Deep Learning: a Replication of "ReGVD: Revisiting Graph Neural Networks for Vulnerability Detection"

Peter Haupt, Aisja Thijssen, David Pennekamp
Jheronimus Academy of Data Science, Den Bosch

Abstract

The ReGVD model is a relatively new graph neural network model made to find vulnerabilities in different coding functions. This replication research paper attempts to replicate the results of the original paper in which the ReGVD model was described. The authors are unsuccessful in replicating the exact same environment and in replicating the exact same results. The lack of replicability of the original results is cause for caution.

Introduction

Nguyen et al. (2022) developed a new technique for vulnerability detection in code. They called their new technique ReGVD (Revisiting Graph neural networks for Vulnerability Detection). They wanted to use the techniques of graph neural networks (GNNs) in combination with residual connections to improve performance. They test their method relative to existing benchmark models in the field of code vulnerability detection and find that they can reach the highest performance on the CodeXGLUE benchmark dataset.

The aim of this paper is to replicate the results of Nguyen et al. (2022) and see if it is possible to produce the same results using the same dataset. The paper makes an attempt at literal replication. In the end, it was not completely possible to replicate the exact environment and results. The results of the best configuration of the ReGVD still outperform the best alternative presented in the original paper. However, other configurations now perform worse.

These results tell us that the ReGVD model might not be as good as it seems to be in the original paper. However, the model still outperforms some benchmark models discussed in the original paper. The fact that the team was not able to fully replicate the original results raises questions about the model's credibility. The fact that the model performance changes based on the machine it was trained on makes the model look less robust. However, the difference was not a dramatic one so the model is still quite reliable. The method can therefore be used in practice but should be used with caution.

The next section of this paper will provide the needed background knowledge for someone to understand the contents of this replication research report. This is followed by a section detailing the procedures of the original paper. Then, a section describing the replication study conducted follows, along with the results of the replication. This paper closes with a conclusion in which the possible application contexts are described and in which the success of the replication study is discussed.

Background

The background knowledge required to understand this study includes how graphs are encoded for graph neural networks. Other relevant knowledge includes how text is tokenized and how neural networks are trained, we assume the training procedures of multilayer perceptrons to be known.

The CodeXGLUE dataset used in the original paper (Nguyen et al., 2022) contains functions that are labeled as vulnerable or not. These functions have different lengths (as one would expect in code). However, for a neural network, we must define a particular structure (with a certain number of input neurons). To get all the information from the code into a graph, and feed the information from that graph into a neural network, encoding is done. First, the raw text is tokenized by a pre-trained programming language model. The graph is built based on the proximity between tokens. If tokens co-occur within a certain window length, an edge is created between the two. This graph is still in a non-standard format. However, we can give each node certain information about itself (here, the node features come from the token embedding layer of a pre-trained programming language). In the next step, the embeddings of nodes are mixed based on edges. For example, if node 1 and 2 are connected, then after one step the node features between 1 and 2 are mixed based on some principles defined by GCN or GGNN. This is repeated for a number of steps, which spreads information about nodes throughout the graph (e.g. in step 2, nodes 2 and 3 may exchange information again, which now also includes information about node 1, which was connected to node 2, but disconnected from node 3). The node features have a certain fixed structure (e.g. 32 pieces of information). As such, the pooling of these node features across all nodes will combine information from all node features into an output that has a fixed length, related to the embedding size (e.g.

32 when using an operation like averaging, or 64 when concatenating). This output is then used as the input layer for the neural network.

The neural network consists of standard fully connected layers with neurons. The number of neurons per layer and the number of layers are subject to parameters. Standard back-propagation is used to tune the parameters in the neural network.

Target Study Overview

The target domain of the study is the domain of cyber security, as well as the domain of neural networks. In recent years there have been multiple papers published that work on the same CodeXGLUE dataset (Coimbra et al., 2021; Tang et al., 2023). The paper proposes a novel approach to vulnerability detection and demonstrates that it outperforms state-of-the-art models (Lu et al., 2021).

