2023

APPLICATIONS OF TRANSFER LEARNING FROM MALICIOUS TO VULNERABLE BINARIES

Sean Patrick McNulty
The University Of Montana

APPLICATIONS OF TRANSFER LEARNING FROM MALICIOUS TO VULNERABLE BINARIES

By

Sean Patrick McNulty

Bachelor of Arts, The University of Montana, Missoula, MT, 2020

Thesis

presented in partial fulfillment of the requirements
for the degree of

Master of Science
in Computer Science

The University of Montana
Missoula, MT

July 2023

Approved by:

Ashby Kinch Dean of The Graduate School
Graduate School Dean

David Opitz Ph.D., Chair
Computer Science

Patricia Duce
Computer Science

Andrew Ware Ph.D.
Physics & Astronomy

ABSTRACT

McNulty, Sean Patrick, M.S., July 2023 Computer Science

Applications of Transfer Learning from Malicious to Vulnerable Binaries

Chairperson: David Opitz

Malware detection and vulnerability detection are important cybersecurity tasks. Previous research has successfully applied a variety of machine learning methods to both. However, despite their potential synergies, previous research has yet to unite these two tasks. Given the recent success of transfer learning in many domains, such as language modeling and image recognition, this thesis investigated the use of transfer learning to improve vulnerability detection. Specifically, we pre-trained a series of models to detect malicious binaries and used the weights from those models to kickstart the detection of vulnerable binaries. In our study, we also investigated five different data representations of portable executable binaries, all but one of which showed positive transfer in at least one experiment. The single-channel image and tf-idf assembly instruction count embedding were particularly successful, increasing the accuracy of a non-transfer randomly initialized model from 77.2% to 95.8%.

1 Introduction

As the importance of computers, particularly networked computers, has grown over the last few decades, the number of cyber-attacks has also grown. Many cyber-attacks involve malware (**mal**icious soft**ware**), a generic term for all programs on a computer performing unwanted behavior. However, despite the generality of the term, different malware programs share similar features The field of malware detection groups malware programs together based on these shared features.

Researchers group malware detection methods into two categories, static and dynamic. Dynamic malware analysis runs the malicious file (often in a virtual environment) and investigates its behavior. If a dynamic malware detector observes malicious behavior, it flags the file as malicious. However, the malicious file may detect the virtual environments in which it is running and may behave correctly in response to testing methods. In addition, running the file for analysis also risks causing the file to be executed and infecting the system running the file. Even a virtual environment can be circumvented in the analysis, allowing the virus to infect the system.

Static malware analysis is a series of techniques in which the malicious file does not need to execute in the malware detection scheme. Signature matching is a standard method in static malware detection. In signature matching, a malware detector compares strings in a sample piece of code to known malicious samples in a database [1, 2]. Many popular antivirus software, like Kaspersky and Kingsoft, use signature matching [2]. However, malware often employs code obfuscation which renders signature matching obsolete. Also, the number of signatures is also rapidly increasing, making signature detection more difficult [1].

While these two traditional malware analysis techniques have many drawbacks, machine learning has recently shown great promise with increased accuracies in malware classification [3, 4, 5, 6, 7] and malware detection [8, 9, 7]. An advantage of machine learning is its ability to address novel problems which may be challenging to solve in traditional malware analysis. Machine learning algorithms and architectures can detect patterns imperceivable or difficult to perceive for humans.

Cybersecurity applications employ machine learning [6] on tasks other than malware detection and classification. One such case is the wide use of machine learning within the field of vulnerability classification and detection[10, 11].

Vulnerability detection often only has small data sets. Deep learning, an effective machine learning technique [12], typically requires extensive data (in the hundreds of thousands or millions) to achieve state-of-the-art results. Such large data sets are impractical in cybersecurity. Even when it is possible to assemble such a data set, it may be impractical for security companies due to the significant monetary and time cost involved. Thus, providing a pre-trained model to initialize the learning process would be beneficial. Transfer learning is the primary way to address small data set sizes within machine learning [13, 14]. Many of the most successful and well-known examples of machine learning within the last few years have utilized transfer learning.

Broadly, transfer learning is the process of developing a model or method focusing on one data set and transferring that learning information to a related problem. The model pre-trained on one data set, which is termed the source data set. This pre-trained model is trained further (i.e. fine-tuned) on a related data set, termed the target data set. Given similarities between the two data sets, pre-training on the source data set may aid the training process on the target data set and increase training speed and overall accuracy [15].

There are three ways transfer learning may increase the accuracy of the target task. Assuming the source and target data set are similar enough, the training may begin at a higher accuracy, have a steeper learning slope, or even reach a higher accuracy at which it plateaus [15]. The target data set is often much smaller than the source data set, and the target model can often be trained for a shorter time, even in cases with state-of-the-art results. However, if the two data sets are not similar enough, or incorrectly applied, transfer learning can negatively impact the learning of the target data set [15]. In recent years transfer learning has shown great promise, specifically in image generation, with models such as Stable Diffusion, and in natural language processing, with large language models such as Chat-GPT.

The goal of this research is to use relatively simple neural networks and small data sets to show the potential for transfer learning between malware and vulnerabilities. Using simple neural networks provides a solid basis for further research into this area.

2 Related Work

With the growth of both machine learning and deep learning in recent years, there has been increasing attention on applying machine learning methods to cybersecurity [6]. Within this domain, there has been a focus on detecting and classifying malware and vulnerabilities. As in all machine learning problems, identifying a suitable data set and feature set have been the two overarching problems.

There are many methods to generate feature sets for malware detection, including opcode frequency [12, 2], control flow graph recovery [2], n-grams [2], strings [2, 16], byte histograms [16], API calls [2, 17], and single channel byte image generation [18, 19, 3]. Like other forms of malware analysis, feature set generation from malware binaries can include static and dynamic methods [2]. However, static methods remain the most popular due to both speed and the danger of running malicious files.

Of interest for this research are techniques that investigate malware by creating a single-channel byte image from a malicious binary. This form of feature generation became popular after the creation of the Malimg data set in 2011 [18]. This feature set intends to utilize pre-existing neural networks which have shown particular success with analyzing images, specifically the many varieties of convolutional neural nets such as ResNet, Inception, and VGG. Much previous research into transfer learning with malware has involved this data representation [1, 7, 20, 21]. However, all known examples in the literature involve transferring from models pre-trained on image classification (typically ImageNet) to models detecting malware images [1, 7] or a classification task between two malicious data sets [20, 21]. There are no known examples of transferring from malware to vulnerabilities.

Similarly, many examples exist of machine learning techniques in vulnerability detection, analysis, and fixing [22, 10]. However, most of these methods utilize natural language processing techniques on vulnerable source code [11, 23, 24, 25]. There are some examples of binary analysis rather than source code analysis, with feature sets of these including decompilation to assembly instructions [26, 27], creation of control flow graphs [28], and generation of numeric features from functions (e.g., number of parameters, number of instructions, size of local variables) [28]. We use decompilation to assembly and creation of control flow graphs (CFGs) in our research. In particular, creating CFGs through a decompiler gives us access to the functions of a CFG and the assembly language instructions.

The assumption behind many vulnerability data sets, and experiments operating on source

code, is that organizations utilizing these methods would have access to the vulnerable source code for their applications. However, these techniques are notably not applicable to malware analysis since an organization is not likely to have access to data sets containing malicious source code. Depending on the specific case, source code may not be available to an organization either.

Theoretically, vulnerability and malware detection share some commonalities, which previous researchers note [27, 28]. The presumed similarity in detection methods is because malicious software attempts to exploit vulnerabilities to infect and infiltrate host systems. However, as far as we know, there has been no attempt to use machine learning methods to link malware and vulnerability detection.

3 Methods

3.1 Data Sets

Our task is to transfer information learned from a malware detection task to a vulnerability detection task. The first data set contains binaries labeled as either malware or benign, which will be our source data set. The second data set contains binaries labeled as vulnerable or benign, which will be our target data set.

3.1.1 Source Data Set (Malware-Benign)

The primary source data set contains pre-compiled portable executable (PE) binary samples of malicious and benign cases. We obtained the source data set from Practical Security Analytics [29], which contains 86,812 benign and 114,737 malicious binaries [29]. The benign PE binaries are primarily drawn from Windows 7, and the malicious PE binaries from VirusShare, MalShare, and TheZoo [29]. From these, we randomly chose 4,000 malicious and 4,000 benign binaries. Montana State University (MSU) also provided a secondary source data set of 1,860 malicious and 1,860 benign binaries.

Many existing malware data sets contain information about the specific type of malware [18, 30]. We do not need the malware type for our research. To utilize all data representations, it is necessary for the data set to contain a PE binary for each case or for a PE binary to be obtainable for each case, for example, by compiling source code. The Practical Security Analytics [29] data set fulfills our requirements as each case is a pre-compiled PE binary, not a pre-processed data representation as in many other malicious data sets [18, 16].

3.1.2 Target Data Set (Vulnerability-Benign)

The target source data set contains PE binaries labelled as vulnerable or benign. We drew the vulnerable binaries from the NIST Software Assurance Reference Data set (SARD), which has many available cases to choose from [24, 31]. We gathered the benign binaries from the same sources as the source data set. We assumed the benign PE binaries are benign in the case of vulnerability and malware. The benign examples we chose for the target data set were different than those for the source data set.

Most previous research into vulnerability detection has used source code rather than PE binaries [11, 24, 25, 22]. Many source code data sets are not guaranteed to compile [24]. Since

we need vulnerable PE binaries rather than source code, we chose the NIST SARD due to its compilation guarantee [24, 31].

The NIST SARD contains more than 450,000 test cases. Within the NIST SARD, there are also dozens of test suites, which are curated sets of vulnerable test cases. The NIST SARD labels all test cases by common weakness enumeration (CWE). CWE specifies the specific type of vulnerability represented by the code. For example, CWE-126 is a buffer over-read. There are nearly a thousand CWEs [32], over 150 of which are represented in the NIST SARD [31].

We used all the accepted test cases from the year 2017, the most recent year a test case was accepted. Each test case's language is either C, C++, C#, Java, or PHP. We only considered test cases written in C, since C was the most common language for accepted test cases in SARD in 2017. C is also a widespread language in vulnerability detection data sets [24, 22, 10], which may aid in comparison in future experiments.

There were 28 different CWEs within the test cases we selected. However, only six successfully compiled a large enough number of their test cases. Table 1 presents the specific CWEs and the number of compiled examples. We used MinGW to compile the source code into a window executable on a Linux machine. We compiled all source code into Windows binaries since the pre-compiled PE binaries were also Windows binaries.

