Declaration

I declare that the work in this dissertation was carried out in accordance with the Regulations of the University of Bristol. The work is original, except where indicated by special reference in the text, and no part of the dissertation has been submitted for any other academic award. For all ideas taken from other sources (books, articles, internet), the source of the ideas is mentioned in the main text and fully referenced at the end of the report. All material which is quoted essentially word-for-word from other sources is given in quotation marks and referenced. Pictures and diagrams copied from the internet or other sources are labelled with a reference to the web page or book, article etc. Any views expressed in the dissertation are those of the author.

Signed: Junfan Huang Date: 2019-09-05

Machine Learning for Automated Vulnerability Detection

Junfan Huang

September 2019

Abstract

Automated vulnerability detection is considered as a non-trivial problem. Most researchers in this field have focused on Nature Language Processing and few have exploited the power of code structure for vulnerability detection. In this paper, we implement vulnerability detection models on buffer overflow, utilising Abstract Syntax Trees from which we extract adjacency matrices and build a convolutional neural network model, obtaining 98.8% accuracy for buffer overflow detection. For further interpretation, we use inductive logic programming from 40 examples (20 example pairs) to guide the system to generate the inference rules.

1 Introduction

Software security is an increasing concern for individuals and businesses alike. Straddling the border between the outside world and our personal computers, web browsers are an important piece of the security puzzle. Over 40% of Mozilla Firefox's Common Vulnerabilities and Exposures(CVE) disclosures have been memory corruption vulnerabilities[32].

However, automated vulnerability detection is a non-trivial task. Machine learning, especially Deep Learning, now plays an important role in a variety of industries, and naturally, security researchers have started to explore ways to solve this problem using the new machine learning techniques; For example, in 2018, Russell et al.[28] introduced a convolutional neural representation-learning approach to achieve vulnerability detection. Li et al.[17] developed a neural networks tool (called VulDeePecker) to detect software vulnerabilities; Choi et al.[6] built a memory network for their artificial source codes which significantly improved their results compared with traditional CNN and LSTM methods.

But every coin has two sides. Deep learning not only requires millions of data but, because of interpolation training process, lacks interpretability. Inductive logic programming (ILP) as a symbolic machine learning technique provides another potential tool for security researchers. In 2003, Moyle and Heasman[20] illustrated that logic programming is a convenient tool for determining the semantics of attacks. This provides a framework for detecting attacks in formerly unknown inputs. This project therefore exploits these two approaches (ILP and Neural Network) for automatics vulnerabilities detection.

After confirming the blueprint for implementing vulnerability detection, the inevitable problem is how to achieve it. Clearly, feature-extraction is extremely important for this task; In other words, the difficulty is to extract the source code into a machine-readable embedding. Motivated by the code2vec paper[3] which extracts information from code using abstract syntax tree (AST), we explored using AST representation for intrusion detection. After that, we extracted the code properties from the code property graph as the input of ILP.

Overall, we explored different means of code representation and continued on retaining source code syntactic structure using AST and Code Property Graph(CPG). We found that adjacency matrices of AST performed well, with 98.8% accuracy. We also explored a new method of code representation by exploiting the Joern tool[11] to obtain ILP database for 20 example pairs.

2 Code Representation

The core issue of this vulnerability detection journey is representing source code, this section is about the code representation techniques we used in our project.

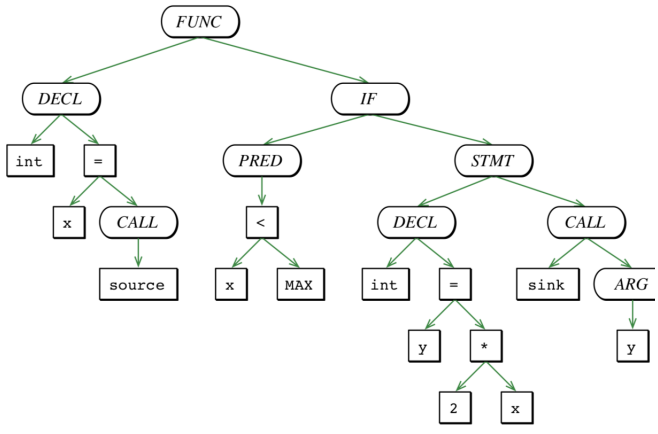
2.1 Related Work

Most work in the field of code representation has explored the natural language processing (NLP) by code tokenisation and token analysis. For example, it is widely used by source code plagiarism detection and source code analysis[9]. However, the information of the source code structure, which sometimes is vital for vulnerability detection, will be lost when using natural language methods. Hence, there is a trade-off for the source code information extraction on structure remaining. We should keep balance between source code itself and code structure as well. As for keeping the code structure, Abstract Syntax Trees are widely used by other researchers[2, 15]

2.2 Abstract Syntax Tree (AST) and other code representation graphs

Abstract syntax tree is a tree representation which retains the syntactic architecture of source code.[30] It is a abstract representation for source code structure with the form of a tree. Every node indicates a construct occurring in the source code and will not contain all the details of the source code. For example, the tree structure could include grouping parentheses information easily, so it is not necessary to include the information in the node. In addition, an if-condition-then expression could be represented using a node with three branches. Usually, the ASTs are generated from traditionally designated parse trees, which are obtained during the compiling process. Once obtaining ASTs, the remaining properties of source code structure could be a useful tool for code analysis.

Here below is an example of an Abstract Syntax Tree, and the corresponding source code is from [31].



(a) AST

```
void foo()
{
    int x = source();
    if (x < MAX)
    {
        int y = 2 * x;
        sink(y);
    }
}
```

(b) Source code

Figure 1: Source code and its AST

As we can see, the AST extracted the operations and values from the source code which could represent the source code function without losing too much information.

Apart from AST, there exist other graphs to represent program, such as control flow graph indicating the code executed order, program dependence graphs[8] showing the predicates and statements dependencies. They more or less retain other properties in source code. It would be helpful to detect vulnerabilities when we consider all the potential code properties together.

2.3 Code Property Graph

Yamaguchi et al.[31], in 2014, proposed code property graph (CPG) which did that by combining the code representation of Abstract Syntax Trees (AST), Control Flow Graphs (CFG) and Program Dependence Graphs (PDG) for source code vulnerabilities detection. CPG based on AST nodes, combining all the properties of AST, CFG and PDG into one graph can be used to generate the background knowledge of our ILP system. Details would be in the vulnerability rule generalisation section.