Techniques used

The use of graph neural networks is not new in the field. None of the individual steps in the process are novel, but the combination of techniques is what distinguishes it from previous studies. In the first place, the authors build on existing PPLM techniques for tokenizing the code. These tokens are then used to build a graph by looking at the distance between tokens in the code and building the network structure. Existing GNN models (GCN and GGNN) are then used to embed the graph. Lastly, well-known pooling methods (sum- and max-pooling) are then combined to feed into a single hidden layer, which produces a classification output.

The replicated study uses a Graph Neural Network (GNN) combined with a pre-trained programming language model (PPLM) to detect vulnerabilities in code functions. In simple terms, the PPLM splits the function code into tokens. These tokens are used to build a graph based on their distance. This graph is then fed into the GNN. The GNN then learns an embedding for the graph. Finally, this graph embedding is used to predict whether vulnerabilities are present in the code.

In more technical terms; the raw text sequence is split into tokens by the PPLM. Then, there are two methods to construct the adjacency matrix: unique token-focused construction and index-focused construction. In the unique token-focused case, the same tokens are collapsed into one node (e.g. there could be one node for `if`, one node for `for`). The index-focused method yields a node for every token (e.g. there could be 5 nodes for `if` in case of 5 if statements in the code). In both cases, a link exists between two nodes if the tokens represented by the node occur within a specific sliding window.

The tokens returned from the PPLM are then used to provide the initial node features. These node features are updated using a GNN. In the paper Graph Convolutional Networks (GCNs) and Gated Graph Neural Networks (GGNNs) are proposed as possible mechanisms. Moreover, the paper suggests using residual connections to pass information from early layers directly to later layers.

The last step in the process of predicting the outcome variable is the single dense hidden layer. To get the information into a fixed-length vector to feed to a fully connected layer, the authors propose a combination of sum pooling and max pooling over a vector e_v, calculated as the element-wise multiplication of a soft attention mechanism over the nodes and the updated node features, times their weight plus a bias. The proposed strategies for combining these pooling results are summing, multiplying, or concatenating. This results in a single fully connected layer with a softmax output layer. To train this network, the cross-entropy loss function is used.

Validation

The paper tries to show the superiority of their newfound method by comparing it to the current methods available for function code vulnerability detection. Using a benchmark dataset CodeXGLUE the paper trains multiple different configurations of their own model and compares the performance in terms of accuracy to other existing models. The models they compare their ReGVD model to are BiLSTM (Hochreiter and Schmidhuber, 1997), TextCNN (Kim, 2014), RoBERTa (Liu et al., 2019), CodeBERT (Feng et al., 2020), GraphCodeBERT (Guo et al., 2021), and Devign (Zhou et al., 2019). In the different configurations of their own model, they vary the graph construction method between unique-token focused and index-focused, where the sliding window size is a hyperparameter. They also vary between GGNNs and GCNs, here the number of hidden layers is again a hyperparameter that is tuned. another parameter is the node feature initialization where they use the token embedding layer of either CodeBERT or GraphCodeBERT. The final hyperparameter being tuned is the learning rate. The graph construction also varies from the Devign algorithm where 4 different configurations are tried.

Optimization To find an appropriate embedding for the graph, well-known GNN methods are used. The training of the hidden layer uses back-propagation with the Adam optimizer to optimize the cross-entropy. The loss function used is the cross-entropy for binary classification, also known as the log-loss. For evaluating the performance of the model, accuracy is used.

Data As mentioned before, the dataset used in the paper being replicated is called CodeXGLUE (Lu et al., 2021). This dataset contains different coding functions that are already labeled in terms of vulnerability allowing the authors to use it and calculate the accuracy of their methods. The dataset is split into a train, validation, and test set where we only compare the different methods based on the test set. This dataset is seen as a benchmark dataset for code vulnerability detection and the train-validation-test split is done by Lu et al. (2021) in advance so the methods of different papers can be compared without having to re-execute every single method. The dataset was randomly split into 80% testing, 10% validation, and 10% testing (Lu et al., 2021).