CWE Identifier	Description	Number Compiled
121	Stack-based Buffer Overflow	453
122	Heap-based Buffer Overflow	1900
124	Buffer Underwrite (Buffer Underflow)	280
127	Buffer Under-read	280
78	Improper Neutralization of Special Elements used in an OS Command (OS Command Injection)	2057
789	Memory Allocation with Excessive Size Value	372

Table 1: The six chosen CWEs, their identifier, and their name.

3.2 Data Representations

Much of this research compares the efficiency of different data representations in producing successful transfers from malware data sets to vulnerability data sets. Previous research has investigated some of the following data representations more than others. The single-channel byte image representation, in particular, attracts a large amount of interest.

The number of generated cases for each data representation differs due to the capabilities of the feature-generating algorithms. We were able to generate byte images and EMBER features for all binaries. However, not all PE binaries could be processed into the assembly instruction count and graph embedding representations, since they require require CFG recovery.

3.2.1 Single-Channel Byte Images

The single-channel byte image data representation of PE binaries is common when applying machine learning methods to malware analysis [18, 33, 1, 34, 7, 3, 8, 9]. Experiments using an

CWE	Single-Channel Images	EMBER	Instruction Counts	Graph Embeddings
121	453	453	453	130
122	1900	1900	1842	540
124	280	280	280	87
127	280	280	278	61
78	2057	2057	1570	426
789	372	372	223	65

Table 2: The number of examples generated per CWE for each data representation.

image data representation report a high accuracy on malware analysis tasks. However, these studies have not tested the ability to transfer between malware detection and vulnerability detection.

We create a single-channel image by reading the individual bytes of the PE binary one-by-one. Each ith byte corresponds to a scalar $x_i \in [0, 255]$. The value of a byte will be between the binary values 00000000 and 11111111, which corresponds to the decimal values 0 and 255. All x_i together are equivalent to a vector **x**, where the length of **x** is the number of bytes in the binary. Then, based on the size of the image (i.e., the length of **x**), the vector **x** is reshaped to have a width as defined in Table 3 and height as dictated by the number of bytes. For example, a 230 kB file will have a width of 384 and a height of $ceil((230 * 1024)/384) = 614$. Therefore the reshaped vector will have the shape $(614, 384)$. The number of bytes may not cleanly divide into the width. If so, we pad the rest of the last row with zeros to keep the image rectangular.

File Size (kB)	Width
< 11	1
$11 - 30$	32
$31 - 60$	64
$61 - 100$	128
$101 - 200$	256
$201 - 500$	384
$501 - 1000$	512
1001	768
> 1001	1024

Table 3: The size of binaries and corresponding width used in image creation.

The resulting matrix creates an image where every element corresponds to a pixel. The size of the binary dictates the image size. The image only has a single channel. Refer to Figure 1 for examples of benign, malicious, and vulnerable images.

Resizing. Before training or inference by our models, the images undergo further resizing to 299x299. We used bilinear interpolation to calculate the values of new pixels when resizing. Bilinear interpolation performs linear interpolation along the x axis followed by linear interpolation along the y axis.

Linear interpolation calculates the value $f(x_2)$ of a point x_2 given two points x_0, x_1 with known values $f(x_0), f(x_1)$. The presumption of linear interpolation is that the slope is the same

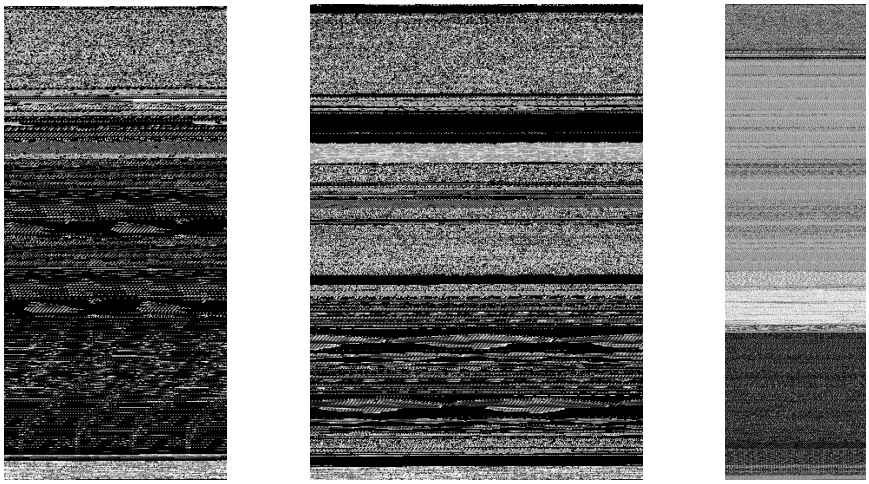

Figure 1: Examples of benign (left), malicious (center), and vulnerable (right) images created from binaries. Their heights are equivalent for visualization here.

between the first point and the unknown point and the unknown point and the second point. Thus, $\frac{f(x_2) - f(x_0)}{x_2 - x_0} = \frac{f(x_1) - f(x_2)}{x_1 - x_2}$, which reduces to:

$$f(x_2) = f(x_1)\frac{x_2 - x_0}{x_1 - x_0} + f(x_0)\frac{x_1 - x_2}{x_1 - x_0}$$

Given a scalar function over two variables $f(x, y)$, and four points with known values, which together form the points of a rectangle, $(x_0, y_0), (x_0, y_1), (x_1, y_1), (x_1, y_0)$, bilinear interpolation finds the value $f(x_2, y_2)$ of a fifth point (x_2, y_2) contained within those four points. First we perform linear interpolation along the x-axis as below.

$$f(x_2, y_1) = f(x_1, y_1)\frac{x_2 - x_0}{x_1 - x_0} + f(x_0, y_1)\frac{x_1 - x_2}{x_1 - x_0}$$

$$f(x_2, y_0) = f(x_1, y_0)\frac{x_2 - x_0}{x_1 - x_0} + f(x_0, y_0)\frac{x_1 - x_2}{x_1 - x_0}$$

Second, we perform linear interpolation along the y-axis.

$$f(x_2, y_2) = f(x_2, y_1)\frac{y_2 - y_0}{y_1 - y_0} + f(x_2, y_0)\frac{y_1 - y_2}{y_1 - y_0}$$

3.2.2 EMBER Features

Anderson et al. (2018) created a standardized data set for malware detection; the Endgame Malware BEnchmark for Research (EMBER) [16]. They defined a data pipeline which intakes a collection of malicious and benign binaries and, for each binary, creates a feature vector. We will use the EMBER data creation pipeline to generate features from our source and target data sets.

Anderson et al. groups the raw EMBER features into parsed features and format-agnostic features [16]. The parsed features include general file information and header information. The

format-agnostic features include a byte histogram (i.e., counts of how many times a specific byte value appears) and information on printed strings.

The EMBER data pipeline does not output raw features, rather it creates the raw features as an intermediate step before vectorization. The final vector has a dimension of 2381. The vectorization process occurs through feature hashing, which reduces the dimensionality and sparsity of a vector [16, 35].

Feature Hashing. Consider a case where a group of features can each take on one of a set number of distinct values. An example is a sentence where each word in the sentence can take on the value of any word in a set vocabulary. Researchers can count the number of occurrences of each word in the sentence and generate a vector of the size of the vocabulary for each sentence. These vectors are often sparse and increase rapidly as the data set grows.

Feature hashing addresses both the sparsity and large size of these vectors. It uses two hash functions. The first hash function maps from the initial vector dimension to the desired, reduced dimension. The second hash function maps from the initial vector dimension to $\{-1, 1\}$. The first function uses the hash values as indices directly in a vector embedding in the desired reduced dimension [35].

Formally consider m features that can take on n distinct values. We create a feature vector $\mathbf{x}$ of size n by performing a Boolean mapping of these features. Next, consider a vector $\mathbf{y}$ with the desired dimensionality k, and two integers $i \in \{0, ..., n-1\}$ and $j \in \{0, .., k-1\}$. Given two hash functions $h : \{0, ..., n-1\} \to \{0, .., k-1\}$ and $g : \{0, ..., n-1\} \to \{-1, 1\}$, $\mathbf{y}_j = \sum_{i:h(i)=j} \mathbf{x}_i g(i)$. For a specific example, consider Figure 2. Anderson et al. used a similar process in the EMBER features data pipeline, but they do not explicitly provide the feature map [16].

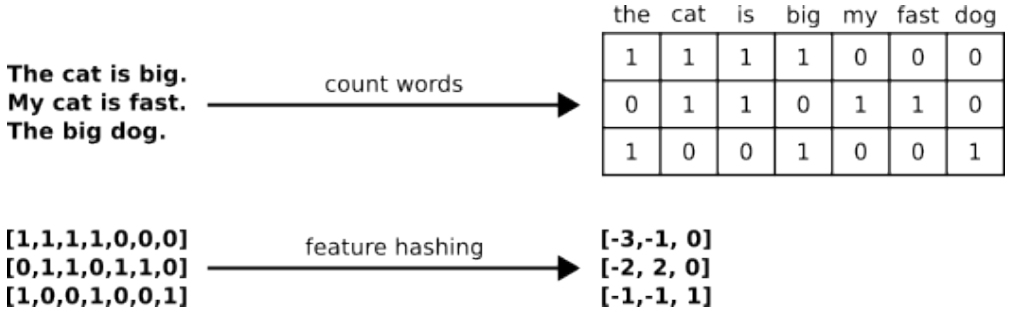

Figure 2: An example of feature hashing. Consider three sentences with a vocabulary of seven words. After we create vectors by counting the words in each sentence, the feature vectors are sparse and of dimension 7. Using the simple hashing functions $h(i) = floor(i/3)$ and $g(i) = sgn(i-3)$, we can construct new feature vectors that have reduced dimensionality (7 to 3) and sparsity. The original vectors are zero in about 48% of the original elements but only in about 22% of the new elements.

We created a t-distributed stochastic neighbor embedding (t-SNE) of dimension two from the EMBER features to visualize how well they cluster. A t-SNE is a statistical technique for displaying high dimensional data in two- or three-dimensional space. In Figure 3, we present a plot of the t-SNE, which shows the potential for malware and vulnerability detection using the EMBER features. There is an overlap in the source data set, but it is not an exact overlap. There is no overlap in the target data set. We considered all vulnerable CWEs in the target data set for the t-SNE.