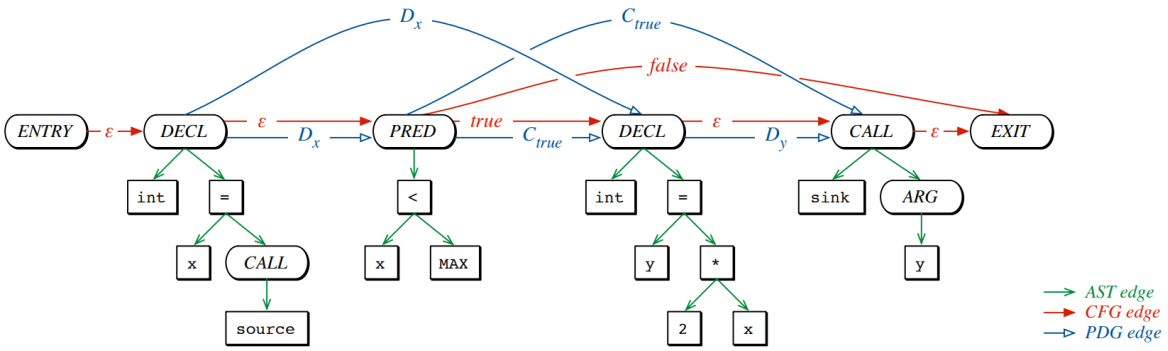


Figure 2: An example of a code property graph

This example above shows that CPG based on AST nodes, connected with CFG and PDG edges, fully retains the most of the source code properties.

3 Graph embedding

Since we decided to use graphs to represent our source code, graph representation is a following challenge before being fed into neural networks. Naturally, we transformed the code representation problem into a graph embedding problem.

3.1 Node2vec

A natural way of obtaining a neural network input is to transform every node in a graph into a continuous feature representation like a vector. Stanford researchers Grover and Leskovec[10] in 2016 proposed an approach *node2vec* to do so. This paper transformed the node feature extraction problem into a optimisation problem to maximise the likelihood of preserving network neighborhoods of nodes by using Random Walk. The extraction method of their work is similar to the supervise learning approach - k-means clustering. They both utilise the relationship between the neighbours. Thanks to the random walk, node2vec could apply to different types of graphs. For our graph embedding task, ASTs are able to be extracted in this way to keep the important information.

Once gaining the vector representation of each node, we are able to classify the node type afterwards. However, there are still couple of issues for node2vec implementation:

- Neural network input format. As the number of nodes in every example is variant, we should consider whether the input should be a graph or a node.
- Take buggy nodes as inputs directly would cause the unbalanced issue, because there are only few buggy nodes that could be found in the whole AST tree. If our inputs are every single node, our goal is to distinguish the buggy node. It is easy to feed it into neural network. However, the buggy node is much fewer compared with the normal node. In our implementation, it would cause an issue that predict all the buggy ones as normal to obtain a high accuracy.

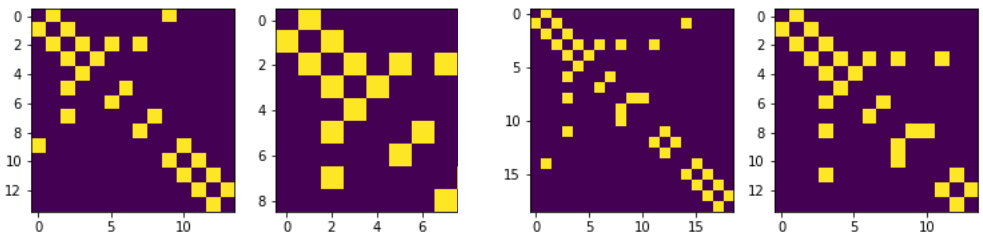
At our current stage, we haven't solved this issue. Instead of solving that, we looked through other paper for graph embedding.

3.2 Graph2vec

Another reasonable way is graph vectorisation. In 2017, building upon the research of node2vec and other vectorisation approaches[1], Narayanan et al.[23] came up with *graph2vec* for graph representation. Graph2vec is able to learn the data-driven distributed representations of arbitrary sized graphs. When it trained on large volumes of graphs, the data-driven representation learning approach would be able to obtain the true inside potentials, like the graph structure, relationship with neighbors, etc.

3.3 Adjacency matrices and Feature matrices

An edgelist corresponds to a square matrix is known as adjacency matrix which contains the connecting information of a finite graph. For the vulnerability involved source code, it might contains some particular connection structures. To find these out, besides vectorising the ASTs, representing the ASTs by using adjacency matrices could be a wise choice. Those adjacency matrices, in some sense, can be treat as image inputs for convolutional neural network. This paper[24] presents a malware behaviour detection by using malware binaries file visualisation, similarly, the buggy code could also include a buggy pattern in the adjacency matrices representation. The figures below show the visualisation of buggy and normal source code adjacency representation. Ideally, neural networks are able to detect more difference between them.



(a) Example 1 (b) Example 2

Figure 3: Pairs of code adjacency representation. (Left is good example, Right is buffer overflow)

In addition, ASTs are not only containing the connecting information, but involving the node type of the source code. After our pretesting stage which can be found in the **Data Pretesting** section, we decided to focus on buffer overflow only. Our implementation is made by constructing feature matrices and combining with adjacency matrices to detect the buffer overflow. Because buffer overflow usually corresponds to an allocation operation and memory overrunning problem, we extract the following below properties for our feature matrices:

- Alloc
- WriteToPointer
- SizeOf

These features also correspond to the ILP predicates and will be explained in the **section 5** - vulnerability inference rules generalisation.

4 Neural Networks

An appropriate source code representation is able to feed into different machine learning algorithms to do supervised learning. Our exploration mainly focused on neural networks which have been used into different papers[28, 17, 6]. Because the neural networks architecture influenced the performance of classifiers, our main implementation is to explore the different architecture of neural network. The following section starts with a simple dense neural network.

4.1 Dense Neural Network

Our neural networks implementation for graph2vec began with neural network with 3 dense layers. In this stage, the main concern is to choose the activation function.

Here below are some activation function choices for classification problem (x, x_i are inputs of activation functions):

- Sigmoid: it is also called Logistic Function which returns the values between 0 and 1.

$$f(x) = \frac{1}{1 + e^{-x}}$$

- Softmax: softmax function can be treat as a general sigmoid function which has the following form:

$$f(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}}$$

  It usually used for multi-classification problems as the sum of the outputs are equal to 1.

Because our main experiment is detecting buffer overflow, as a binary classification problem with one output, the property of sigmoid function, which maps any input into a (0,1) interval, could be a good choice as an output activation function to output a bad or good class.

4.2 Pairwise Neural Network

Noted that, our example exits in pairs. Normal feedforward neural network learned a general buggy pattern, however, it is unable to find the connection between every pair example. Learning the information by pairs potentially would guide Neural Networks to learn the relationship between the corresponding pairs. When the inputs are two corresponding examples, their vector representations could have a strong relationship. Our pairwise neural network aims to learn a pattern to distinguish them. Because it is a multi-classification problem, **softmax** as the output layers activation function could guarantee there is only one example detected as a buffer overflow in each pair. Ideally, this architecture would be helpful for this balanced data, and the model would learn more information about buffer overflow.