Limitations

The authors do not mention the limitations of their study. The team has identified some limitations; the paper mentions using the best model checkpoint on the validation dataset, rather than an average over multiple runs and multiple validation sets, such as with cross-validation. Moreover, the ac-

Set	Vulnerable	Not vulnerable
Train	10018	11836
Validation	1187	1545
Test	1255	1477
All	12460	14858

Table 1: Overview of label distribution of dataset

curacy of the model does not mean much. It would be important to also look at the types of errors made. For example, a model that only predicts 0 (not vulnerable) for this dataset, would already achieve a 54.2% accuracy. In this replication research, we check the claims of the researchers and focus on the accuracy claims made.

Replication Study Definition

Decision Tree of the Replication Research Design Decisions

The decisions of the replication research design are visualized in Figure 1. The decisions were made in five steps.

First, it was decided to perform an attempt at a literal replication of the original study (Nguyen et al., 2022). Therefore, it was decided to only use the original dataset of the study which was collected by Lu et al. (2021).

Second, in an ideal literal replication study, all model configurations of the original study would have been replicated. Due to limited computational resources, it was decided to replicate the GCN and GGNN which had the best performance in the original study.

Third, due to computational resource constraints, it was decided to replicate the study on a local Macbook with an M1 chip and a local Windows machine. The original study was performed with Python 3.7. For Windows, this version was still available. For the Macbook with an M1 chip, the lowest compatible Python version was 3.8. Therefore, Python 3.8 was used on the Macbook with an M1 chip.

Fourth, to train the GCN and GGNN minor changes in the Python code of the original study were necessary. More specifically, the import statement for 'scipy' needed to be changed.

Dataflow Diagram of the Replication

The dataflow diagram can be found in Figure 2. The data was processed in the following five steps. First, the data was loaded from the original CodeXGLUE dataset (Lu et al., 2021). This dataset contains 27318 coding functions from open sources projects. Second, there was no need for preprocessing the data, because it was already available in the required form. Third, the data were analyzed and descriptive statistics were evaluated. Most importantly the dataset is only slightly imbalanced at 54% towards non-vulnerable (Table 1). Fourth, the GCN and GGNN were trained with this dataset on different machines. Fifth, the results could only approximately be replicated. Notably, the results were slightly different between different devices. On the same device, it was possible to replicate the exact same results.

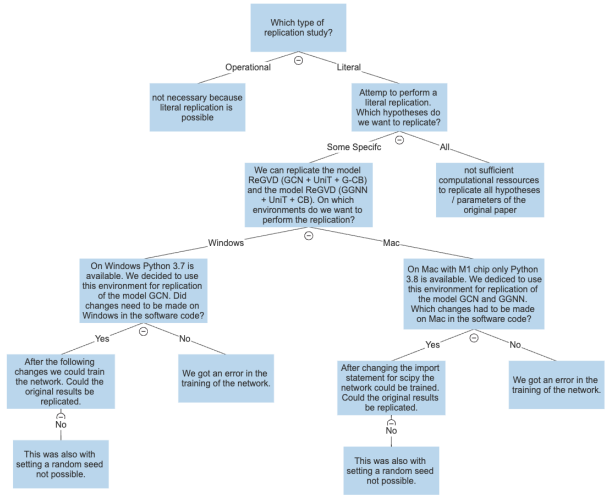

Figure 1: decision tree of the research decisions

Replication Results

An overview of the replication results can be found in Table 2. The replicated accuracies are on average one percentage point below the original results. Additionally, to the original study also a confusion matrix was generated for the results of this replication study see Figure 3. Based on the confusion matrix the recall and specificity can be calculated. The recall is 44.14% and the specificity is 78.06%. For this application, it is much more important to avoid false negatives than false positives. Therefore, a high recall is much more important. A recall of 44.14% is relatively low.

