**CSEE 5542 Project Report**

**Project Title:**

Vulnerability Detection with Control Flow Graph Convolutional NN

Team Overfitters Members

Keenan Flynn; kpfxn8@umsystem.edu

Mustavi Islam; mizmc@umsystem.edu

Ruoyao Xiao(Eric): exdxg@umsystem.edu

Hoyun Yoon; hynkz@umsystem.edu

[Presentation Video](https://youtu.be/RfpPda57iFs)

[Code Explanation Video](https://youtu.be/cefnVgKU6oQ)

Aim

In this project, we wish to use Graph Learning to analyze the structure and control flow of a Java program. Using Control Flow Graphs, we will use Deep Learning techniques to classify Java program based on the type of code vulnerability or Common Weakness Enumeration (CWE) is present in the program if any.

Problem

Code vulnerabilities are responsible for on average 3.86$ million dollars per breach. This is a staggering number and is enough to sink small companies. If programmers are able to decrease the number of vulnerabilities in code, making it harder for malicious entities to gain a foothold into modern code system. Detecting these types of vulnerabilities or CWEs is a job for Deep Learning.

There are many current classification algorithms which use NLP to generate prediction judgements from a text based input. The meaning of this text is converted into vectors which are processed and classified by certain linear methods such as LSTMs or other recurrent networks. The efficiency of these models is very good for ordered natural language. However, when entering a non-linear sequence of text, such as an if-else code block, linear classification models fail to judge the correct semantics. Coding languages in general are non-linear, requiring a different solution which can analyze the structure of programming.

Solution

Our solution uses Control Flow Graphs to deconstruct and re-represent institutional Java programs in a Graph format. Control Flow Graphs, or CFGs, are an abstract representation of code blocks which are typically used by a compiler. CFGs follow the typical graph structure and contain nodes and edges. Within a CFG, each line of code is a node and all possible paths from that node are represented as edges. Take the example below in figure 1. If cond S2 is True, S5 and S6 will never be executed. This is where a linear approach fails but a graphical approach succeeds.

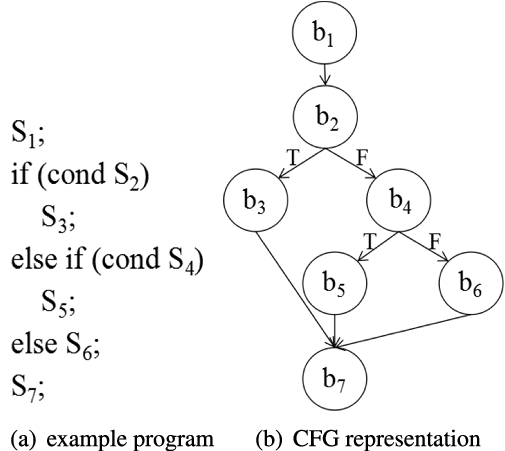


Figure 1

If we can use the form of a Knowledge Graph to express these CFGs, we can implement Deep Learning graphical algorithms which will have the correct semantical meaning of the text input. We propose the use of 3 different Deep Learning algorithms to tackle this problem. We will use Graph Convolutional NNs as our baseline which is the most cited type of architecture in the realm of graph learning. We will compare this against Graph Attention and Transformer Convolutional networks.

## Abstract

Diagram

Description automatically generated

Figure 2

Figure 2 shows an abstract representation of our project.

The input is Java Methods which have been gathered on the web. The CFG converter converts these files into Control Flow Graphs. These 10,000+ CFGs, formatted as .dot files, are our dataset and are split into Training, Testing, and Validation subsets. These CFGs are preprocessed and split into 3 subfiles: the edge file, label file, and node file. The node is then embedded using Sentence Transformer from Bert. Before entering the Graph Neural Networks, they are recombined into a non-readable .pt file. The Graph Attention NN, Graph Convolution NN, and Transformer Convolution NN are built and trained and then evaluated with the validation dataset. Our metric in this project is F1 score.

## Dataset

For this case of multi-classification, we selected 43 common problems in CWE as the problematic data set. We selected common programs found in the stone-soup project and JDK as non-vulnerable data.

To represent the code samples as Control Flow Graphs, we needed to pass them through a Control Flow Graph Generator. This generator was contributed from GitHub user Andrei Rimsa who is a professor at CEFET-MG. The team treated this CFG generator as a blackbox and it can be found on his [GitHub](https://github.com/rimsa/CFGgrind).

The gathered Java programs were run through the CFG generator which produces Control Flow graphs in the form of .dot files. These files are editable in any word editor and include a list of nodes and directed edges.

After converting all Java code to CFGs, we split the data into training, validation, and test sets. Figure 2 shows the statistics of this split.

Text, letter

Description automatically generated

Text, letter

Description automatically generated

Text, letter

Description automatically generated

Figure 3

A picture containing graphical user interface

Description automatically generated

Nodes

Edges

Figure 4

To express the Java programs in the form of knowledge graphs in which the control flow represents the program structure, each Java function gets converted to a subgraph. Each of these subgraphs has a label which is the name of the program. Each of these labels belong to a large category which describes the codes vulnerability (such as CWE-510 or non-vulnerable). Each internal line of code represents a node. Figure 3 shows the .dot file along with its graphical representation of one of our Java samples.

