our vulnerability detection, and construct a vulnerability detection process capability. Do we look for flaws in individual programs, functions, basic blocks, or program slices?
How do we determine which code is susceptible and which is not?

Second, Deep learning systems, for example, require features to distinguish between vulnerable and non-vulnerable code. What characteristics should we look for in binary code.

Our method generates a binary code dataset by compiling C/C++ programs from the SARD dataset, which is extensively used as a testbed for discovering source code vulnerabilities at the function level, the SARD dataset includes labels for susceptible and nonvulnerable code. We chose to discover vulnerabilities at the function level so that we could use the labels that came with the dataset right away. Unlike previous work that analyzes program code as a collection of words or tokens, our method exploits the semantic information contained in binary code assembly instructions as features to train machine learning and deep learning models. Instruction mnemonics, operand types, operand placements, and operand names are among these properties.

We utilize grid search to train an LSTM model with multiple values of hyperparameters on the dataset and compare the performance of the models to find the best hyperparameters. We use the hyperparameter values that yield the best results.

3.NEURAL NETWORKS

As input to deep learning models, the vector arrays were transformed from two-dimensional vectors to three-dimensional vectors.

3.1 BLSTM

The Bidirectional Long Short-Term Memory model outperforms the LSTM model by a little margin. The decay kinematics of top-quark pairs created in high-energy proton-proton collisions are presented as a probabilistic reconstruction utilizing machine learning. The four-momenta of the two top quarks created in the hard-scattering process are inferred using a deep neural network with a Bidirectional Long Short-Term Memory (BLSTM) at its core. It has a loss rate of 0.31, whereas LSTM has a loss rate of 0.32. The accuracy rate of the BLSTM is 82 percent, which is greater than the accuracy rate of the LSTM model, which is 81 percent. After Epoch 33, the BLSTM model begins to overfit, comparable to the LSTM model, which begins to overfit after Epoch 31. Beyond epoch 30, the learning rate was changed in the next model to reduce the overfitting impact.

3.2 Hyperparameters

To find the best hyperparameters for our deep learning models, we use two distinct base models. Long short-term memory (LSTM) is used in both base models, which has 256 neuron units and two hidden layers with a batch size of 32 for 100 epochs. The output layer was given the "Sigmoid" function. The binary cross-entropy loss is

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 11 Issue 02, February-2022

chosen as the loss function since it can accelerate the learning and convergence process. The "Adam" optimizer is used in the models, and the learning rate is 0.001. Base model 1 employs the "ReLu" activation function for the hidden layer, while base model 2 uses the "Tanh" activation function.

After Epoch 31, Base Model 2 begins to be overfitted as the validation loss hits a minimum and remains constant while the training loss decreases. Base model 2 ("Tanh") performs somewhat better than base model 1 ("ReLu") because it has a greater accuracy rate and a lower loss rate. As a result, we've decided to build our neural networks using the "Tanh" activation function.

After Epoch 40, the validation loss begins to increase while the training loss continues to decrease, indicating that base model 1 is overfitted. The greatest validation accuracy is 81.10 percent with the loss rate of 0.34 \s(Epoch 40). (Epoch 40).

3.3 Gated Recurrent Unit

Long Short-Term Memory and GRUs are quite similar. GRU, like LSTM, controls the flow of information through gates. In comparison to LSTM, they are quite new. This is why they outperform LSTM and have a more straightforward architecture. During this phase, the neural network was used to continuously build the model using GRU and Bidirectional Recurrent Neural Networks. Because GRU lacks an explicit memory unit, as well as a forget and update gate, it trains the model faster than LSTM, albeit at the expense of accuracy. The GRU has a simpler design than the LSTM, which minimizes the number of hyperparameters. The BGRU model outperforms the LSTM model by a little margin.

Another intriguing feature of GRU is that, unlike LSTM, it lacks a distinct cell state (Ct). It only has one state: hidden (Ht). GRUs are easier to train because of their simpler architecture. It has a little lower accuracy rate (validation) than the BLSTM model, at 81 percent.