4.3 Convolutional Neural Network

Convolutional neural network had a huge breakthrough in the image recognition field[16] as it is able to find local correlation of different graphs. A classic CNN architecture would include:

- Convolutional layers: for feature extraction
- ReLU layers: as an activation function
- Pooling layers: zip data or reduce the dimension

- A fully connected layer: as a classifier

As we discussed before, our code adjacency matrices could be considered as 0-1 images. Through our visualisation, we could somehow recognise the buggy pattern. So it might indicate that, in general, buffer overflow have a special structure of ASTs. In that case, applying convolutional neural network technique could be a potential direction. We hope it would be able to find the buggy pattern through a large number of training examples. To solve the inconsistent size of adjacency matrices of different code snippets, we directly padded them into a fixed size (614*614) matrices. However, this padding pattern model didn't learn the expected buggy structure when it was tested by random padding examples. So another thing is that we tried to test our convolutional neural network's translational invariance by doing random padding, rather than always padding 0s into the same side. However, as we just generated random padding for each datapoint once, it cannot learn the buggy pattern too well. In this case, duplicating the random padding data might be doable in the future exploration.

4.4 Concatenated model

Although our convolutional neural network performs well to detect buffer overflow in Juliet dataset, however, the adjacency matrices representation didn't contain too much useful information of our source code, it is still worth combining the adjacency matrices convolutional neural network with the feature matrices to gain more information. Our concatenated model architecture shows below.

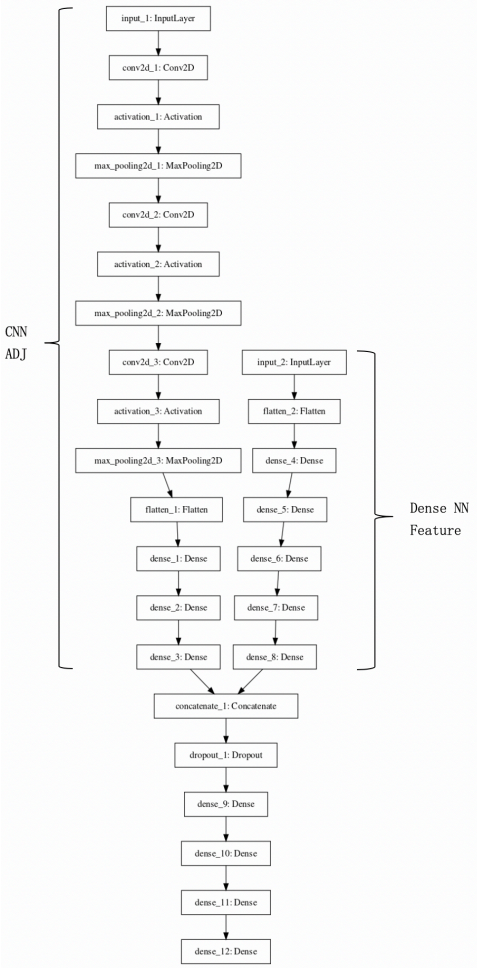


Figure 4: Concatenated model

The concatenated model actually could be divided into three parts: CNN for adjacency matrices, dense

layers for feature matrices, concatenated dense layers. By retaining the original convolutional neural network from input_1 to concatenate_1, we concatenated a feedforward neural network for feature matrices at the middle stage. As for the concatenate layer, we simply padded these two vectors with length 32 into a longer vector with length 64. We believed in this way it keeps balance between the importance of adjacency matrices and feature matrices.

5 Vulnerability inference rules generalisation

Unlike neural networks, Logic Programming does not require a large number of training examples and works fine on a discrete feature representation. Since Code Property Graphs provide us more information about the source code, generating the background knowledge as well as positive and negative examples from CPG for ILP system to generalise the vulnerability inference rules is a good option.

In our project, we extracted some edge properties from joern: [11, 31]:

- AST: Syntax tree edge - structure
- CFG: Control flow edge - execution order and conditions
- REF: corresponds to a reference relationship

Corresponding there are also some node properties:

- assignment: assign value to a variable
- sizeOf: generate the size of a variable or datatype
- sizeOfInt: get the size of a integer
- alloc: give a specific size of memory space
- writeToPointer: represent the memcpy and memmove operation in our example

These properties are potentially useful if we focus on the buggy example below.

5.1 Buggy example

Obviously, there contains a number of buffer overflow examples. Before our experiment, we pick a common one to do analysis. Here is the classic buggy code:

```
#include "std_testcase.h"

void CWE122_Heap_Based_Buffer_Overflow__CWE131_memcpy_01_bad()
{
    int * data;
    data = NULL;
    /* FLAW: Allocate memory without using sizeof(int) */
    data = (int *)malloc(10);
    if (data == NULL) {exit(-1);}
    {
        int source[10] = {0};
        /* POTENTIAL FLAW: Possible buffer overflow if data was not allocated correctly in the source */
        memcpy(data, source, 10*sizeof(int));
        printIntLine(data[0]);
        free(data);
    }
}

int main(int argc, char * argv[])
{
```

```
    /* seed randomness */
    srand( (unsigned)time(NULL) );
    printLine("Calling bad()...");
    CWE122_Heap_Based_Buffer_Overflow__CWE131_memcpy_01_bad();
    printLine("Finished bad()");
    return 0;
}
```

Listing 1: C example

In this case, the memory allocation size didn't match the memcpy size, which could cause a buffer overflow. As the buffer overflow from this example is caused by allocating memory without using sizeof - a node property in code property graph, it is possible to learn the buggy rules by extracting background knowledge from our code property graph.

5.2 Inductive logic programming (ILP) exploration

ILP is described as a symbolic machine learning technique which constructs logic programs from examples. In a cybersecurity context of code vulnerability detection, it has been shown to characterise the general features of a bug, with some flexibility around linguistic variation [20, 26]. There are two basic induction ILP methods namely bottom-up and top-down. Bottom-up method started with extracting most specific clauses from the positive and negative examples, and then generalise these clauses to get the inference rules. The representation of Bottom-up methods is progol which has been proposed by Muggleton[21] in 1995. in contrast, top-down methods extract the clauses from the specific language template and test on the background knowledge and positive and negative examples. In our situation, bottom-up learning strategy seems more reasonable to extract the rules from the examples directly. So in our ILP journey, our success implementations are based on progol 4.4[22].

Based on ILP learning procedure, our idea is to generate enough background knowledge and correct positive and negative examples, ideally, given enough searching space, ILP could generalise the proper rules for our picked examples.

6 Implementation & Results

6.1 Data Description

Internet provides a large number of labelled source code data which brings about great convenience for automatic vulnerability detection. In this paper, we trained on SATE IV Juliet Test Suite with **80% training set and 20% testing set** splitting.

The Juliet Test Suite[25] contains 64099 individual test cases in C/C++ with 118 different Common Weakness Enumeration (CWEs) and 28881 examples in Java with 112 different CWEs. In addition, it points out the location and name of weakness for each example. There is another dataset named Draper VDISC Dataset[13] which could be used for out of sample testing. It includes 1.27 million functions labelled by static analysis for potential vulnerabilities. However, as the label is added by static analysis tools, mislabelled vulnerabilities might exist and without an accurate location. This could be an issue that would influence the out of sample testing performance.