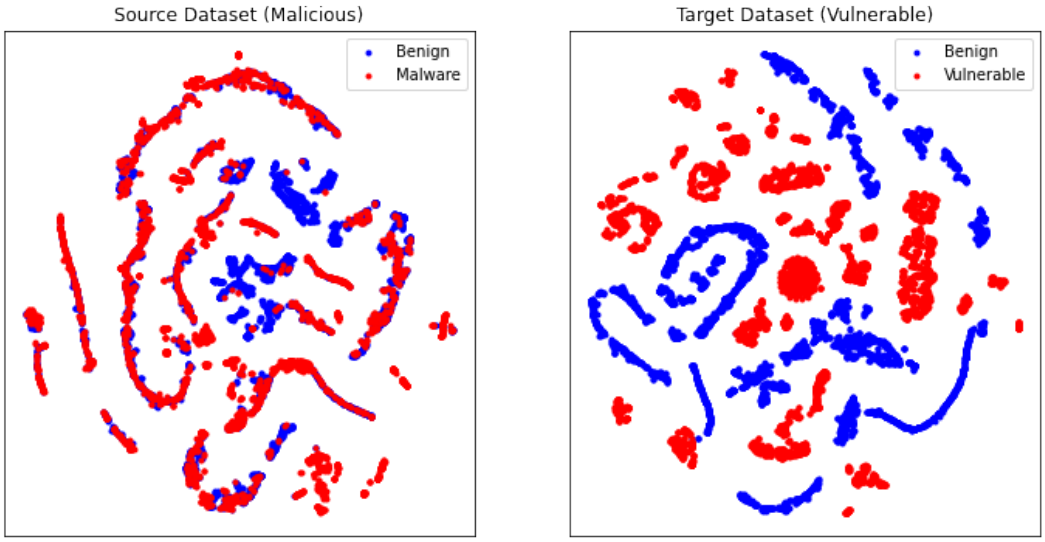

Figure 3: A 2-dimensional t-SNE of the EMBER features.

3.2.3 Control Flow-Graphs

We generated the final three data representations by first creating CFGs. We used the disassembler from the angr library [36] to take machine instructions from each binary and create a CFG. A CFG contains as its nodes basic blocks of assembly instructions (i.e., functions) with directed edges indicating an exit from one block and entrance to another (e.g., jumps, calls, and returns [37]). Refer to Figure 4 for an example of a portion of a CFG.

Previous experiments used assembly instruction counts in malware [5, 4] and vulnerability detection [26, 27]. However, they do not explore further embeddings than frequency and use limited architectures to investigate accuracy improvements. Therefore, transfer from malware to vulnerability detection with assembly instruction counts presents a novel opportunity for research.

We analogize counts of assembly instructions in a binary to counts of words in a document. It is impossible to generate a meaningful order of assembly instructions since there is no guaranteed execution order for instructions in a binary. The exact order of execution is dependent the system rather than the binary. Thus, we only use embeddings which are independent of instruction order.

We evaluate three representations from the CFGs. The first is term frequency-inverse document frequency (tf-idf) weighted instruction counts, the second is doc2vec distributed bag of words (DBOW) embedded instruction counts, and the third are graph2vec graph embeddings.

Tf-Idf. A tf-idf embedding considers every instruction's frequency in each binary and inverse frequency across the data set. The more common an instruction is in a specific binary, the more important it is to the identification of that binary. The more common it is across all the binaries in the data set, the less important it will be when classifying binaries.

Formally, a tf-idf embedding consists of tf (term frequency) and idf (inverse document frequency). Given a set of documents D, a document from that set $d \in D$, a vocabulary W, and a word from that vocabulary w, we can define a tf-idf value for a particular word in a docu-

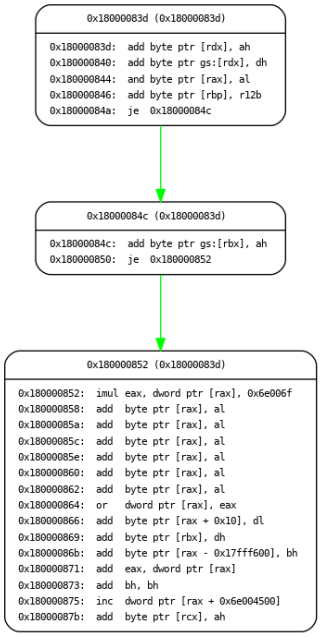

Figure 4: A example portion of the CFG of the binary with id 200267 from the source data set. The second column in each node contains assembly instructions. We created this visualization using the angr-utils Python library.

ment as $tfidf(w, d, D) = tf(w, d)idf(w, D)$. We first define the term frequency $tf(w, d) = \dfrac{f_{w,d}}{len(d)}$ where $f_{w,d}$ is the raw count of the number of occurrences of that word in the given document and $len(d)$ is the total number of words in the document. Next, we define the inverse document frequency $idf(w, D) = log\left(\dfrac{len(D)}{len(\{d \in D : w \in d\})}\right)$ where $len(D)$ is the total number of documents and $len(\{d \in D : w \in d\})$ is the number of documents where w appears. We calculate the tf-idf value for each word in the document using these formulae. For a specific example, consider Figure 5.

For transfer learning to occur, the tf-idf embeddings of both the source and target data sets must have the same dimension, since we do not vary the input dimension of the models across samples. Thus, our experiments use the same vocabulary for both. We use the source data set vocabulary since all instructions from the target are present in the source data set.

The tf-idf embeddeding has a dimensionality of 759. Every feature is non-zero in atleast one vector in the source data set. Only 684 features have non-zero entries in the target data set. The target data set is less sparse, with 0.574 non-zero entries compared to 0.114 in the source data set.

We created a t-SNE of dimension two from the tf-idf embedding to visualize how well it clusters. Refer to Figure 6 for a plot of the t-SNE. The t-SNE plot shows potential for malware and vulnerability detection. There is an overlap in the source data set; however, it is not an exact overlap. There is no overlap in the target data set. We considered all vulnerable CWEs for the target data set for the t-SNE plot.

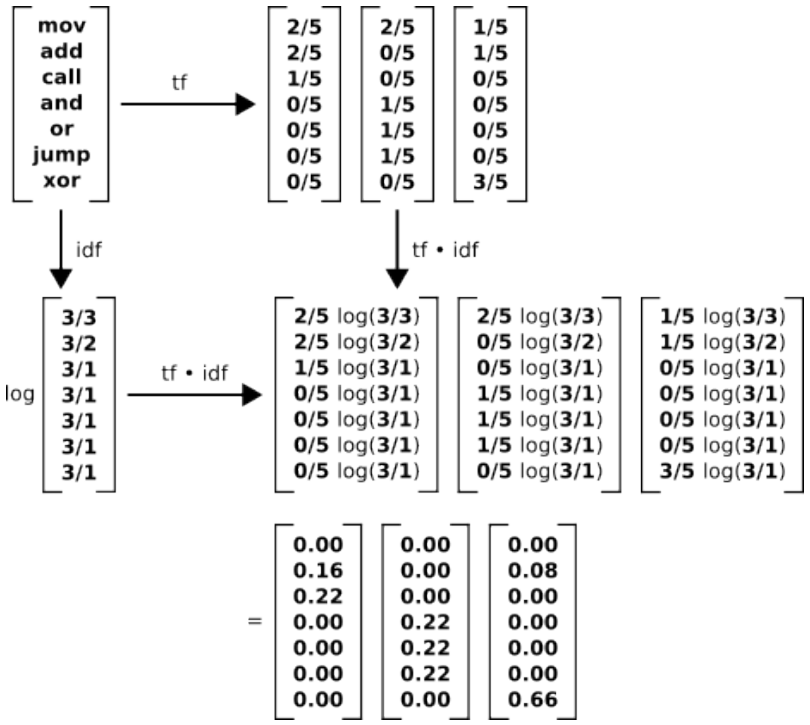

Figure 5: An example of a tf-idf embedding. Consider three assembly instruction sequences **mov add add mov call**, **mov mov and or jump**, and **xor add xor xor mov**. These three sequences have a total vocabulary length of seven. The first vector corresponds to the first sequence, second vector to the second sequence, and third vector to the third sequence. The tf values are computed for each instruction and sequence. The idf values are computed for each instruction. The final tf-idf vector results from the element-wise multiplication of each tf vector with the idf vector.

Doc2Vec. The doc2vec DBOW algorithm [38] projects a document onto a vector of a desired dimensionality and predicts randomly sampled words using the projection [38]. Randomly sampling words fulfills our requirement for the document embedding vector to be word-order agnostic.

A related method to DBOW is distributed memory (DM), which learns word and document embeddings from a context window. Collectively DBOW and DM are known as doc2vec [38]. We so not use the DM method of document embedding since the context window necessitates word order. The two doc2vec methods are extensions of two methods for embedding words in a document as vectors, termed word2vec [39]. The word2vec continuous bag of words (CBOW) model predicts a word given the surrounding context. The word2vec skip-gram model predicts the surrounding context from a given word. The skip-gram and CBOW models also rely on context windows and thus we did not use them in our research.

We used a dimension of 64 to compare the doc2vec embedding to the graph embedding data representation. We trained the doc2vec model for 50 epochs. We created a t-SNE of dimension two from the DBOW embeddings to visualize how well they cluster. Refer to Figure 7 for a plot of the t-SNE. There is little clustering and much overlap between the target and source data sets. We do not expect the doc2vec embedding will produce good results. We consider all vulnerable CWEs for the target data set.

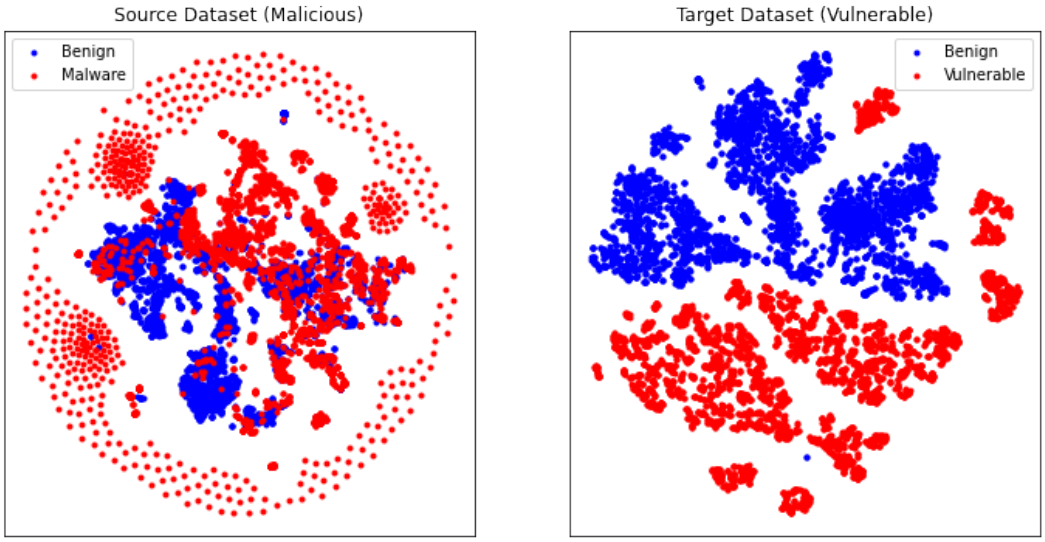

Figure 6: A 2-dimensional t-SNE of the tf-idf embedding.