The replication on different hardware devices led to the following results. When using the same Macbook, the results for ReGVD (GCN) could be exactly replicated in a second training of the model. But on a Windows machine where the code was executed using a GPU, the results were different. The differences between the Macbook with CPU and the Windows machine with GPU could be due to differences in the implementation of the random seed on different devices (GPU versus CPU) and the usage of multiple cores. Several random seed functions for different components of the training procedure have been implemented by the original study. But, for example, the seed for the Cuda environment was only set for one GPU and not for multiple GPUs which would have been possible with the function TORCH.CUDA.SEED_ALL. Other explanations for the differences between the original study and this replication are differences in several Python packages and differences in the used hardware. Unfortunately, the used Python packages have only been described vaguely by the authors of the original study and the hardware has not been described at all.

The results identify three problems with the initial study. First, the results are on average one percentage point lower than the original study. Second, the random seed section in the Python code is not implemented correctly. We found that it is not robust to run on different devices with or without GPU. Third, the initial study did not report a confusion ma-

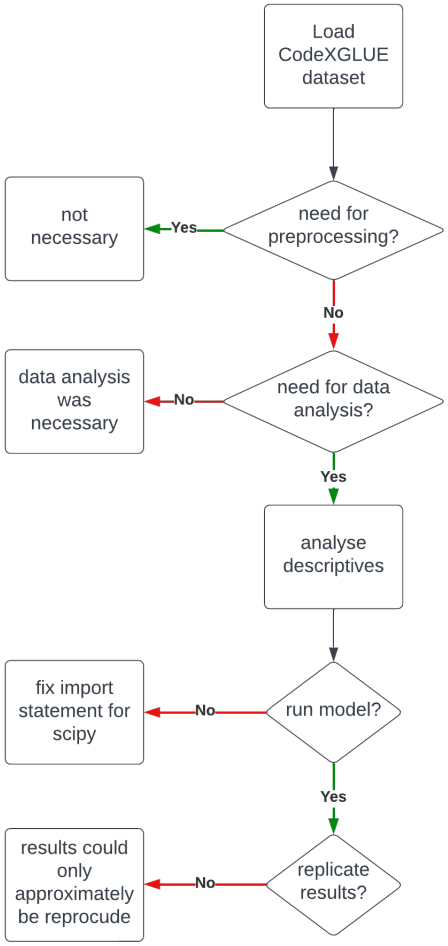

Figure 2: dataflow diagram of the replication

trix and a recall score. In our replication, the recall score was low which is problematic for detecting vulnerabilities in Python functions.

Threats to Validity

The validity of the original and the replication study is threatened by several aspects.

First, the replication study could not exactly replicate the results of the initial study. Especially, the exact causes for the differences between the replication study and the original study could not be determined. A major cause for this is the lack of documentation in the GitHub repository of the original study. Many details like exact Python packages and results from the initial training procedure are not documented.

Second, the relatively high accuracy suggests that the model is capable to identify correctly vulnerable functions. But the analysis of the confusion matrix in the replication study reveals a low recall. Therefore, the model can in many cases not correctly identify vulnerable functions.

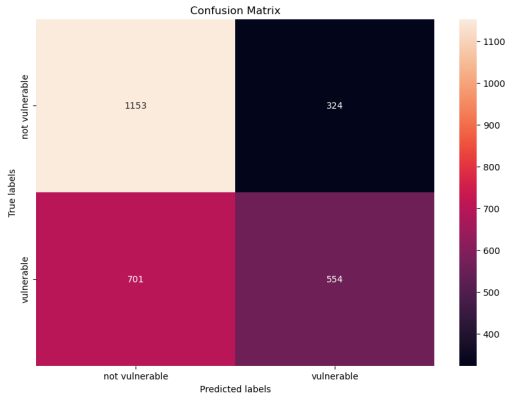

Figure 3: confusion matrix

Model	Original Study	Replication
ReGVD (GCN)	63.69	62.48
ReGVD (GCN + Idx)	62.70	61.71
ReGVD (GGNN)	63.62	62.04

Table 2: Overview of the Replicated Accuracies

Conclusions

All in all, the replication was relatively successful, the team managed to run the configurations of the ReGVD they wanted to run and the results were quite similar to the ones in the original paper, even if they were not exactly the same. The team has learned about the application purposes of this method and the different contexts in which it can be applied.