CWE-114: Process Control on line(s): 134

```
123 void CWE114_Process_Control__w32_char_connect_socket_21_bad()
124 {
125     char * data;
126     char dataBuffer[100] = "";
127     data = dataBuffer;
128     badStatic = 1; /* true */
129     data = badSource(data);
130     {
131         HMODULE hModule;
132         /* POTENTIAL FLAW: If the path to the library is not specified, an attacker may be able to
133          * replace his own file with the intended library */
134         hModule = LoadLibraryA(data);
135         if (hModule != NULL)
136         {
137             FreeLibrary(hModule);
138             printLine("Library loaded and freed successfully");
139         }
140         else
141         {
142             printLine("Unable to load library");
143         }
144     }
145 }
```

Figure 5: A bad example in Juliet Dataset

6.2 Evaluation metrics

Model evaluation is an important part for model comparison. Because it is a classification problem, besides accuracy(ACC), we use false positive rate(FPR), false negative rate(FNR), recall(TPR), precision, F1-measure (F1), Receiver Operating Characteristic(ROC) Curves, Precision-Recall(PR) Curves, Matthews correlation coefficient(MCC) and confusion matrix to compare our models' performance. By using TP, TN, FP, FN to represent true positive, true negative, false positive, false negative respectively, our metrics could be defined as below:

- Accuracy(ACC)
$$\mathrm{ACC} = \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{TP} + \mathrm{TN} + \mathrm{FP} + \mathrm{FN}}$$
- False positive rate(FPR)
$$\mathrm{FPR} = \frac{\mathrm{FP}}{\mathrm{FP} + \mathrm{TN}}$$
- False negative rate(FNR)
$$\mathrm{FNR} = \frac{\mathrm{FN}}{\mathrm{FN} + \mathrm{TP}}$$
- Recall(TPR)
$$\mathrm{TPR} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}$$
- Precision
$$\mathrm{Precision} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}}$$
- F1-measure (F1)
$$\mathrm{F}_1 = \frac{2\mathrm{TP}}{2\mathrm{TP} + \mathrm{FP} + \mathrm{FN}}$$
- ROC curves and PR curves: they are statistical tools to evaluate the performance of classifiers.
- ROC AUC: it is the area under a ROC curve.
- PR AUC: it is the area under a PR curve.

- Matthews correlation coefficient(MCC)[19]: it is used in machine learning to evaluate binary classification which corresponds to correlation coefficient, and it can be calculate in this way:

$$\text{MCC} = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

Ideally, a good classifier will have low FPR, low FNR and high Ture positive rate. Correspondingly, the ROC curves is toward top-left corner.

6.3 AST extraction

Every source code could be transformed into AST representation, in order to get that our first thing was AST extraction. There are a wide range of choices for AST extraction:

- Astexplorer[14]: it is an online ASTs generator for different programming languages, but it doesn't support C, C++ currently.
- Clang[4]: it is a front-end compiler for C, C++ and Objective C, and it can transform source code into an AST by using Python Clang Bindings.
- Pycparser[5]: pycparser is python package for parsing C language, which could help to generate ASTs.

The ASTs obtained by clang returned CursorKind classes which provided a kind of "tree structure" and easy to traverse, so we used clang as our AST generator for Juliet data. It could be extracted a lot of information from the returned CursorKind class. In our experiment, we used 'kind', 'displayname', and 'spelling' which corresponds to kind, source code, and name of that AST node. However, the extracting process required us a lot of memory space due to the large size of dataset. This issue was fixed by using DASK[27] in our implementation.

In general, the AST extraction was a time consuming process as it required tons of code to be compiled. DASK could be a recommendation to make it executable for normal machine. In the future, it should be noticed that clang would make mistakes when compiling source code with some libraries loaded.

6.4 Data Pretesting

In order to confirm the direction of our exploration of the dataset, we decided to do data pretesting, ie. building a baseline model for Juliet.

In this stage, based on our code and graph representation, we built our neural dense baseline model directly after implementing *graph2vec* approach.

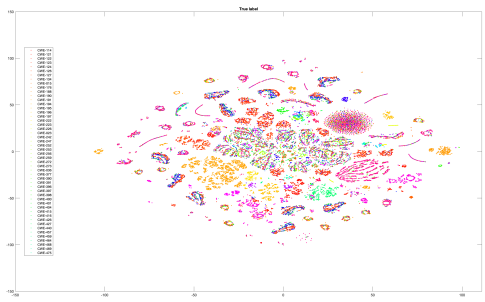


Figure 6: Source code datapoints with True label

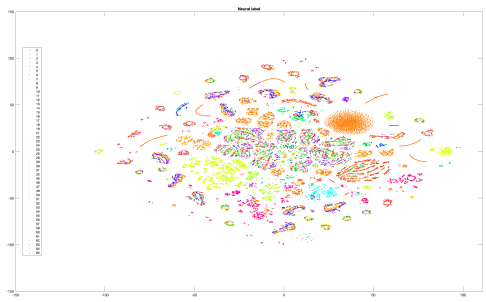


Figure 7: Neural network prediction

Figure 6 and 7 indicate the performance of our baseline model by using the T-SNE[18] visualisation techniques. Different colors represent different classes. It is easy to see that for some particular vulnerabilities, like CWE-191, neural network could distinguish them well. In addition, we implemented other machine learning technique on this source code representation. The baseline model in a way shows that graph2vec has the ability to distinguish different vulnerabilities, so we believe it would be a correct direction to implement graph2vec embedding process to automatically detect bugs.

CWE_ID	Name	counts
CWE-122	Heap-based Buffer Overflow	20090
CWE-121	Stack-based Buffer Overflow	16210
CWE-190	Integer Overflow or Wraparound	13320
CWE-762	Mismatched Memory Management Routines	12776
CWE-134	Use of Externally-Controlled Format String	10320

Table 1: Top 5 Vulnerabilities in Juliet dataset

As it can be seen in this table, buffer overflows are the most frequent vulnerabilities, we agreed to start focusing only on CWE-122 and CWE-121, which corresponds to buffer overflow vulnerabilities, to continue our following experiment.

6.5 Graph2vec embedding approach

In our project, the graph2vec embedding process generated a vector with 128 dimensions for each datapoint (source code example) in Juliet buffer overflow examples. In reality, it could generate arbitrary size of vectors, but it didn't have a significant improvement when the dimensions had been increased to 256. So our implementation just focused on embedded code with 128 dimensions.

6.5.1 Dense Neural Network

Following up with the pretesting step, we filtered our Juliet dataset into buffer overflow and bug free examples with 21052 datapoints in total. By simply feeding them into a 3 layers' neural network, we got 95.4% accuracy.