Graph Embeddings. We created a graph embedding of the PE binary by considering the code blocks as nodes and their connections as edges. An algorithm, graph2vec [40], generated embedding vectors from the graph structure. The graph2vec algorithm is similar to the doc2vec DBOW model. Instead of randomly sampling words from a document to learn an embedding of the document, the graph2vec algorithm randomly samples rooted subgraphs from a graph to learn an embedding of the graph [40].

A rooted subgraph is the subgraph of all nodes and related edges reachable from a root node n within d hops. An example of a rooted subgraph is the CFG in Figure 4, containing only three nodes and two edges. A vocabulary of rooted subgraphs is the set of all rooted subgraphs of all the graphs considered. Graphs with similar rooted subgraphs will have closer embeddings and graphs with dissimilar rooted subgraphs will have dissimilar embeddings [40].

Our research partners at MSU provided us with the graph embedding data representation vectors for the source data set. These contained only 1860 malicious and 1860 benign vectors of dimension 16, 32, and 64. We used the 64-dimension vectors to ensure that the largest amount of information was captured by the vector features. We presented our target data set to the graph2vec algorithm to generate 64-dimensional vectorized representations of the CFGs. We only used the 64-dimension graph embedding to ensure that the source data set and target data set had the same embedding dimension. We created only 1600 target vectors due to time constraints in generating CFGs. The vectorized CFGs followed the same distribution as the binaries (i.e., a specific CWE with more binaries successfully generated more g2v vectors).

We created a t-SNE of dimension two from graph embeddings to visualize how well they cluster. Refer to Figure 8 for a plot of the t-SNE. There is an overlap in the source data set; however, it is not an exact overlap. There is no overlap in the target data set. Contrast the t-SNE of the graph embeddings with the t-SNE of the DBOW embeddings. The graph embeddings cluster more despite both extending the word2vec skip-gram model. We considered all vulnerable CWEs for the target data set in the t-SNE plot.

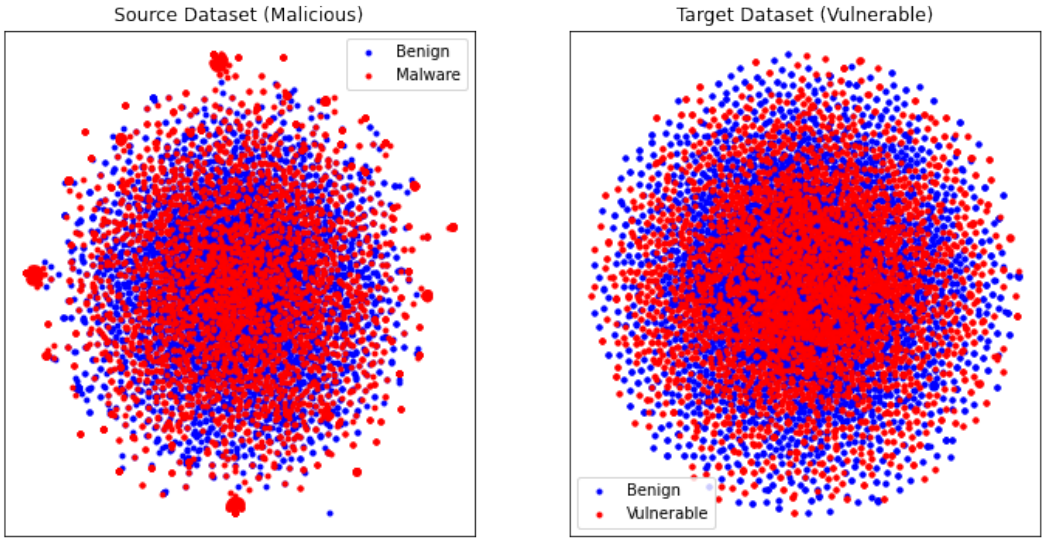

Figure 7: A 2-dimensional t-SNE of the doc2vec DBOW embedding.

3.3 Architectures

We tested our data with four learning approaches, k-nearest neighbors (k-NN), feed-forward neural networks, convolutional neural networks (CNN), and long short-term memory (LSTM) networks. We trained all data representations on the k-NN, feed-forward, and CNN models. Since the single-channel image data representation is not a vector embedding, but rather a 2-D image, we did not train it on an LSTM. We trained all data representations other than the images on the LSTM model.

We implemented all of our models using PyTorch Lightning, except the k-NN algorithm, to utilize early stopping and to greatly simplify each model's code. We used the pre-implemented k-NN model in scikit-learn.

3.3.1 K-NN

K-NN evaluates a vector with an unknown label on a data set of vectors with known labels. It compares the unknown vector to the k closest labelled vectors. The majority label of the nearest neighbors determines the label of the unknown vector. The value of k is typically odd so a majority will always be reached.

We presented the data to the k-NN classifier as a flattened array of floats for every data representation. Flattening a feature set to use in a k-NN classifier is inaccurate, especially in the case of images. In image data neighboring pixel values depend on each other and flattening an image removes that information. Some research studies create other feature sets from their image data sets to use with k-NN models [7, 1, 34]. Since maintaining data representations is essential for our purposes, we used flattened image matrices.

In order to compare to cases of transfer learning, we fit a k-NN model to our source data set and then determine the labels of our target data set. We vary the value of k between 1 and 200. We use the k-NN results as a baseline, which is common in previous research [7, 1, 34].

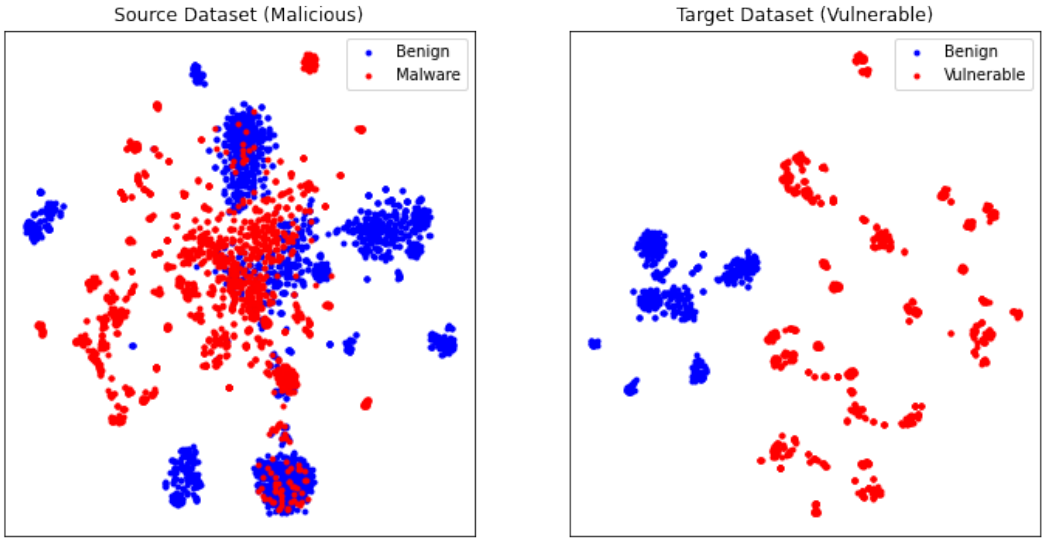

Figure 8: A 2-dimensional t-SNE of the graph embeddings.

3.3.2 Feed-forward Networks

We used a feed-forward neural network for all data representations. Feed-forward networks are neural networks in which information flows directly from the input to the output, typically through a series of hidden layers. Each layer has a linear transformation and a non-linear element-wise function called an activation function. The activation function allows the feed-forward network to approximate non-linear functions.

There are many choices of activation function. We present a few examples in Table 4. We chose ReLU activations because they are much faster than other activation functions. A ReLU only needs a single if statement for computation, but both a sigmoid activation and a hyberbolic tangent activation require computing exponents. Specifically, we chose the LeakyReLU variant of the ReLU function. A basic ReLU may experience the dying ReLU problem. A node in a network can reach a state where it will return negative values for all inputs. The ReLU activation will always return zero after the node and no gradients will flow backward through the neuron. This node effectively dies and becomes inactive. However, introducing a slight positive gradient to negative inputs mitigates neuron death. A leaky ReLU is near-zero for negative values but not exactly zero.

Activation	**Function**
Sigmoid	$f(x) = \dfrac{1}{1+e^{-x}}$
Hyperbolic Tangent	$f(x) = \dfrac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$
Rectified Linear Unit (ReLU)	$f(x) = \begin{cases} x, & x > 0 \\ 0, & x \leq 0 \end{cases}$

Table 4: Three examples of activation functions.

Our network has five hidden layers, each of dimension 10. The architecture design is `input_dimension` x 10 x 10 x 10 x 10 x 10 x 2. The input dimension varies depending on the dimension of the data set presented to the model (e.g. 64 for the graph embedding and DBOW embedding and 759 for the tf-idf embedding). Feed-forward networks often produce state-of-the-art results, however they are a relatively simple architecture. Successful transfer with feed-forward networks would present good evidence for the future success of malware-to-vulnerability transfer with other, more specifically designed, architectures.

3.3.3 Le-Net Style CNN

A CNN is comprised of convolutional layers containing kernels that convolve over the input to a layer to produce outputs. CNNs are known to perform exceptionally well on images since they take neighboring information into account. Other learning methods, such as feed-forward neural networks and k-NN, do not take neighboring information into account. There are many possible CNN architectures. The LeNet-5 architecture design influenced the specific CNN architecture used in our research [41]. Refer to Figure 10 for a diagram of our specific convolutional architecture.

Our network is comprised of five convolutional layers. A convolution layer operates by taking a matrix of values, called a kernel, and moving them across the input to that layer. At each step, the kernel will fall on a portion of the input the same size as the kernel. A CNN element-wise multiplies the kernel and the portion of the input the kernel overlays. It then adds them together and assigns them to a corresponding location in the output. The CNN learns the values of the kernel through training. Refer to Figure 9 for an example of a convolution.