In this section, the possible application contexts are explained as well as to what extent the replication of the original paper (Nguyen et al., 2022) was successful.

Application Context

The original paper intended for this method to be used in vulnerability detection for code. More specifically the ReGVD model was tested on a dataset with functions that were labeled as vulnerable or not vulnerable. Hence the ReGVD model can be used in coding and this model can help identify some vulnerabilities but is unlikely to capture all vulnerabilities. This falls under the domain of cybersecurity.

The replication described here also focuses on this code function vulnerability as it uses the same dataset. The cybersecurity context is the context where the model is proven to be effective. The ReGVD outperforms all existing benchmark models for this given problem on the CodeXGLUE dataset.

Another factor that makes the ReGVD model successful in this particular context is the availability of pre-trained programming language (PL) models. These allow for better initialization of node features for the input graph representing the code.

Companies could implement this method in practice and use it to test the code written in development that will be used in production to assist developers to the company less

vulnerable to attacks from outside of the company. However, the performance of the model on the CodeXGLUE dataset is not very satisfactory yet. When companies decide to use this model it should be done with caution.

The method was created specifically to work in this context, that does not mean it will not work in different contexts but it has not been tested in different contexts. In essence, what the technique does is text classification. It takes as an input a graph created based on a textual object (in this code a coding function) and using a labeled dataset creates a classification model predicting if the input belongs to one class or the other. As long as we can transform another textual input into a similar graph it should be able to build a classification model for this as well.

However, it may be less straightforward to create these graphs for alternative methods. The sliding window technique could work for other types of texts as long as preserving the nature of a word is not needed for the model to make a good prediction. Additionally, for the feature initialization of the nodes in these networks, it could be beneficial to use a pre-trained model similar to CodeBERT. The authors use the token embedding layer of CodeBERT or GraphCodeBERT for node feature initialization. They say the availability of these pre-trained models helps the ReGVD reach such high performance. If a similar method for node feature initialization is not possible in a different context the performance of the model might decrease. Therefore applying the method in different fields might require the training of language models specific to the text being used and should be approached with caution.

Examples of a different context where the ReGVD could be used are vulnerability detection in legal documents, provided that it is possible to create an insightful graph from a multi-page legal document, or perhaps even the classification of emails.

The original paper has been published quite recently (a little over 1 year ago). Potentially new papers will show up applying the same principles to a different context. If these papers offer good alternatives similar to CodeBERT for the initialization of node features the method could also be more easily applied in these different contexts.

Replication Success

As mentioned above the intention of the team was to do a literal replication of the original report by Nguyen et al. (2022). Due to some issues already shown in the decision tree, not all aspects of the original research were replicated. The versions used for the different Python packages are newer than the ones listed in the documentation of the code accompanying the paper. In this documentation, it was mentioned to use Python 3.7, PyTorch version 1.9, and transformer version 4.4. The first issue was that Python 3.7 was not available on the Macbook used to run most of the simulations. After having decided on using Python 3.8 instead. The team tested the random seed in numpy to make sure the different python versions would give the same results with the same random seed, this was the case. The next problem was that the documentation used the incorrect name for the transformer package which was actually called transformers. Finally, the scipy package was needed for the code to run but was not included in the documentation so the version used by the authors of this package is unclear.

Due to the use of Python 3.8 and the compatibility of the python packages the team ended up using newer versions of the different packages. In the end, the team used version 1.13 of the PyTorch package instead of 1.9 and version 4.26 from transformers instead of 4.4. For scipy, the package with version 1.10 was used in the end.

No big changes had to be made to the code in order for it to run properly. The only issue encountered was a scipy import issue caused due to the use of a too-new version of the scipy package. One of the scipy functions imported had changed location within the package so this import statement had to be altered from `from scipy.sparse.linalg.eigen.arpack import eigsh` to `from scipy.sparse.linalg import eigsh`.