## Implementation

The implementation of this project is divided into 4 categories: Preprocessing, Model Building, Training, and Evaluation.

1. Preprocessing

To preprocess our .dot files, we use the Python library pydot to read and split the data. The .dot files are split into three distinct files: Edges, Nodes, and Label. The edges are converted into a .adj file which contains the adjacency relationship between all nodes. The second type of file is the .gf file contains the label of each CFG (such as CWE-510). These labels are one-hot encoded. There were 44 distinct labels in this project. The third type of file is the .ndf file. This file stores the information of the nodes. Each node, which represents a line of code, is embedded using a pretrained model which is the Bert Sentence Transformer. If the CFG has n nodes, the .ndf file will be an array with n rows and 768 columns. This is because the Sentence Transformer converts sentences into 768-dimensional arrays.

1. Model Building

We created 3 types of Neural Networks in this project: Graph Convolutional NN (GCNN), Graph Attention NN (GAT), and Transformer Convolution NN (TransConv NN). The GCNN has been the most cited algorithm in the domain of Graph NNs. For this reason, we will use the GCNN as a benchmark and compare it against the other 2 types of models. Our GCNN consists of 3 GCNConv layers with a Relu activation, an Average Pooling layer, Dropout layer, and a Linear layer for the output. The GAT NN contains 2 GATConv layers with Relu activation, a Pooling layer, Dropout layer, and a Linear layer for the output. The TransConv NN uses 3 TransformerConv layers with a Relu activation, an Average pooling layer, Dropout layer, and a Linear layer for the output. Each model uses the Adam optimizer and the Multi Label Soft Margin Loss function

1. Classification and Training

Before training our neural networks, we first split the data into 3 subsets: training, validation, and testing. The ratio is 8:1:1 with most of the data belonging to the training subset. This split was done in the k-fold Cross-Validation fashion with 4 folds. k-fold Cross-Validation allows the model to train on more data but adds some dropout between passes/epochs. We also need to combine our 3 files into a file that is readable by pytorch. We use a class called CFGDataset, which was inspired by the TUDataset of pytorch geometric. The TUDataset class is a collection of benchmark datasets for graph classification and regression. We pass the dataset into an constructor of the CFGDataset class which then creates .pt files.

1. Evaluation & Result

We divided the experimental results into 2 parts.

The first part of the results is the F1 score metric. The confusion matrix for each group of experiments was used to calculate the F1 score. We ran successive experiments searching for optimal hyperparameters. We adopted a naming format for our results files which is named after the hyperparameters used. An example results file looks like this: "multi\_final\_1\_GATConv\_0.5\_1024\_0\_0.0005". The “1” indicates the number of folds used in K-fold Cross-Validation. “0.5” represents the threshold. “1024” represents the number of hidden channels. “0” indicates the weight decay. “0.0005” represents the learning rate. More examples can be seen in Figure 5. The results file for each run were exported to VisualDL for visualization. Figure 4 shows this visualization. Figures 6, 7, and 8 show the F1 score for the best run of each model. The traditional benchmark GCNN is worse than the other two.

Chart, line chart

Description automatically generated Text

Description automatically generated

Figure 4 Figure 5

Graphical user interface, application

Description automatically generated

Figure 6 (GCNN F1 Score)

Graphical user interface, application

Description automatically generated

Figure 7 (GAT F1 Score)

Graphical user interface

Description automatically generated

Figure 8 (TransConv F1 Score)

The second part of the evaluation is the accuracy representation of each class (CWE).

This is shown as a readout of precision, recall, F1 score. Figures 9, 10, and 11 are the calculation results of the accuracy of GCNN, GAT, and TransConv for a single category.

Table

Description automatically generated Table

Description automatically generated

Figure 9 (GCNN) Figure 10 (GAT)

Table

Description automatically generated

Figure 11 (TransConv)

## Conclusion

Graph Learning can successfully be used to gain knowledge about code vulnerabilities. All three of our algorithms showed accuracy and F1 score around the ~90% range, with the GAT and TransConv networks performing better than our GCNN benchmark. Graph Learning is a successful classifier because it can evaluate the non-linear structure of code.

In future work we would plan to pursue these items:

* Implement a Grid Search to find the optimal values for our hyper-parameters.
* Adding and removing layers from the networks.
* We would like to explore other convolutional algorithms. We found that even though GCNN is the most cited algorithm in Graph Learning, it is not necessarily the best.
* Change the approach of our word embedding by implementing and testing other sentence embedders.

## Resources

<https://www.privateinternetaccess.com/blog/hacking-the-world-part-4-the-cost-and-future-of-hacking-plus-safety-tips/#:~:text=Breaches%20cost%20companies%2C%20on%20average,record%20was%20%24150%20in%202020>.

<https://www.researchgate.net/figure/shows-the-procedure-of-the-control-flow-graph-with-three-conditional-statements-Assume-a_fig11_331460321>

<https://theaisummer.com/gnn-architectures/>

<https://chrsmrrs.github.io/datasets/>