A convolution may pad the border of the input or vary the kernel's step size (i.e. stride) as it moves across the input. In our architecture, the kernels are of size five, with no padding and a stride of one. We added three more convolutional layers, since our images (299 x 299) were larger than LeNet's images (28 x 28). A non-linear activation function follows each convolution to ensure that the CNN architecture can model non-linear functions as in the feed-forward networks. We again chose a LeakyReLU as our activation.

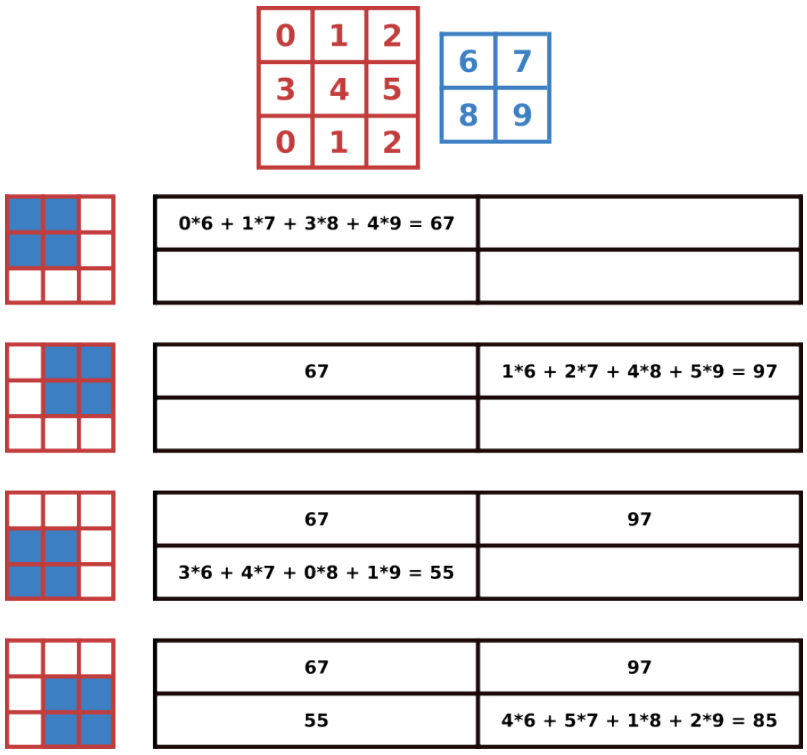

Figure 9: An example of a kernel convolution of over an input. Consider the red input and blue kernel. As the blue kernel is moved over the red input, an output matrix is constructed from a linear combination of the elements in the kernel and the area in the input which the kernel lies over.

We place a pooling layer after each activation. A pooling layer consists of a kernel which is passed over an input similar to a convolution. However, a pooling layer does not have any kernel elements and performs a pre-defined operation over the input elements which lie under the kernel.

Two common pooling layers are average pooling and max pooling. Average pooling calculates the mean of the input elements and max pooling calculates the maximum of the input elements. A pooling layer may pad the input to the kernel or change the kernel's stride like a convolutional layer. We used max pooling rather than average pooling in our pooling layers. Max pooling is preferred over average pooling since max pooling extracts the most important features. In contrast, average pooling smooths all the features together and may lose important nuances identified by the convolution.

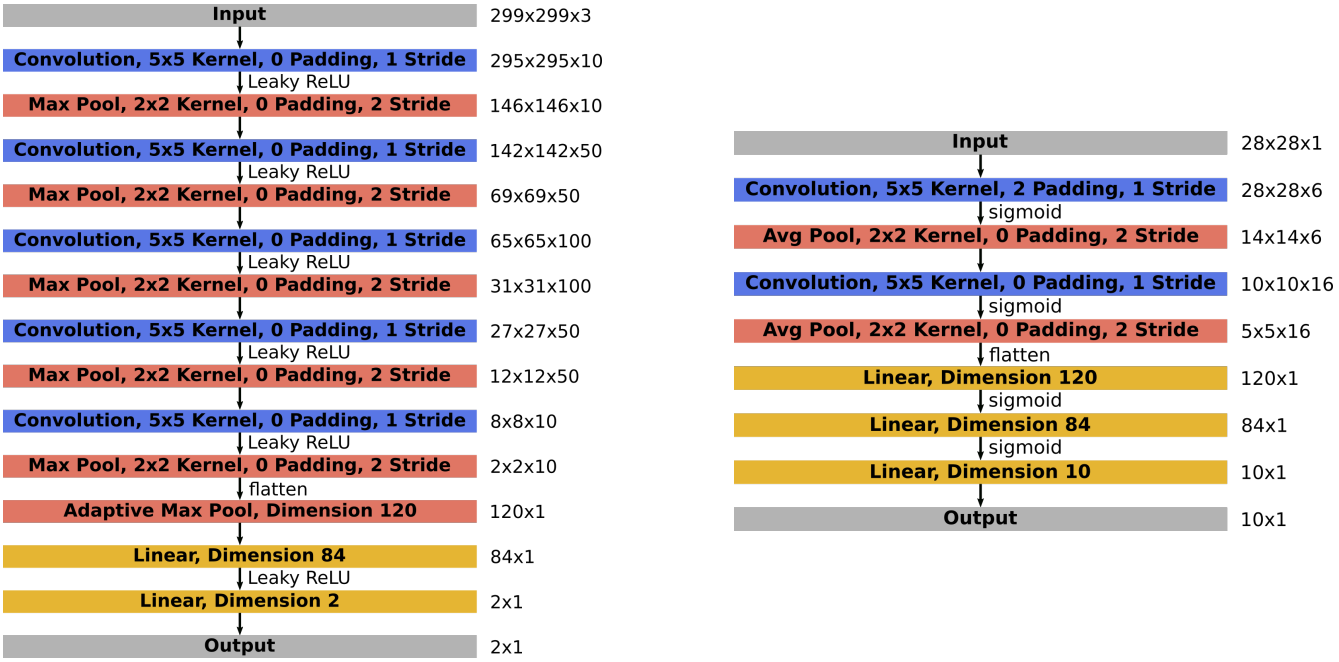

Figure 10: Our CNN implementation for images (left) alongside the LeNet implementation (right). The output data size for each layer is to the right of the layer.

The 2-D convolutions used with the single-channel image data representation do not work with non-image data. The single-channel image data representation has the shape 299x299, but all other data representations have the shape `vector_dimension`x1. We changed the CNN model architecture to account for the non-image data representations by using 1-D max pooling and 1-D convolutions. We used a zero-padding of two for the the 1-D max pooling and 1-D convolution to prevent the feature dimension from decreasing as the convolutions occur. We also used an adaptive max pooling layer, instead of the first 120-dimension linear dimension in LeNet, to account for different input dimensions.

3.3.4 LSTM[42]

An LSTM network is a type of recurrent network (RNN). RNNs are a broad class of networks that maintain looped connections (i.e., the output from a node can affect the same node at future time steps). We use the input vectors directly rather than including an embedding layer in our LSTM implementation. The input vectors are embedded during pre-processing into different data representations. We do not use the single-channel image data representation with the LSTM architecture.

We used LSTMs rather than other variants of RNNs because they are good at remembering information across many time-steps. Our LSTM inputs are not sequences, but are streams of malware. Streaming malware mimics the actual process that an organization may go through in detecting malware. A stream of information will come in, and the model will have to classify each new piece of information as it arrives. In this case, the LSTM will learn the initial inputs to the system and use that to generate vectors for the later inputs.

We did not present the data to the LSTM as a batch of a series of vectors. We presented the LSTM a matrix where the batch size is equivalent to sequence length, and each sequence element is the embedded vector. The model processes the data in a final linear layer after it

presents the data to the LSTM.

3.4 Parameters

Cross Validation. We performed a 10-fold cross-validation in our research in which we held 10% of the target data aside for testing and used 90% for fine-tuning the model. In each fold we varied the particular cases in the testing and training data set so that no two folds had the same testing cases. We used the mean accuracy across all folds as our preferred metric since the labels are evenly split in every set. We averaged the fold accuracies to prevent bias of the model on any one specific case or set of cases.

Transfer Learning. Our research used two types of transfer learning.

The first consists of freezing all the layers of the source network other than the last fully-connected layer where all learning occurs. Layer freezing is faster than other types of transfer learning since only the unfrozen layers are updated during fine-tuning.

The second involves freezing no layers and retraining all the networks' weights. Retraining all weights is more computationally expensive than retraining some of the weights. However, since target data sets are typically smaller than source data sets, fine-tuning will be quicker than pre-training. In this case, the pre-trained network may provide a better starting weight initialization for training than random weights.

Regularization. Regularization prevents a model from overfitting on a data set. Overfitting occurs when the model memorizes specific cases in a data set, and thus loses generalizability, which is a goal of machine learning.

All our models used batch normalization on the fully-connected layers as a method of regularization. The inputs to a model are presented to a model in batches, since the size of the training data often exceeds the RAM of the machine hosting the model. Batch normalization computes each batch's mean and variance, then subtracts the mean and divides by the square root of the variance. We add a small constant to the variance before taking the square root, which is standard practice [43].

Batch normalization provides many benefits, including faster convergence, higher accuracy in classification tasks, and regularization [43]. Regularization occurs because both the mean and variance are tied to the specific batch and will have some variability compared to other batches. This produces the effect of adding a small amount of noise into each batch.

When performing batch normalization, it is unnecessary also to use other forms of regularization such as dropout [43]. Dropout randomly zeroes input elements with a pre-defined probability [44]. Previous research also suggests that dropout is ineffective for regularization with CNNs [1]. We did not use batch normalization with our LSTM model since the variable length of inputs to LSTMs makes normalization unreliable. Applying batch normalization horizontally within an RNN (i.e., between time-steps) would also be untenable due to exploding/vanishing gradients [45].

Learning Rate Scheduler. We used an initial learning rate of 0.001 and an Adam optimizer with default PyTorch β values for all models. We also used a step learning rate scheduler, which decays the learning rate by a parameter after a specified number of epochs. Our research used a decay parameter of 0.1 and an epoch of 1 for the learning rate scheduler.

Early Stopping. Our research utilized the early-stopping callback feature of PyTorch Lightning. PyTorch Lightning callbacks are self-contained programs that run at a specific point in the flow of execution [46]. For example, a callback could print a message at the end of training. We used the callback feature to implement early stopping, which stops training once the loss function has reached a minimum. We used cross entropy as our loss function, which is standard in classification tasks.

4 Results

We presented each model seven different percentages of the target data set, 0%, 1%, 10%, 20%, 40%, 70%, and 100%. We expect that the models will have a low classification accuracy when presented with 0% of the data. We performed inference on the models without fine-tuning to evaluate the performance at 0%. We expect the model to perform better as we present it more data, with the highest accuracy on 100% of the data. We performed 10-fold cross-validation for each percentage of the target data set. Since we examined six CWEs, four data representations for the LSTM, and five data representations for the other three architectures, we performed 7980 total experiments.

4.1 Single-Channel Images

4.1.1 k-NN

K-NN performed the second worst on the image data representation. The highest accuracy achieved using k-NN was with $k = 4$, where the accuracy ranged from 0.852 to 0.868. We present the image k-NN results in Table 5.

	121	**122**	**124**	**127**	**78**	**789**
$k = 1$	0.770	0.794	0.768	0.795	0.787	0.786
2	0.845	0.861	0.85	0.857	0.857	0.862
3	0.804	0.817	0.811	0.830	0.812	0.816
4	0.857	0.864	0.852	0.868	0.859	0.866
5	0.660	0.680	0.646	0.670	0.669	0.675
10	0.724	0.727	0.698	0.718	0.723	0.728
25	0.727	0.719	0.700	0.704	0.723	0.731

Table 5: The accuracy using k-NN to transfer information from the source data set to the target data set for a flattened single-channel image.

4.1.2 Feedforward

The no-freezing feed-forward networks produced positive transfer, whereas the freezing feed-forward networks produced negative transfer. All CWEs either matched or exceeded the best-case k-NN models. The transfer accuracy for the freezing networks in Figure 11 remains around 0.5, but the no-transfer accuracy increases. The non-freezing accuracy occasionally reaches state-of-the-art results. For example, the non-freezing accuracy for CWE 122 is 0.958 at 1.0 of the target data set.

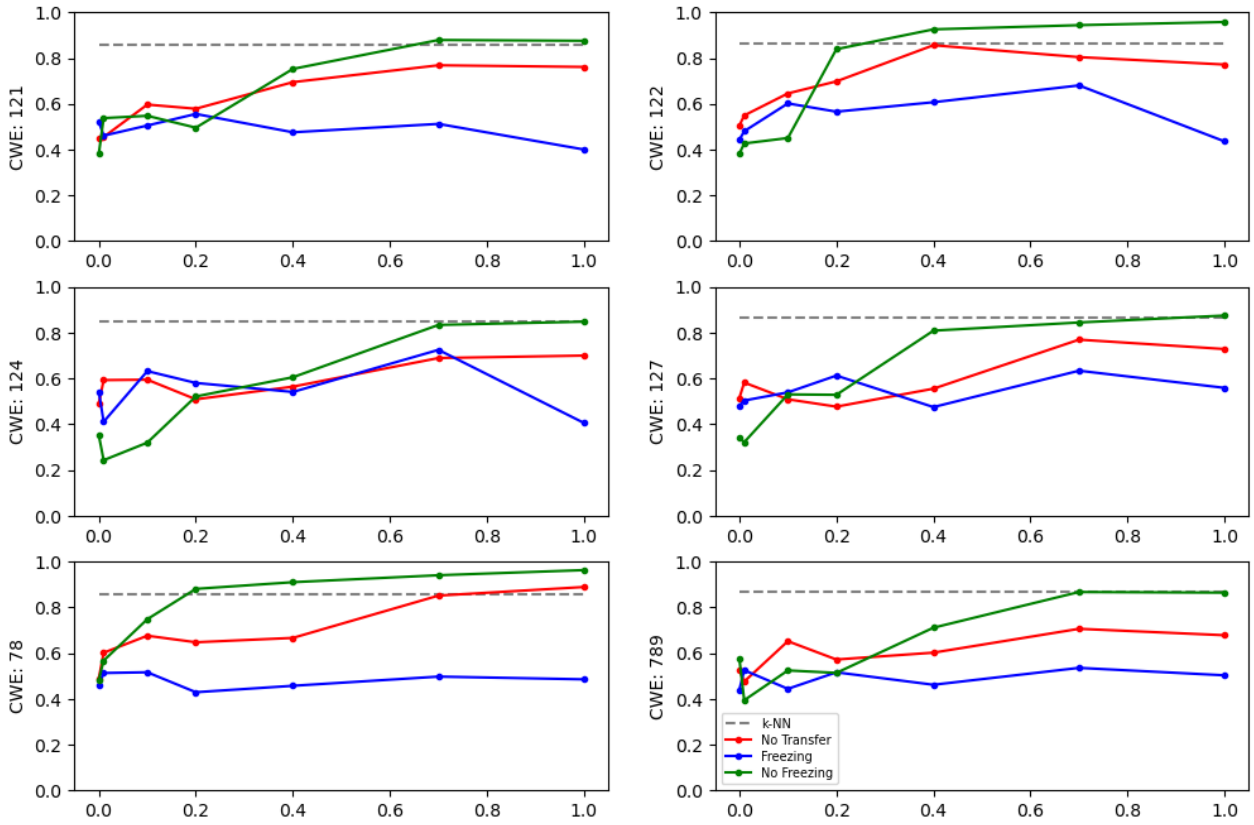

Figure 11: The transfer learning results on the feed-forward model with freezing and without freezing for all six CWEs. We compare the accuracy (y-axis) against the percentage of the data set presented to the model (x-axis). The k-NN accuracy represents the k-value with the best results (in this case, 4).

4.1.3 CNN

The CNNs produced negative transfer for the freezing networks and positive transfer for the non-freezing networks. The freezing networks were successful at producing non-random accuracies in some cases (e.g. CWE 122 with accuracy (0.867)). Yet, the no-transfer case had a superior accuracy for all freezing models. The non-freezing models always had an initial positive transfer, with all models converging to a near 1.0 accuracy when presented with 1.0 of the target data set. Refer to Figure 12 for all single-channel image CNN results.

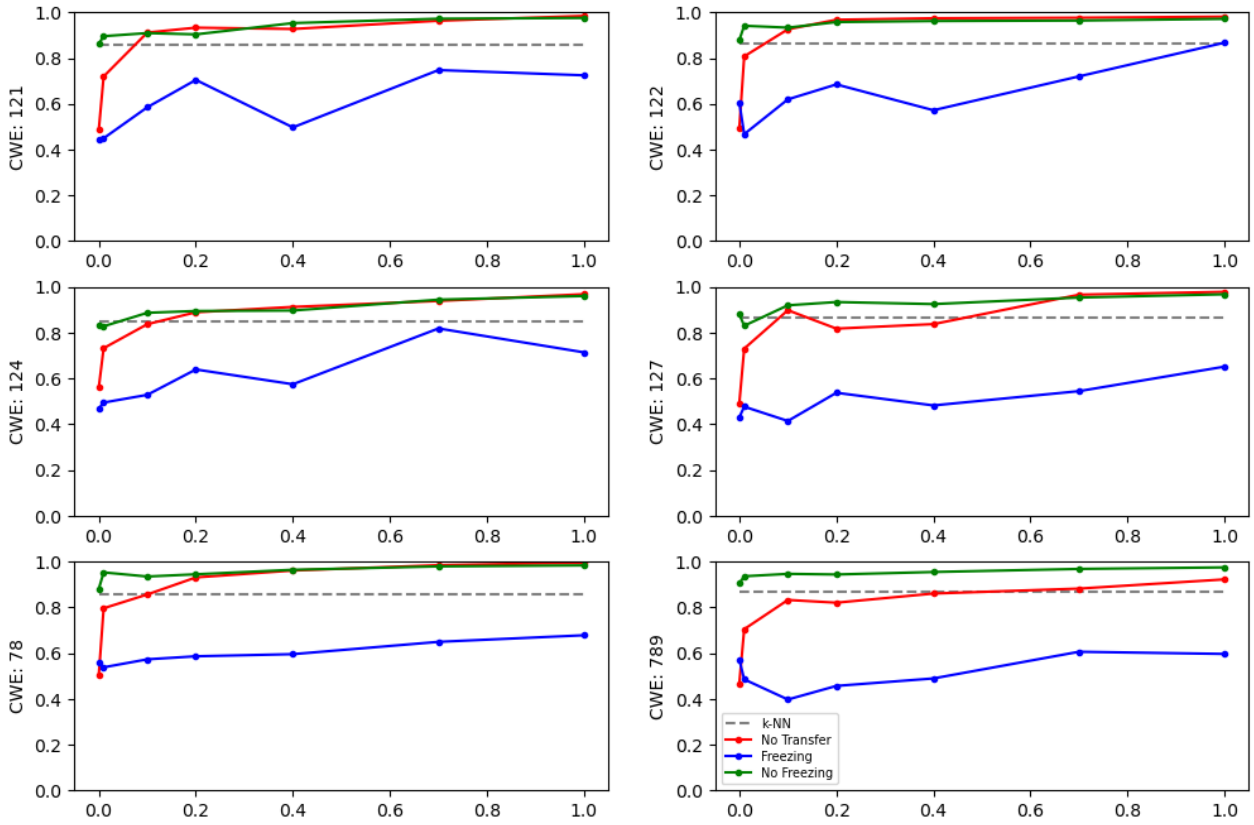

Figure 12: The transfer learning results on the CNN model with freezing and without freezing for all six CWEs. We compare the accuracy (y-axis) against the percentage of the data set presented to the model (x-axis). The k-NN accuracy represents the k-value with the best results (in this case, 4).

4.2 EMBER Features

4.2.1 k-NN

The k-NN model performed the best when $k = 20$. The accuracies for $k = 20$ ranged from 0.902 to 0.913. Refer to Table 6 for all k-NN results for the EMBER features data representation.

	121	**122**	**124**	**127**	**78**	**789**
$k = 1$	0.435	0.436	0.423	0.436	0.544	0.618
3	0.429	0.433	0.439	0.434	0.544	0.620
10	0.423	0.437	0.427	0.430	0.536	0.622
18	0.414	0.431	0.418	0.416	0.530	0.608
19	0.900	0.901	0.907	0.898	0.897	0.895
20	0.912	0.911	0.913	0.902	0.909	0.903
50	0.894	0.889	0.900	0.891	0.895	0.888

Table 6: The accuracy using k-NN to transfer information from the source data set to the target data set for the EMBER features data representation.

4.2.2 Feedforward

The feed-forward network freezing cases in Figure 13 all performed poorly for the EMBER features. Occasionally the non-freezing cases produced a positive transfer, such as CWE 78 where the transfer accuracy (0.829) exceeded the no-transfer accuracy (0.612). All models have a high accuracy when trained on 0.0 of the data set. Upon training the model the accuracies decreased, often to below random. The model performance only returned to above random in the case of CWE 78 and CWE 122.