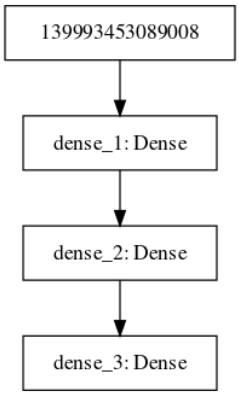


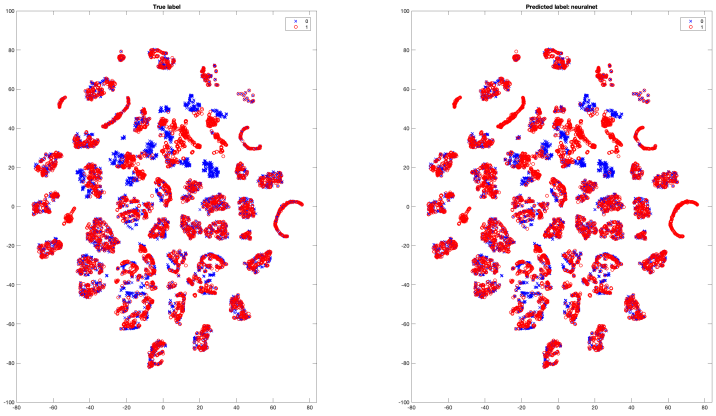
Figure 8: Dense Neural Networks architecture

Besides that, we also implemented a 1D convolutional neural network. Combined with other machine learning algorithms, we can obtain this table below:

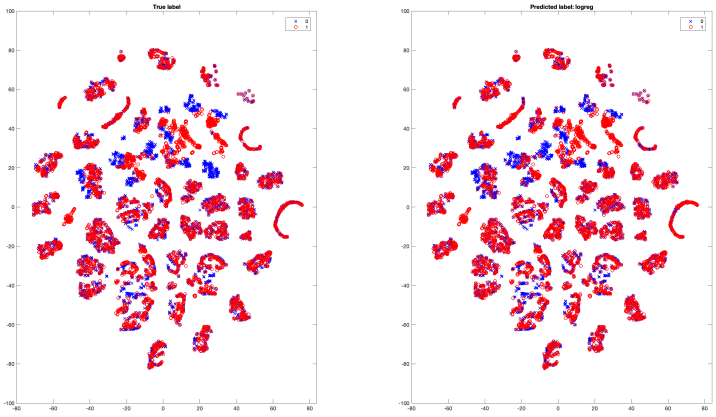
Table 2: Model comparison under the graph2vec embedding

Model	Accuracy	False Positive Rate	False Negative Rate
Dense Neural Network	**0.954**	**0.031**	**0.059**
Convolutional Neural Network	**0.950**	**0.060**	**0.039**
Logistic Regression	0.902	0.108	0.089
Random Forest	0.891	0.098	0.120
SVM	0.885	0.141	0.087
Logistic Regression	0.902	0.108	0.089
1 Nearest Neighbour	0.831	0.178	0.159
Nearest Neighbours	0.820	0.199	0.159
AdaBoost	0.794	0.199	0.212
XGBoost	0.691	0.310	0.309
Gaussian Naive Bayes	0.603	0.405	0.388

Overall, our neural network models performed better than other machine learning algorithms. The performance of CNN and Dense Neural Network performs is similar: one has a lower false negative rate, the other has a lower false positive rate. After applying T-SNE[18] technique, the graphs below show the true label and predicted label comparison results.



(a) Neural Network Prediction



(b) Logistic Regression Prediction

Figure 9: True label and predicted label comparison

In Figure 9, the red points are buggy points and the blue ones are bug-free. It shows that the neural network predictions matched the majority of true labels.

6.5.2 Stacking models

Because we have trained different models for the buffer overflow classification, stacking models could be a good implementation to increase the performance of our classifiers. This example below shows how the stacking models work. It is an example for stacking two models in a 5-fold coss-validation. In 5-fold Coss-validation, we predicted every validation set based on model 1 and take all the predictions as the meta feature to feed into model 2. As for obtaining the test set for the next model (Model 2), a way to do so is to take the average of the testing set predictions predicted from training process. After holding the new features and testing set, new model could be trained based on them.

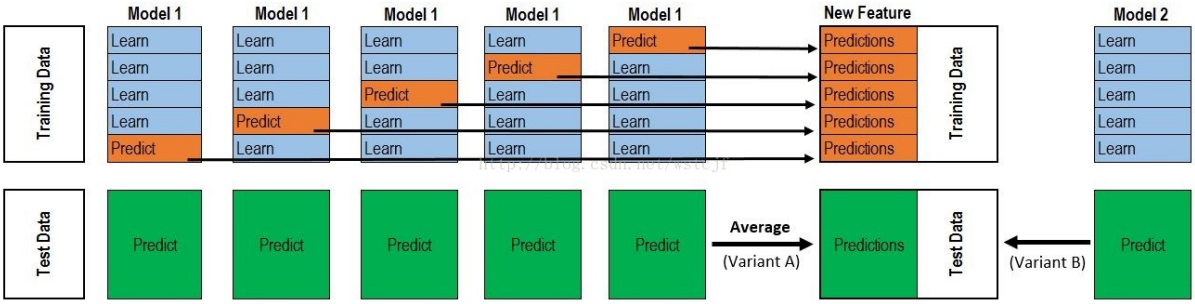


Figure 10: Stacking process

In our experiments[12], we stacked our models together with SVM and logistic regression which obtained 90.1% accuracy and 89.5% F1 score. Continuing stacking with one nearest neighbour, the accuracy and F1 score both increased to 92.1%.

6.5.3 Pairwise Neural Network

From Juliet dataset, we constructed training data pairs for pairwise neural network. In this way, it made the buggy pattern extracting feasible in the future analysis. In our implementation, we simply padded the inputs from 128 with one graph into 256 with pair graphs and added an additional output neuron. The trained model can not only learn the in-sample testing set well (ACC: 96.1%), but performs well on artificial swapped data (ACC: 96.4%). It indicated that the pairwise model learned something we want from the example pairs.

6.6 Adjacency matrices representation

6.6.1 Convolutional Neural network

Our convolutional neural network used 2*2 kernel and 32 filters with 3 classic CNN architectures: ConV, Activation, Pooling layer. The Pooling layers helped us to reduce the dimension of our 'huge' adjacency matrices. After a long time training, here are the results we obtained.

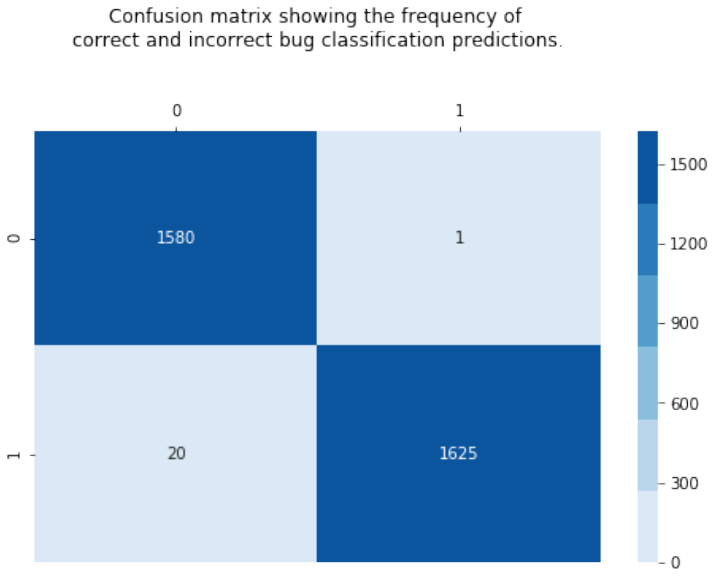


Figure 11: Confusion matrix of CNN model

The confusion matrix above shows that our model has a good performance (ACC: 98.8%) with low false positive rate. In order to test CNN's spatial invariance property, we tested it on random padding adjacency matrices. However, our model don't have the spatial invariance property. So we decided to train our model to a random padding adjacency matrices to increase our model's generalisation performance. When applied random padding on our training set, the model (ACC: 93.6%) started to obtain the spatial invariance property.

6.6.2 Concatenated model

After obtaining the feature matrices using clang, we concatenated our CNN with feature matrices feed-forward neural network and got 97.6% accuracy. However, the random padding training set performed worse (ACC: 86.8%) than others. Overall, these results are quite unexpected, as we expected that the more information our model get, the more better performance it will obtain. It could result from the architecture of concatenated model which could be improved in the future.

To sum up, in the adjacency matrices representation journey, although adjacency matrices representation preserved our code pretty well which could be a good exploration example, the potentially overfitting issue of neural network models cannot be ignored in our future work.

6.7 Results of Neural Network models

These tables below show the comparison of our model results.

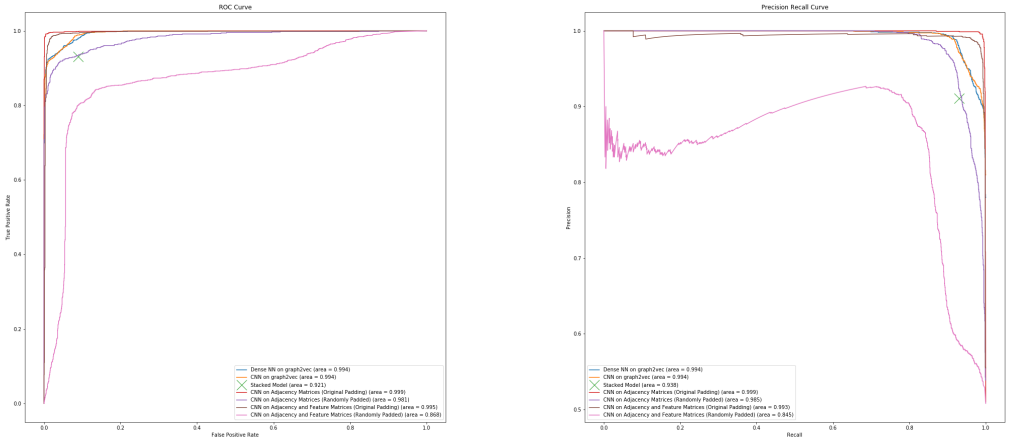
Table 3: Summary statistics table

Model	Accuracy	Precision	Recall	FPR	FNR
G2V + Dense NN	0.954	0.969	0.938	0.031	0.059
G2V+ CNN	0.950	0.940	0.962	0.060	0.039
G2V+ Stacked Model	0.920	0.911	0.931	0.089	0.070
ADJ + CNN	**0.988**	**0.999**	**0.977**	**0.001**	**0.024**
ADJ + CNN (Random)	0.936	0.963	0.911	0.037	0.088
ADJ + CNN + Feature	0.976	0.961	0.992	0.039	0.008
ADJ + CNN + Feature (Random)	0.854	0.886	0.814	0.114	0.177

Table 4: Performance matrices on the Juliet Suite buffer overflow data for our Neural Network models

Model	ROC AUC	PR AUC	F1	MCC
G2V + Dense NN	0.994	0.994	0.953	0.909
G2V + CNN	0.994	0.994	0.950	0.901
G2V + Stacked Model	0.921	0.938	0.921	0.841
ADJ + CNN	**0.999**	**0.999**	**0.988**	**0.975**
ADJ + CNN (Random)	0.981	0.985	0.936	0.874
ADJ + CNN + Feature	0.995	0.993	0.976	0.951
ADJ + CNN + Feature (Random)	0.868	0.845	0.848	0.707

Overall, the adjacency matrices representation plus Convolutional neural network operation has the best performance with a high accuracy and a low false rate. Compared with this paper[28], our model have a small improvement without introducing too much complexity.



(a) ROC curves comparison for different ML approaches. (b) PR curves comparison for different ML approaches.

Figure 12: ROC and PR curves comparison

From ROC and PR curves or Table 4, it is clear that the ADJ + CNN performs best, and the ROC AUC and PR AUC both reached to 99.9%.

These three pictures below show the loss curves on different neural network models.

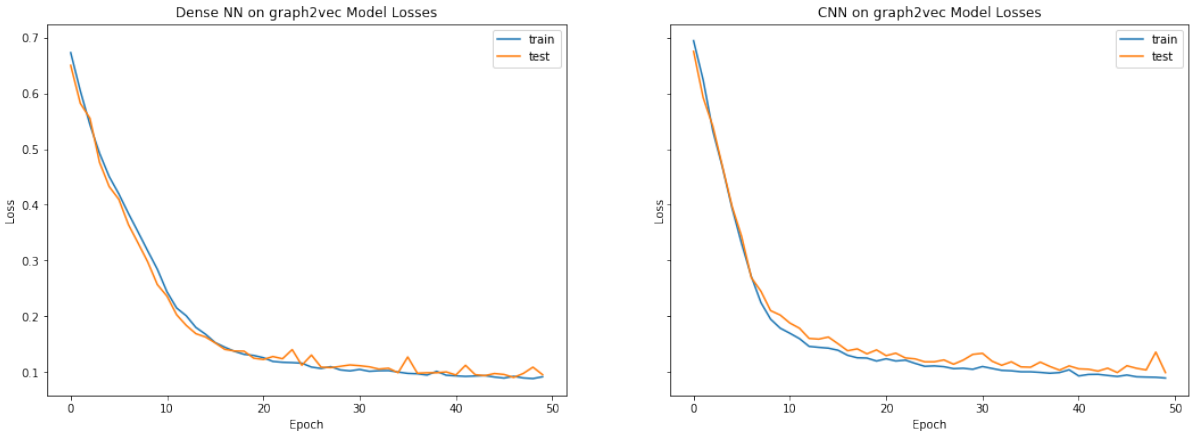


Figure 13: Losses reducing curves for Dense Neural Network and CNN on graph2vec embedding

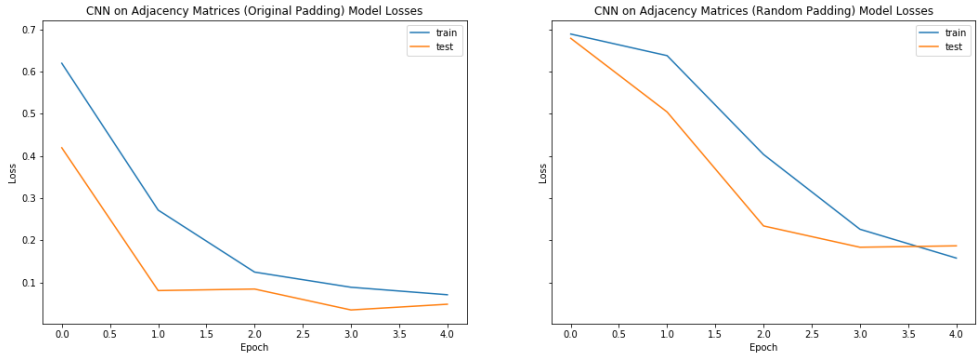


Figure 14: Losses curves for ADJ + CNN model

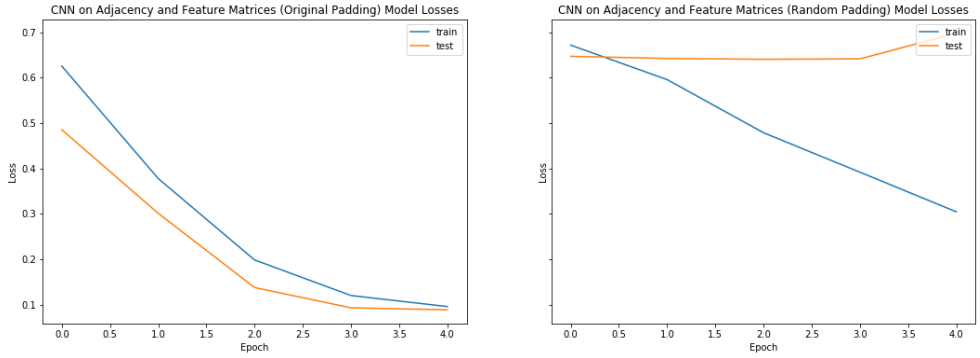


Figure 15: Losses curves for ADJ + CNN + Feature model

Generally, models learned well. But figure 15 indicated an overfitting problem in the random padding model on ADJ + CNN + Feature because the training set loss was decreasing while the testing set loss retained the same. Reducing the learning rate is a potential way to fix it in the future.

The performance of buffer overflow predictions was great for neural networks models by just simply utilising AST nodes especially for adjacency matrices. However, different models have different computational costs. For graph2vec embedding, the embedding procedure time complexity dominated the most time of graph2vec methods. As for adjacency matrices embedding, the convolutional neural network took the majority of the training time. Overall, neural networks models in our implementation is not time-efficient. In addition, even if we got quite high accuracy, it is hard to interpret their decisions as neural networks are lack of model interpretability.

In order to have more explanation of buffer overflow, we definitely needs more information. Because the only AST would potentially lose some source code information. So at the next stage, instead of just using AST nodes, we explored CPG methods on ILP implementation by combining AST, CFG, REF edges.

6.8 Inference rules generations

In our experiments, we tagged the properties on our AST, CFG, REF edges and added the node kind properties based on the node location. So as to get the a brief compression, we defined the **ancestor** and **runs_before** predicates to shorten the ast and cfg predicates in the inference rules.

```
ancestor(A,C) :- ast(A,B), ancestor(B,C).
ancestor(A,C) :- ast(A,C).

```

```
runs_before(A,C) :- cfg(A,B), runs_before(B,C).
runs_before(A,C) :- cfg(A,C).
```

Listing 2: prolog rules

In order to decrease the search space and make the generated rules more human-readable, we manually added another rules to guide the ILP learning process.

```
contains_sizeOf_call(A) :- ancestor(A, B), sizeof(B).
alloc_doesnt_check_sizeOf(A) :- alloc(A), not(contains_sizeOf_call(A)).
```

Listing 3: alloc_doesnt_check_sizeOf rules

If there exists a sizeof checking node A and node A contains the AST edges, then we could say A has sizeOf_call. In addition, since our examples are picked in the similar pattern as shown in **buggy example section 5.1**, the alloc_doesnt_check_sizeOf rule obviously could be more helpful in this particular inference rules generation.

After couple of progol setting, our progol snippets successful generated this rule:

```
bug(A,B) :- ancestor(A,C), ancestor(B,D), assignment(A), writeToPointer(B), alloc_doesnt_check_sizeOf(C),
      sizeOfInt(D).
```

Listing 4: alloc_doesnt_check_sizeOf rules

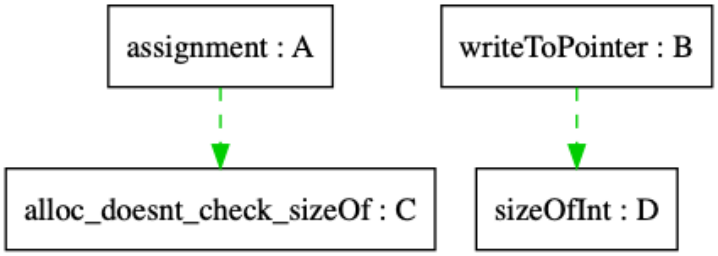


Figure 16: Inference rule visualisation using Graphviz[7]

A and B together cause the buffer overflow, which can be corresponded to this buggy example below:

```
data = (int *)malloc(10);  //A
memcpy(data, source, 10*sizeof(int)); //B
```

Listing 5: C example

In line A, the code doesn't assign data with a proper 10*int size but still do memcpy in line B with the wrong size which would cause a buffer overflow, as allocated data don't have enough space to store more things. Our inference rule captured these features, and they find that if there is a bug in given example:

- In the assignment part, the assignment node has a AST link to the node C that allocates the data but does not call the sizeof function to check the size, which corresponds to 'malloc(10)' in our example. Because our allocation doesn't check the size of, the corresponding good example is '(int *)malloc(10*sizeof(int))' which checked the sizeof while doing allocation.
- In the writeToPointer part - memcpy in our example, the writeToPointer node has a link to sizeOfInt node D which corresponds to the 'sizeof(int)'. As for a good example, '10*sizeof(int)' should be transformed to things like 'sizeof(data)' in order to fit the size of data.

7 Discussion and Future Work

In this project, different ways to represent source code were explored to detect the source code vulnerabilities automatically by feeding it into different machine learning models. The key challenge was taking

a variable-sized input, like program, and converting it into a fixed-sized, continuous representation for most machine learning models.

In general, our CNN adjacency matrices representation was the best according to summary statistics and performance matrices; however, the padding pattern does not have noise robustness as when it was tested on the random padding representation, it predicted randomly. This indicates that the pure adjacency matrices representation is not able to represent a buffer overflow. Concerning noise robustness, adjacency matrices trained on randomly padded data did well on both normal, and randomly padded data, which more or less fixed the issue.

Initially, the expectation was that the more information was extracted from the code, the better performance would be obtained. However, surprisingly, the adjacency matrices plus feature matrices did consistently worse than the adjacency matrices only representation. This could be improved by restructuring the concatenated model in the future work.

As for the time requirement, these two embedding approaches are both time consuming processes. For graph2vec embedding, the speed of the vectorisation of the source code depends on the size of the datasets as well as the size of the ast trees. When it comes to CNN for adjacency matrices, the size of the matrices influences the training time.

Overall, the results show that our code representation performs well in terms of buffer overflow classification on Juliet dataset. The neural network can learn the buggy pattern more or less from the vectors as well as adjacency matrices. Particularly, the adjacency matrices visualisation made the buggy pattern comparison easier for humans to understand. However, there is still a lot of work that could be done in the future.

- Node2vec data utilisation: based on feature matrices and the ast connection, potentially, the vectors could be more powerful. Exploring a direct convolution on graphs like [29] did could be a potential choice.
- Figure out the explanation of why convolutional neural network performs well in buffer overflow maybe it suit for other type of vulnerabilities.
- Build Markov chains (one for good, the other for bad examples) from our feature matrices to summarise the buggy pattern using a statistic perspective.
- Obtain the rule prediction from feature matrices for ILP background knowledge. Since our feature matrices contain the node kind which is feasible to be predicted by neural network. Then from this output, it might be doable to add into our ILP background knowledge to improve our generated rule.
- Out of sample testing. A better way to evaluate our model performance should be tested on out of sample. We have done a few implementation on VDISC dataset which could be found here. But limited to our buggy type selection and the different buffer overflow types between VDISC and Juliet dataset, our best model didn't obtain a good result. In the future, we would like to have a deep exploration of that.
- Extend the buggy detection journey to other programming language. Because the adjacency matrices just retains the AST structure, if it works well for out-of-sample, ideally it could be applied to other programming language as well.

As for ILP, from the background knowledge obtained by code property graph, the progol scripts could generalise the brief human-readable information. Only 40 examples were used to obtain this, which is much more effectively compared to neural networks, which has 21052 datapoints. However, the shortcomings cannot be ignored.

- ILP is not scalable for training. This buffer overflow exploration finds that it requires researchers to manually define at least six rules to obtain the generalised result.

- Current results didn't indicate the connection of the reference relationship. This might have been cause by the examples selection.

Adding more intermediate rules or examples to guide ILP is a potential direction for future implementation in order to obtain a better output. In the future, it is also worthwhile to expand the ILP from buffer overflow into other types of vulnerabilities. As the feature matrices in the neural network model correspond to the ILP predicates, more approaches can be explored from there. In the future, more suitable examples could be selected to create a more general rule for more types of buffer overflows. Evaluating the ILP's performance in terms of testing datasets is also important to complete this research.

References

[1] Bijaya Adhikari, Yao Zhang, Naren Ramakrishnan, and B Aditya Prakash. Distributed representations of subgraphs. In *2017 IEEE International Conference on Data Mining Workshops (ICDMW)*, pages 111–117. IEEE, 2017.

[2] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. *CoRR*, abs/1711.00740, 2017.

[3] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *CoRR*, abs/1803.09473, 2018.

[4] Eli Bendersky. Parsing c++ in python with clang, 2011. [Online]. Available: https://eli.thegreenplace.net/2011/07/03/parsing-c-in-python-with-clang. [Accessed: 28th Aug 2019].

[5] Eli Bendersky. pycparser: C parser and ast generator written in python (2011), 2011. [Online]. Available: https://github.com/eliben/pycparser. [Accessed: 28th Aug 2019].

[6] Min-je Choi, Sehun Jeong, Hakjoo Oh, and Jaegul Choo. End-to-end prediction of buffer overruns from raw source code via neural memory networks. *arXiv preprint arXiv:1703.02458*, 2017.

[7] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C North, and Gordon Woodhull. Graphviz—open source graph drawing tools. In *International Symposium on Graph Drawing*, pages 483–484. Springer, 2001.

[8] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[9] Tapan P Gondaliya, Hiren D Joshi, and Hardik Joshi. Source code plagiarism detection'scpdet': A review. *International Journal of Computer Applications*, 105(17), 2014.

[10] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. *CoRR*, abs/1607.00653, 2016.

[11] ShiftLeft Inc. Joern documentation, 2019. [Online]. Available: https://joern.io/docs/. [Accessed: 28th Aug 2019].

[12] Igor Ivanov. Vecstack, 2016. [Online]. Available: https://github.com/vecxoz/vecstack. [Accessed: 28th Aug 2019].

[13] Louis Kim and Rebecca Russell. Draper vdisc dataset - vulnerability detection in source code, 2018. [Online]. Available: https://osf.io/d45bw/. [Accessed: 28th Aug 2019].

[14] Felix Kling. Ast explorer, 2019. [Online]. Available: https://astexplorer.net. [Accessed: 28th Aug 2019].

[15] Flavius-Mihai Lazar and Ovidiu Banias. Clone detection algorithm based on the abstract syntax tree approach. In *2014 IEEE 9th IEEE International Symposium on Applied Computational Intelligence and Informatics (SACI)*, pages 73–78. IEEE, 2014.

[16] Yann LeCun, Yoshua Bengio, et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995.

[17] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*, 2018.

[18] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605, 2008.

[19] Brian W Matthews. Comparison of the predicted and observed secondary structure of t4 phage lysozyme. *Biochimica et Biophysica Acta (BBA)-Protein Structure*, 405(2):442–451, 1975.

[20] Steve Moyle and John Heasman. Machine learning to detect intrusion strategies. In *Knowledge-Based Intelligent Information and Engineering Systems, 7th International Conference, KES 2003, Oxford, UK, September 3-5, 2003, Proceedings, Part I*, pages 371–378, 2003.

[21] Stephen Muggleton. Inverse entailment and progol. *New generation computing*, 13(3-4):245–286, 1995.

[22] Stephen Muggleton. Progol, 2003. [Online]. Available: https://www.doc.ic.ac.uk/ shm/progol.html. [Accessed: 28th Aug 2019].

[23] Annamalai Narayanan, Mahinthan Chandramohan, Rajasekar Venkatesan, Lihui Chen, Yang Liu, and Shantanu Jaiswal. graph2vec: Learning distributed representations of graphs. *CoRR*, abs/1707.05005, 2017.

[24] Lakshmanan Nataraj, Sreejith Karthikeyan, Gregoire Jacob, and BS Manjunath. Malware images: visualization and automatic classification. In *Proceedings of the 8th international symposium on visualization for cyber security*, page 4. ACM, 2011.

[25] NIST. Juliet test suite v1.3, 2017. [Online]. Available: https://samate.nist.gov/SRD/testsuite.php. [Accessed: 28th Aug 2019].

[26] Oliver Ray, Samuel Hicks, and Steve Moyle. Using ILP to analyse ransomware attacks. In *Proceedings of the 26th International Conference on Inductive Logic Programming (Short papers), London, UK, 2016.*, pages 54–59, 2016.

[27] Matthew Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th python in science conference*, number 130-136. Citeseer, 2015.

[28] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 757–762. IEEE, 2018.

[29] Aditya Sharma. Convolutional neural networks in python, 2017. [Online]. Available: https://www.datacamp.com/community/tutorials/convolutional-neural-networks-python. [Accessed: 28th Aug 2019].

[30] Wikipedia. Abstract syntax tree, 2019. [Online]. Available: https://en.wikipedia.org/wiki/Abstract_syntax_tree. [Accessed: 28th Aug 2019].

[31] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pages 590–604, May 2014.

[32] Serkan Özkan. Cve details: Mozilla firefox, 2010. [Online]. Available: https://www.cvedetails.com/product/3264/mozilla-firefox.html. [Accessed: 28th Aug 2019].