3.4 Vulnerability Detection

Our BLSTM model has 256 neuron units and two hidden layers, each with a batch size of 32 epochs. The model's accuracy curve is depicted in Figure 8. With an accuracy rate of 81 percent, the model performs well. In discovering insecure code, the BLSTM model has an F1 score of 75% and a recall of 95%. It has a specificity of 75%. The macro average and weighted average results are nearly identical. This suggests that the model does a good job of distinguishing between susceptible and non-vulnerable code.

3.5 Threshold Value for Binary Classification

The previous models applied a 0.5 threshold to outputs in order to forecast the target class. If the outputs are less than or equal to 0.5, for example, they are categorized as class 0 outputs (Non-vulnerability functions). According to the prior models' performance, the models have some Type I and Type II mistakes, thus the threshold value must be adjusted to minimize the cost of Type 1 and Type 2 errors.

In some cases, such as when using Precision-Recall Curves and ROC Curves, the optimal threshold for the classifier can be calculated directly. A grid search can be used in different situations to fine-tune the threshold and discover the best value.

To construct the Neural Network outputs in the last layer, the Sigmoid activation function was applied to all preceding models. The outputs are decimal numbers ranging from 0 to 1, indicating whether the outputs are more likely to be categorized as class 0 or 1 depending on the threshold.

4. DEEP LEARNING

Wu et al. use the sequences of C library function calls as the dataset to create neural network models to forecast vulnerabilities [10]. They turn each sequence of C library function calls into a list of word tokens, then train three neural network models with the vectors: CNN, LSTM, and CNN-LSTM. Their analysis reveals that the neural network models outperform the MLP utilized by VDiscover by a significant margin.

VulDeePecker trains neural networks to discover vulnerabilities using code gadgets, which are computer statements that are data or control reliant on one other. It retrieves relevant program slices as code gadgets and turns them into vectors using word2vec, focusing on library and API function calls. It then uses the vectors to create BLSTM models for vulnerability identification. VulDeePecker considerably lowers false positives when compared to previous machine learning-based studies, such as VulPecker[5].

5. MACHINE LEARNING

Yamaguchi et al. extract information relevant to API function calls for all functions of a target program from the target program's source code, convert the extracted information into vectors, and use principal component analysis (PCA) on the vectors to identify the dominant API function usage pattern for each function in the target program. It predicts vulnerable functions by comparing the usage pattern of functions to that of a known vulnerable function. An example of a vulnerability is used by the authors to demonstrate the usefulness of the strategy.

6.CONCLUSION

In the field of cybersecurity, software vulnerability has long been an important but critical research topic. Machine learning (ML)-based approaches have recently sparked increased interest in software vulnerability detection research. The detection performance of existing ML-based approaches, on the other hand, has to be improved. The first is code representation for machine learning, while the second is a class imbalance between susceptible and nonvulnerable code.

This study describes how we used deep learning to find vulnerabilities in binary code. The SARD dataset contains binary code compiled from C/C++ programs, and we leverage the semantic information on assembly instructions as features to train deep learning models. The BLSTM model outperforms the BGRU model in our evaluation. It detects vulnerabilities with an accuracy of 81 percent. The models perform effectively in categorizing both susceptible and nonvulnerable code, as evidenced by the close similarity of macro average and weighted average finding.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 11 Issue 02, February-2022

REFERENCES

[1] 533 million Facebook users' phone numbers and personal data have been leaked online 2021. https://www.businessinsider.com/stolen-data-of-533-millionfacebook-users-leaked-online-2021-4.

[2] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J Schwartz, Maverick Woo, and David Brumley. 2014. Automatic exploit generation. Commun. ACM 57, 2 (2014), 74–84

[3] 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, 85âĂŞ96. https://doi.org/10.1145/2857705.2857720

[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), 1–36IEEE, August 2016 [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7605062/?casa_token=gYCfDKXeuQoAAAAA:M7ZVVdmCZIDah8vHkHPf4_WJKT6_qw_A0NYJHFbg-LZt1CbmMvMOJox-EV2Sm_ZDEChddIY6yuObfr8

[5] 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, 201âĂŞ213. https://doi.org/10.1145/2991079.2991102

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

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

[8] 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.