For the one run on the windows machine, the team was able to use Python 3.7 but was required to use newer versions of the transformers and PyTorch packages to allow the code to run on GPU and to prevent an error that occurred when using the transformers package with version 4.4 (the code gave a package incompatibility error as the package version was too old). So even when using Python 3.7 other issues still changed the environment from the one used in the original paper. However, the older python version made it so the scipy import still worked as in the original code. The same package versions were used as on the MacBook only scipy 1.7 was used instead of 1.10.

Other than the issues with the setup of an environment and the change of one import statement, the team could execute the code and change the configuration of the trained ReGVD model. Many options were left as input parameters allowing for replication with limited trouble.

The data used for the replication was the same as used in the original paper. This is a collection of different coding functions often used as a benchmark for vulnerability detection in code named CodeXGLUE (Lu et al., 2021). This dataset contains labeled data of many different coding functions.

As the same data was used as in the replicated paper there should not be a difference in performance compared to that paper. The methods used are the same and the data used is the same.

Seeing as the original paper tries to create a new benchmark vulnerability detection method it makes sense to compare based on a given benchmark dataset.

Given that the team would be able to find a data source with code functions that were either already labeled or could easily be labeled the team could have done operational replication as well. However, due to time restrictions, this was not possible in the current setting. Similarly, for instrumental replication, given sufficient time to create an implementation of the model the team could have attempted to reproduce the methods using for example Keras instead of PyTorch. Constructive replication would also have been possible given sufficient time to implement it as this would require a lot of work.

All in all, all types of replication are possible, the team decided to replicate the procedure followed by the original paper's authors as closely as possible largely due to time constraints and the availability of the source code used by the authors.

References

Coimbra, D.; Reis, S.; Abreu, R.; Păsăreanu, C.; and Erdogmus, H. 2021. On using distributed representations of source code for the detection of c security vulnerabilities.

Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D.; and Zhou, M. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, 1536–1547. Online: Association for Computational Linguistics.

Guo, D.; Ren, S.; Lu, S.; Feng, Z.; Tang, D.; Liu, S.; Zhou, L.; Duan, N.; Svyatkovskiy, A.; Fu, S.; Tufano, M.; Deng, S. K.; Clement, C.; Drain, D.; Sundaresan, N.; Yin, J.; Jiang, D.; and Zhou, M. 2021. Graphcodebert: Pre-training code representations with data flow.

Hochreiter, S., and Schmidhuber, J. 1997. Long Short-Term Memory. *Neural Computation* 9(8):1735–1780.

Kim, Y. 2014. Convolutional neural networks for sentence classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 1746–1751. Doha, Qatar: Association for Computational Linguistics.

Liu, Y.; Ott, M.; Goyal, N.; Du, J.; Joshi, M.; Chen, D.; Levy, O.; Lewis, M.; Zettlemoyer, L.; and Stoyanov, V. 2019. Roberta: A robustly optimized bert pretraining approach.

Lu, S.; Guo, D.; Ren, S.; Huang, J.; Svyatkovskiy, A.; Blanco, A.; Clement, C. B.; Drain, D.; Jiang, D.; Tang, D.; Li, G.; Zhou, L.; Shou, L.; Zhou, L.; Tufano, M.; Gong, M.; Zhou, M.; Duan, N.; Sundaresan, N.; Deng, S. K.; Fu, S.; and Liu, S. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *CoRR* abs/2102.04664.

Nguyen, V.-A.; Nguyen, D. Q.; Nguyen, V.; Le, T.; Tran, Q. H.; and Phung, D. 2022. ReGVD: Revisiting Graph Neural Networks for Vulnerability Detection. *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*.

Tang, W.; Tang, M.; Ban, M.; Zhao, Z.; and Feng, M. 2023. Csgvd: A deep learning approach combining sequence and graph embedding for source code vulnerability detection. *Journal of Systems and Software* 199:111623.

Zhou, Y.; Liu, S.; Siow, J.; Du, X.; and Liu, Y. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks.