Software Security Engineering Semester Project Proposal

Tony Alarcon
palarcon@nd.edu
University of Notre Dame

Theo Chambers
tchambe2@nd.edu
University of Notre Dame

ABSTRACT

This paper will explore techniques used in the research paper "Vulnerability Prediction From Source Code Using Machine Learning" [1]. Software assurance and automation of software vulnerability detection is a crucial aspect of the security industry. To this end, the paper replicates the exact source code embedding techniques utilized by Bilgin et al., and tests the embedding on various machine learning algorithms to perform intelligent analysis on source code in order to correctly predict vulnerabilities. The three machine learning algorithms that we present are a Multi-layer Perceptron (MLP) Model, a Convolutional Neural Network (CNN) Model, and lastly a Support Vector Machine (SVM) model. All three models were able to produce high accuracy, precision, and recall metrics on our curated CWE datasets.

1 PROBLEM DEFINITION

Software vulnerabilities may have widespread impact; depending on it's severity, a single vulnerability can compromise an entire system. As such it is in the developers and investor's best interest to identify possible vulnerabilities early on to mitigate it's impact while in deployment. Nevertheless, vulnerabilities are often difficult to detect, decipher, and prevent. Dynamic and Static Analysis tools are the standard accepted approaches to detecting vulnerabilities, however, they come with limitations, drawbacks and are far from perfect. For example, static analysis tool tend to analyze a program by exploring all the different paths that the program can take, without executing the code. However, this makes them susceptible to vulnerabilities that can occur during runtime. Additionally, exploring all different paths in a program quickly becomes exponentially expensive, and may be time consuming especially for large applications. Furthermore, static analysis tools can only analyze limited properties of the program. For Dynamic Analysis tools, we analyze a program by executing the code to check how it behaves during runtime, however, we are usually limited to analyzing a single path in a program. As such, vulnerability detection becomes ineffective if all path are not properly covered. It has been shown that static and dynamic analysis tools are ineffective for detecting certain vulnerabilities [3]. As such, Bilgin, et. al. proposes an additional approach that leverages state of the art machine learning models for vulnerability detection from source code.

2 INTRODUCTION

Source code is a collection of code identifiers, written in human-readable alphanumeric characters that serves to bridge the gap between machine and human language. As such, academic researchers have leveraged word embedding techniques to capture the predictable statistical properties in source code for many software engineering tasks [4]. Such contemporary word embedding techniques transforms a natural language into contiguous numerical vectors that preserves syntactical and semantical meaning in words/sentences. Nevertheless, the authors in [7], [8] have demonstrated that source code can not only be interpreted as a natural languages, but additionally has many representative forms such as AST, data flow, and control flow. Therefore, numerous studies have been performed that aims to find an efficient representation of source code for ML applications that captures comprehensive semantics to characterize complex and diverse vulnerabilities. This paper will explore, verify and replicate the techniques employed by Belgin, et. al. [1] for predicting vulnerabilities from source code using a Machine Learning Approach.

2.1 Source Code Representation as AST

Belgin, et. al. employ a seven step approach to develop a meaningful representation of the source code, briefly described below.

Source Code. Fragments of source code in a particular high-level programming language are selected such that AST can be obtained using appropriate parsers. Appropriate languages include Python, C++, C, Java, etc.
Function-Level Partition. Source code has various constituents compromising of components, functions, classes, lines, etc. This step extracts function-level code from source code to reduce arbitrarily long lines of code. Function-level source code is chosen given that it captures the overall flow of subroutine and provides a more precise localization for predicting vulnerabilities.
Tokenization. This step first removes unnecessary elements such as spaces, tabs, comments and commas from function-level code and converts the remaining code identifiers into tokens.
AST Generation. Using appropriate parsers for a given language, AST's are generated from tokenized source code that contains syntactic and semantic meanings.
Conversion to Complete Binary AST. AST must be converted into a data structure that conforms to the Machine Learning input prerequisites. To that end, AST are first converted into binary trees to preserve the structural relations of nodes.
Encoding to Numerical Tuples. This step simply maps the nodes into predetermined numerical tuples, each with 3 values. The first element represent the type of token, while the second and third element is used to maintain auxiliary information that exists at the node.
Array Representation. Lastly, numeric arrays are generated from the numerical tuples such that they preserve the locations of the nodes in a Tree.

Software Security Engineering, Notre Dame, US
2022. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

The rest of this paper is organized as follows. In section 3, we provide an overview of the dataset utilized for training and discuss under-sampling method. In section 4, we provide an in-depth discussion of Belgin's novel source code embedding methods with examples. Then in section 5, we provide the machine learning algorithms utilized for this paper and it's corresponding hyper parameters. In section 6, we analyze the results obtained for each training model in the proceeding section. Lastly, section 7 concludes the paper and provides future work considerations.

3 DATASET OVERVIEW

This paper utilizes the public Draper VDISC Dataset. This dataset contains 1.27 million function-level C code samples mined from various open source projects, which includes, but is not limited to, Debian Linux Distribution, NIST's Samate, SATE IV Juilet Test Suite and many other public repositories on Github. For more information about this dataset, please refer to [6].

While this dataset contains extensive training data, roughly 91.35% of the samples could not be compiled (and thus unusable for our purposes) for various reasons, i.e. missing semicolon ';' at the end of statements, syntax errors, etc. As such, this paper focuses on the 122,601 compilable function-level source code. As shown in Fig. 11, the label distribution for the sampled dataset is highly imbalanced, with vulnerable functions corresponding to less than 10% of the available data. This skewed class distribution may lead to unequal miss-classification costs during the training process, as such the machine learning models are ill-equipped to learn meaningful characteristic that are indicative of vulnerable functions. Therefore this issue must be addressed in order to avoid poor predictability performance on the minority class. This paper utilizes the under-sampling technique, in which all vulnerable functions are kept and non-vulnerable functions are randomly sampled without replacement until the size of the labels in the training set are equal. All machine learning models discussed in subsequent sections are then trained on this under-sampled balanced dataset.

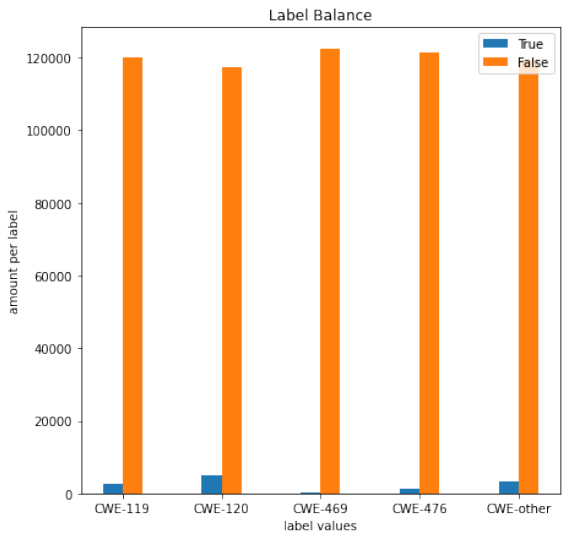
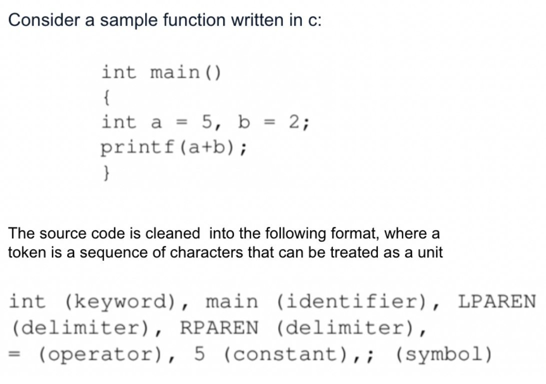


Figure 1: Label Distribution of the 122,601 compilable function-level C source code

4 SOURCE CODE EMBEDDING

Below is a walk-through of the entire process of the source code embedding into a vector representation for the machine learning algorithms. Using the same sample function that the author's used[1], we demonstrate print outs taken directly from our source code with similar results.


Consider a sample function written in c:

```
int main()
{
int a = 5, b = 2;
printf(a+b);
}
```

The source code is cleaned into the following format, where a token is a sequence of characters that can be treated as a unit

```
int (keyword), main (identifier), LPAREN
(delimiter), RPAREN (delimiter),
= (operator), 5 (constant),; (symbol)
```

Figure 2: Tokenizing

After tokenizing and cleaning the source code, we transform the parsed code into an AST tree. In our case, we used a pycparser to transform code into the proper AST structure.

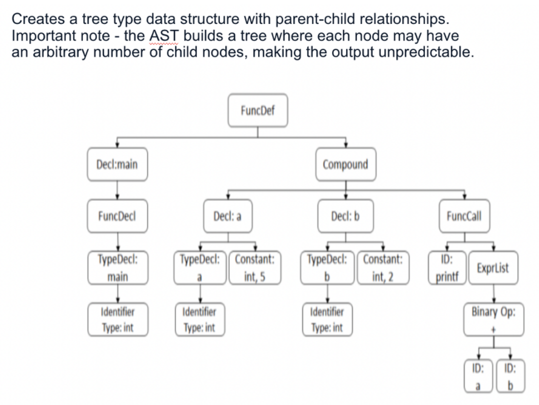
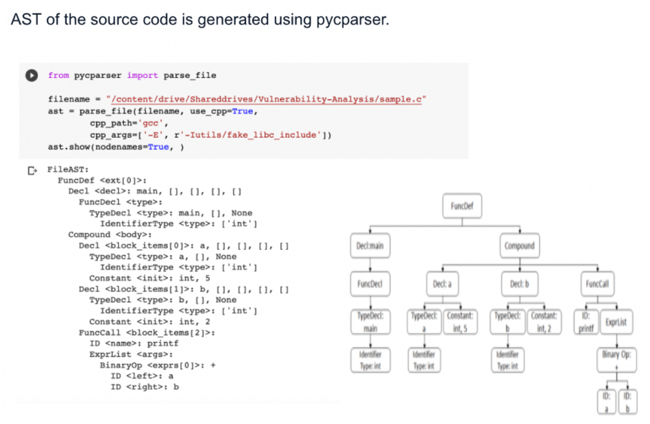

Creates a tree type data structure with parent-child relationships. Important note - the AST builds a tree where each node may have an arbitrary number of child nodes, making the output unpredictable.

Figure 3: AST tree


AST of the source code is generated using pycparser.

```
from pycparser import parse_file

filename = "/content/drive/Shareddrives/Vulnerability-Analysis/sample.c"
ast = parse_file(filename, use_cpp=True,
        cpp_path='gcc',
        cpp_args=['-E', r'-Iutils/fake_libc_include'])
ast.show(nodenames=True, )
```

```
FileAST:
  FuncDef <ext[0]>:
    Decl <decl>: main, [], [], [], []
      FuncDecl <type>:
        TypeDecl <type>: main, [], None
          IdentifierType <type>: ['int']
    Compound <body>:
      Decl <block_items[0]>: a, [], [], [], []
        TypeDecl <type>: a, [], None
          IdentifierType <type>: ['int']
        Constant <init>: int, 5
      Decl <block_items[1]>: b, [], [], [], []
        TypeDecl <type>: b, [], None
          IdentifierType <type>: ['int']
        Constant <init>: int, 2
      FuncCall <block_items[2]>:
        ID <name>: printf
        ExprList <args>:
          BinaryOp <exprs[0]>: +
            ID <left>: a
            ID <right>: b
```

Figure 4: AST tree (our code printout on left, author's diagram on right

The next step requires conversion of the AST to a complete binary tree. All leaves have the same depth, and all internal nodes have two children. This is performed as follows[1]:

(1) The leftmost single child of a node in the m-ary tree (AST) is assigned as the right-child of the corresponding node in the complete binary tree
(2) The right sibling of node-x in the m-ary tree is assigned as left child of node-x in the complete binary tree
(3) If node-x has no children, then its right-child becomes NULL, if node-x is the rightmost child of its parent, then left-child becomes NULL.

The diagram below demonstrates the complete tree structure.

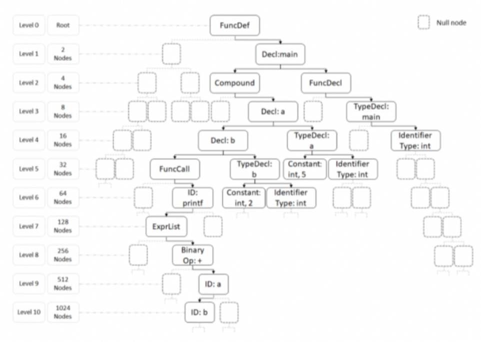
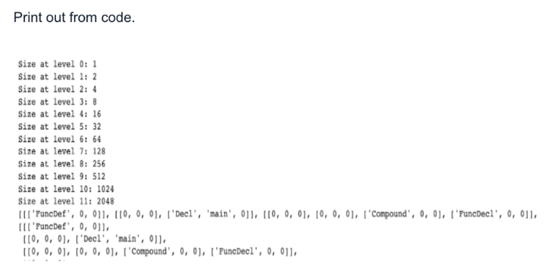


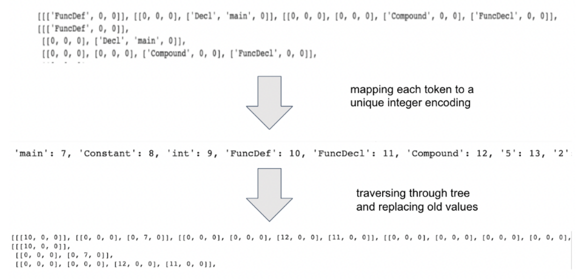
Figure 5: Complete Binary Tree


Print out from code.

```
Size at level 0: 1
Size at level 1: 2
Size at level 2: 4
Size at level 3: 8
Size at level 4: 16
Size at level 5: 32
Size at level 6: 64
Size at level 7: 128
Size at level 8: 256
Size at level 9: 512
Size at level 10: 1024
Size at level 11: 2048
[[['FuncDef', 0, 0]], [[0, 0, 0], ['Decl', 'main', 0]], [[0, 0, 0], [0, 0, 0], ['Compound', 0, 0], ['FuncDecl', 0, 0]],
[[['FuncDef', 0, 0]],
 [[0, 0, 0], ['Decl', 'main', 0]],
 [[0, 0, 0], [0, 0, 0], ['Compound', 0, 0], ['FuncDecl', 0, 0]],
```

Figure 6: Complete Binary Tree: Code Results

The last step requires the mapping of each token to a unique integer encoding. This is done by traversing through the tree structure, and replacing each token by a unique integer. This is done by creating an integer value for each individual instance of a token.

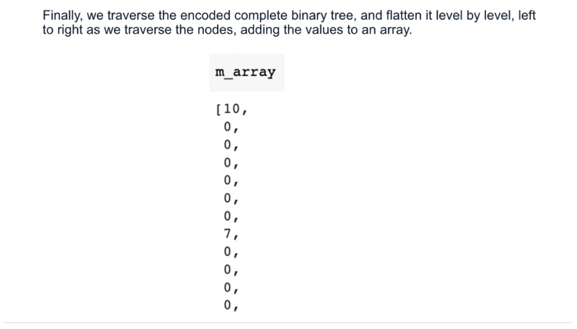


```
[[['FuncDef', 0, 0]], [[0, 0, 0], ['Decl', 'main', 0]], [[0, 0, 0], [0, 0, 0], ['Compound', 0, 0], ['FuncDecl', 0, 0]],
[[['FuncDef', 0, 0]],
 [[0, 0, 0], ['Decl', 'main', 0]],
 [[0, 0, 0], [0, 0, 0], ['Compound', 0, 0], ['FuncDecl', 0, 0]],
```
mapping each token to a unique integer encoding

```
'main': 7, 'Constant': 8, 'int': 9, 'FuncDef': 10, 'FuncDecl': 11, 'Compound': 12, '5': 13, '2'
```
traversing through tree and replacing old values

```
[[[10, 0, 0]], [[0, 0, 0], [0, 7, 0]], [[0, 0, 0], [0, 0, 0], [12, 0, 0], [11, 0, 0]], [[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0],
[[[10, 0, 0]],
 [[0, 0, 0], [0, 7, 0]],
 [[0, 0, 0], [0, 0, 0], [12, 0, 0], [11, 0, 0]],
```

Figure 7: Encoding


Finally, we traverse the encoded complete binary tree, and flatten it level by level, left to right as we traverse the nodes, adding the values to an array.

```
m_array

[10,
 0,
 0,
 0,
 0,
 0,
 0,
 7,
 0,
 0,
 0,
 0,
```

Figure 8: Flattened Array Representation

5 MACHINE LEARNING MODELS

After adequate source code representation has been generated, this paper leverages state of the art machine learning algorithms and present a comparative analysis on its performance. The algorithms presented will extend beyond the algorithms presented in [1]. This subsection provides a quick discussion of the machine leaning models utilized and it's hyperparameters.

5.0.1 Multilayer Perceptron (MLP). Multilayer perceptrons are a class of fully connected layers of neurons, typically subdivided into input, hidden, and output layers. Each neuron in the MLP computes a weighted sum of it's input and uses a non-linear activation function to transform it's input. Such operations performed successively layer by layer has been proven to make the MLP model an universal appropriator, also referred to as the Universal Approximation Theorem [5]. MLP models have a high degree of customizability, and often vary in number and size of their hidden layers, which may have different effects in different types of training data. This paper utilizes the hyperparameters displayed in Table 1 to construct a binary classifier on the under-sampled balanced dataset.

Hyper-parameter	Value
Optimizer	Adam
Hidden Nodes	10
Batch Size	250
Activation	ReLU
Loss Function	Binary Cross Entropy
Epochs	40

Table 1: MLP Hyper Parameters

5.0.2 Convolutional Neural Network (CNN). Convolutional Neural Networks are another archetype of artificial neural networks known for their weight-sharing mechanism which may learn to become invariant to translation and rotation [2]. CNNs may contain several successive layers of convolution and pooling as a pre-processing step prior to a fully connected multi-layer perception. The convolution layer uses various filters/kernals of fixed sized that performs a convolution operation across the entire image creating a distinct feature maps for each kernel. The pooling layer typically operates on the feature map to further sample the output via averaging or max sampling across a fixed region. The CNN's hyper-parameters utilized for this paper are described in Table 2.

Hyper-parameter	Value
Optimizer	Adam
Filters	32
Kernal Size	32
Hidden Nodes	200
Dropout Rate	.5
Batch Size	250
Activation	ReLU
Loss Function	Binary Cross Entropy
Epochs	40

Table 2: CNN Hyper Parameters

5.0.3 Support Vector Machine (SVM). Support Vector Machines are robust machine learning models for classification problems. SVMs map data to a higher dimensional feature space, in order to separate data into classes. This works by first fitting the data with a graphical separator, then attempting to maximize the distance between each class to to the hyperplane boundary. The only hyperparameter used to train the SVM was the decision function shape, "OVO", meaning a one v. one approach for multi-class classification where the number of generated models is proportional to the number of classes. This is the standard approach for SVM design using the sklearn library.

6 EVALUATION AND ANALYSIS

To properly quantify the results of the machine learning models on the 5 training sets, we display precision-recall curves that show the trade-off between precision and recall for different thresholds below. A higher area under the curve represents both high recall and high precision, where high precision corresponds to a low false positive rate, and high recall corresponds to a low false negative rate. High scores for both show that the classifier is returning accurate results (better precision), as well as returning a majority of all positive results (better recall). The goal is to have a large area under the curve for all data sets.

Beginning with the MLP model, we see that the model performed best on CWE-IDs 119 and 120. This is because the model is able to generalize relatively well on a more evenly distributed dataset, whereas the model struggled consequently on CWE-Other, indicative of multiple vulnerabilities lumped into one category. The MLP model was the third best model with its highest accuracy of 88% on CWE-119. Next, the CNN model performed exceedingly well for all CWE-IDs except CWE-469. This is because DNN (Deep Neural Networks) require significant training samples to properly generalize, and the CNN was not able to perform well with a low sampled dataset (around 100 samples), such as CWE-469. However, the CNN performed the second best with a high accuracy score of 93% on CWE-other, and proportionally high precision and recall metrics. Lastly, the SVM model outperformed all other models in accuracy, precision, and recall with the highest score being 95% in all three metrics. The ROC curve looks very straight, because the precision and recall scores were so evenly balanced for each dataset. SVM models are simple, but paradigm models for classification problems.

In this situation, it was able to classify well with a simple classification problem and highly embedded/processed data. We hypothesize that this is due to SVMs consistent performance with two-class problems (binary classifications), and datasets free of noise.

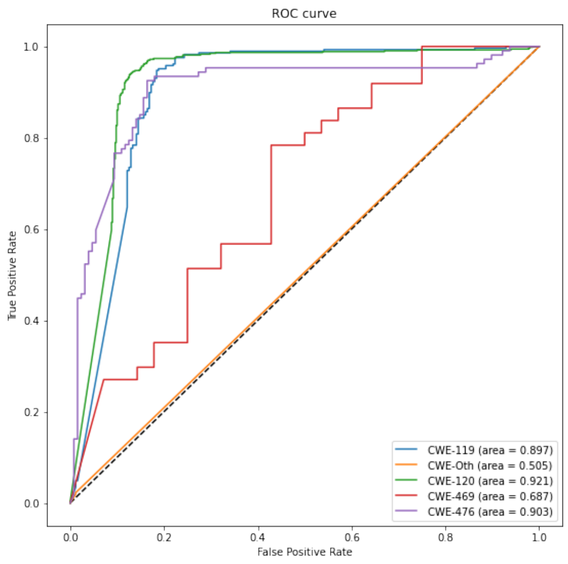


Figure 9: ROC Curve for Multi-layer Perceptron (MLP) Model

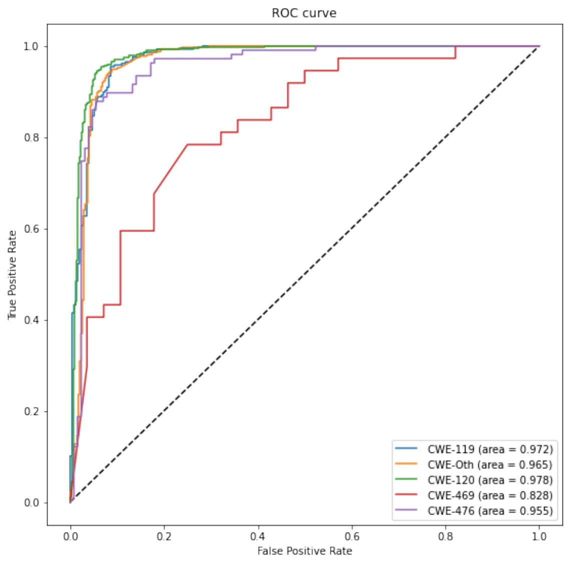


Figure 10: ROC Curve for Convolutional Neural Network (CNN) Model

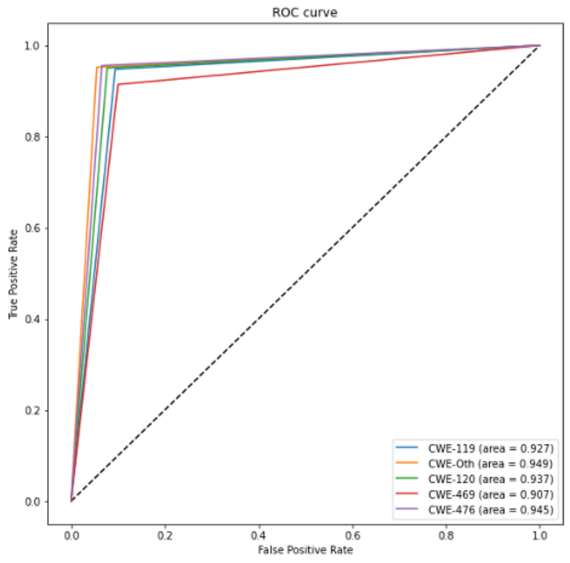


Figure 11: ROC Curve for Support Vector Machine (SVM) Model

7 CONCLUSION AND FUTURE WORK

In conclusion, this article has demonstrated that Bilgin's novel source code embedding methods works well in the context of vulnerability prediction. All three of our machine learning algorithms were able to generate high accuracy, precision, and recall scores with regards to properly classifying data to its CWE-ID. For future work in this project, we would like to explore other alternatives to Belgin's source code embedding method, including BERT (an NLP technique derived by Google), and Code2Vec (an alternative state of the art embedding technique).

REFERENCES

[1] Zeki Bilgin, Mehmet Akif Ersoy, Elif Ustundag Soykan, Emrah Tomur, Pinar Çomak, and Leyli Karaçay. 2020. Vulnerability Prediction From Source Code Using Machine Learning. *IEEE Access* 8 (2020), 150672–150684. https://doi.org/10.1109/ACCESS.2020.3016774

[2] Valerio Biscione and Jeffrey S. Bowers. 2021. Convolutional Neural Networks Are Not Invariant to Translation, but They Can Learn to Be. (2021). https://doi.org/10.48550/ARXIV.2110.05861

[3] Aurélien Delaitre, Bertrand Stivalet, Paul E. Black, Vadim Okun, Athos Ribeiro, and Terry S. Cohen. 2018. SATE V report: ten years of static analysis tool expositions.

[4] Vasiliki Efstathiou and Diomidis Spinellis. 2019. Semantic Source Code Models Using Identifier Embeddings. In *Proceedings of the 16th International Conference on Mining Software Repositories* (Montreal, Quebec, Canada) *(MSR '19)*. IEEE Press, 29–33. https://doi.org/10.1109/MSR.2019.00015

[5] Kurt Hornik, Maxwell B. Stinchcombe, and Halbert L. White. 1989. Multilayer feedforward networks are universal approximators. *Neural Networks* 2 (1989), 359–366.

[6] Rebecca L. Russell, Louis Y. Kim, Lei H. Hamilton, Tomo Lazovich, Jacob A. Harer, Onur Ozdemir, Paul M. Ellingwood, and Marc W. McConley. 2018. Automated Vulnerability Detection in Source Code Using Deep Representation Learning. *CoRR* abs/1807.04320 (2018). arXiv:1807.04320 http://arxiv.org/abs/1807.04320

[7] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A Novel Neural Source Code Representation Based on Abstract Syntax Tree. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 783–794. https://doi.org/10.1109/ICSE.2019.00086

[8] Yaqin Zhou, Shangqing Liu, Jing Kai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. *CoRR* abs/1909.03496 (2019). arXiv:1909.03496 http://arxiv.org/abs/1909.03496

CWE	MLP			CNN			SVM		
	Accuracy	Precision	Recall	Accuracy	Precision	Recall	Accuracy	Precision	Recall
CWE-120	87%	90%	85%	93%	89%	98%	94%	94%	94%
CWE-119	88%	85%	93%	91%	91%	92%	93%	93%	93%
CWE-469	63%	68%	68%	77%	81%	78%	91%	91%	91%
CWE-476	86%	82%	87%	89%	85%	91%	95%	95%	94%
CWE-Oth	53%	64%	02 %	93%	90%	95%	95%	95%	95%

Table 3: Machine Learning Model Results