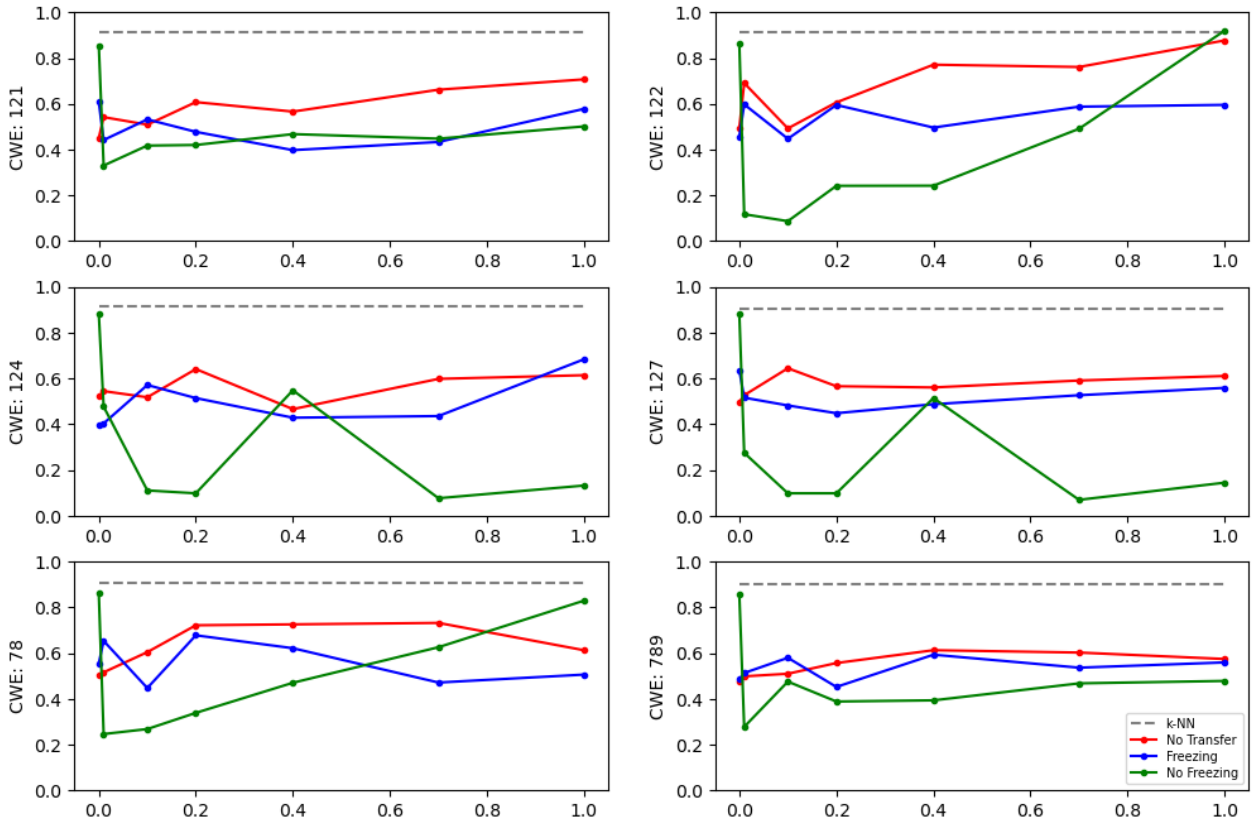

Figure 13: The transfer learning results on the feed-forward model with freezing and without freezing for all six CWEs. We compare the accuracy (y-axis) against the percentage of the data set presented to the model (x-axis). The k-NN accuracy represents the k-value with the best results (in this case, 20).

4.2.3 CNN

The CNN models were more effective than the feed-forward with the EMBER features. The freezing cases never had a positive transfer, but some had high accuracies (e.g. CWE 122 with accuracy (0.931)). We observed the same non-freezing negative learning as in the feed-forward networks. However, no CNN model had an initial high accuracy. Some models recovered from the below-random accuracy (e.g. CWE 122), but others did not (e.g. CWE 121, 124, and 127). CWE 78 and CWE 789 did not have the below-random accuracy drop in initial training. Both models also showed a positive transfer compared to the no-transfer case. Refer to Figure 14 for all EMBER CNN results.

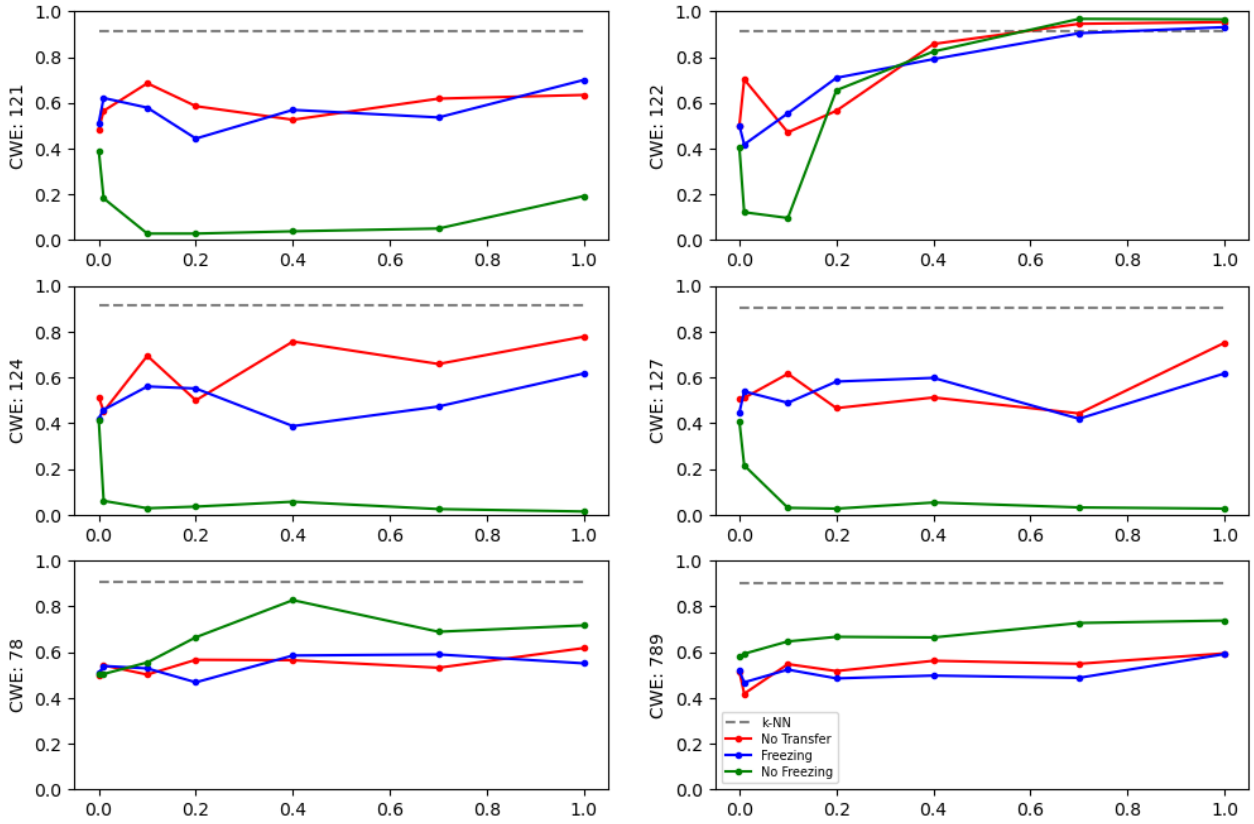

Figure 14: The transfer learning results on the CNN model with freezing and without freezing for all six CWEs. We compare the accuracy (y-axis) against the percentage of the data set presented to the model (x-axis). The k-NN accuracy represents the k-value with the best results,(in this case, 20).

4.2.4 LSTM

The LSTM models performed poorly on the EMBER features. Only the non-freezing case with CWE 789 had a positive transfer and an accuracy above random. The transfer accuracy at 1.0 of the target data set for the CWE 789 non-freezing transfer case was 0.683 compared to the no-transfer 0.629. Refer to Figure 15 for all EMBER LSTM results.

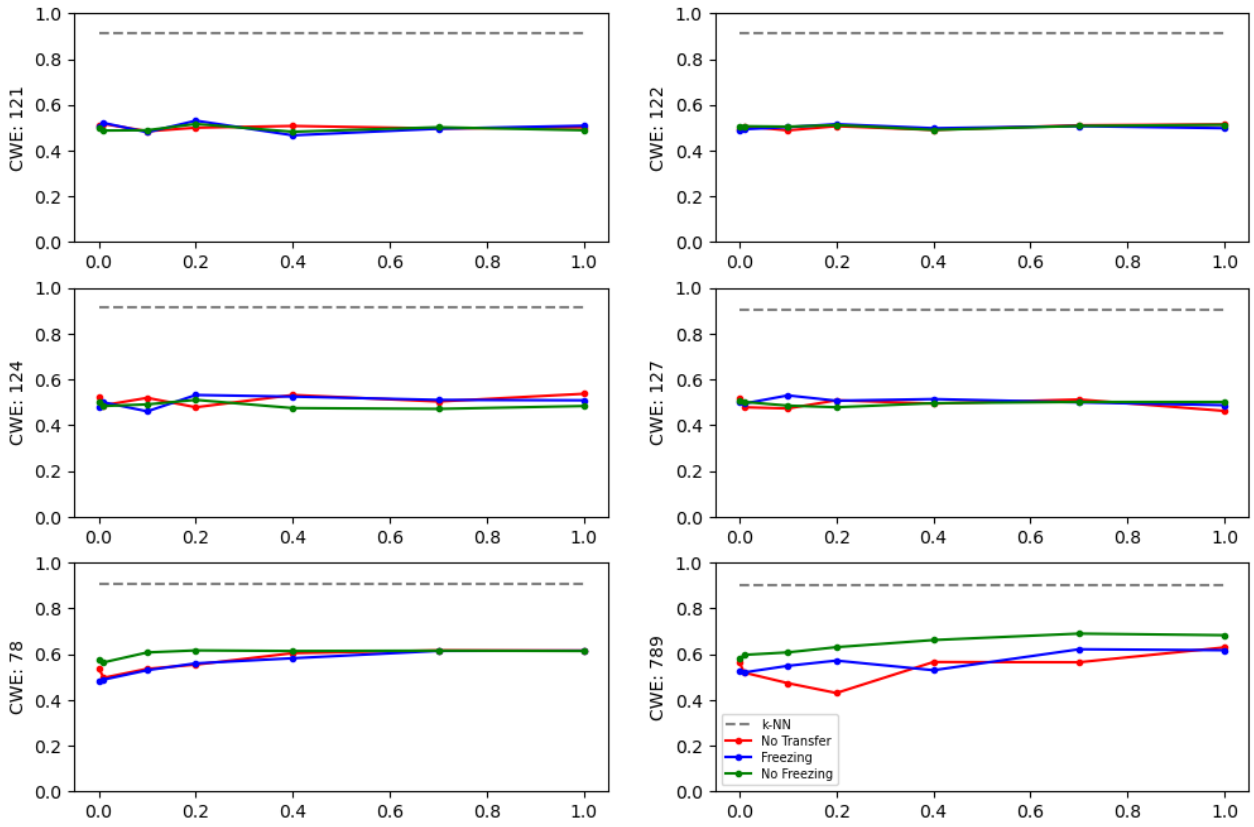

Figure 15: The transfer learning results on the LSTM model with freezing and without freezing for all six CWEs. We compare the accuracy (y-axis) against the percentage of the data set presented to the model (x-axis). The k-NN accuracy represents the k-value with the best results (in this case, 20).

4.3 Assembly Instruction Counts - Tf-Idf Embedding

4.3.1 k-NN

The k-NN model performed well on the tf-idf embedding, producing state-of-the-art detection accuracies ranging from 0.948 to 0.962 when $k = 10$. Refer to Table 7 for all k-NN results with the tf-idf embedding.

	121	**122**	**124**	**127**	**78**	**789**
$k = 1$	0.444	0.457	0.455	0.450	0.454	0.451
3	0.439	0.455	0.441	0.441	0.448	0.448
5	0.440	0.454	0.441	0.446	0.447	0.455
8	0.464	0.461	0.452	0.448	0.473	0.626
9	0.938	0.946	0.939	0.944	0.948	0.962
10	0.948	0.952	0.948	0.950	0.954	0.962
50	0.440	0.444	0.455	0.424	0.464	0.435

Table 7: The accuracy using k-NN to transfer information from the source data set to the target data set for the tf-idf data representation.

4.3.2 Feedforward

The feed-forward freezing cases performed poorly. No model had a non-random accuracy, and there were no cases of positive transfer. The no-freezing case performed well for the feed-forward model. All CWEs had a positive transfer. CWE 78 achieved a state-of-the-art transfer accuracy (0.945) and no-transfer accuracy (0.938). Refer to Figure 16 for all tf-idf feed-forward results.

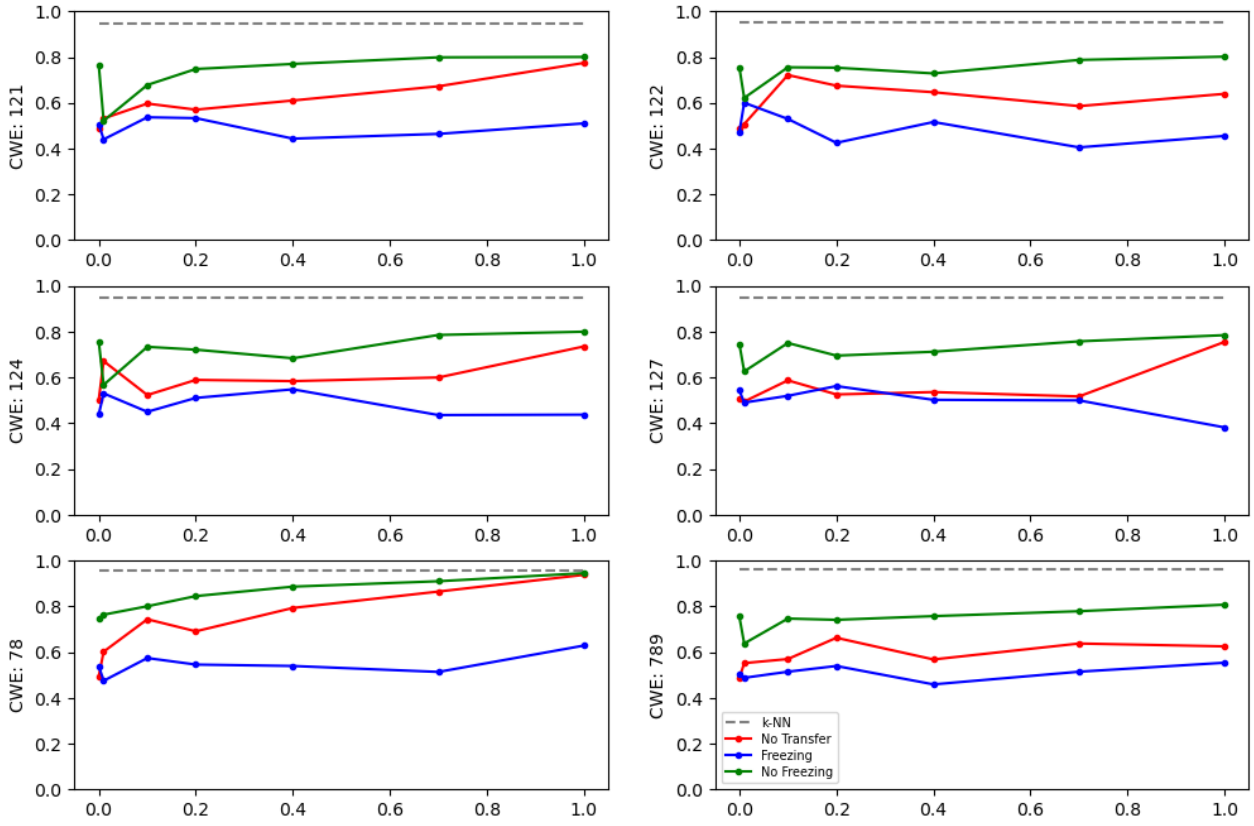

Figure 16: The transfer learning results on the feed-forward model with freezing and without freezing for all six CWEs. We compare the accuracy (y-axis) against the percentage of the data set presented to the model (x-axis). The k-NN accuracy represents the k-value with the best results (in this case, 10).

4.3.3 CNN

The CNN freezing cases performed poorly. No model had a non-random accuracy and there were no cases of positive transfer. Most non-freezing cases did not produce positive transfer, but nearly identical results with the no-transfer case. Only CWE 78 had a positive transfer with a non-freezing accuracy (0.963) compared to a no-transfer accuracy (0.898). However, some non-freezing cases did not produce high accuracies, such as CWE 124 which achieved a transfer accuracy (0.848) and a no-transfer accuracy (0.832). Refer to Figure 17 for all tf-idf CNN results.

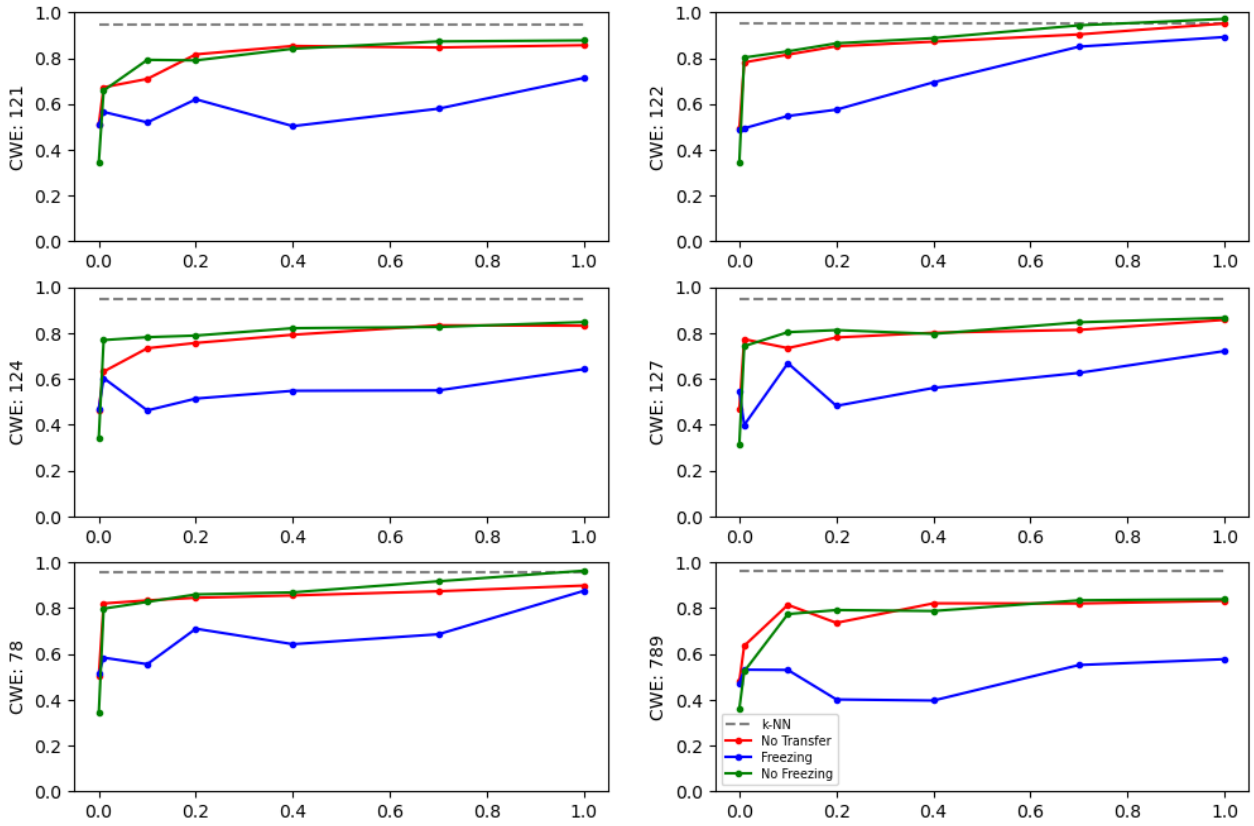

Figure 17: The transfer learning results on the CNN model with freezing and without freezing for all six CWEs. We compare the accuracy (y-axis) against the percentage of the data set presented to the model (x-axis). The k-NN accuracy represents the k-value with the best results (in this case, 10).

4.3.4 LSTM

No positive transfer was observed for the LSTM models. The freezing models performed worse than the no-transfer models, and the no-freezing models performed identically to the no-transfer models. In some cases the models reached non-random accuracies, such as CWE 122 (0.814). Refer to Figure 18 for all tf-idf LSTM results.

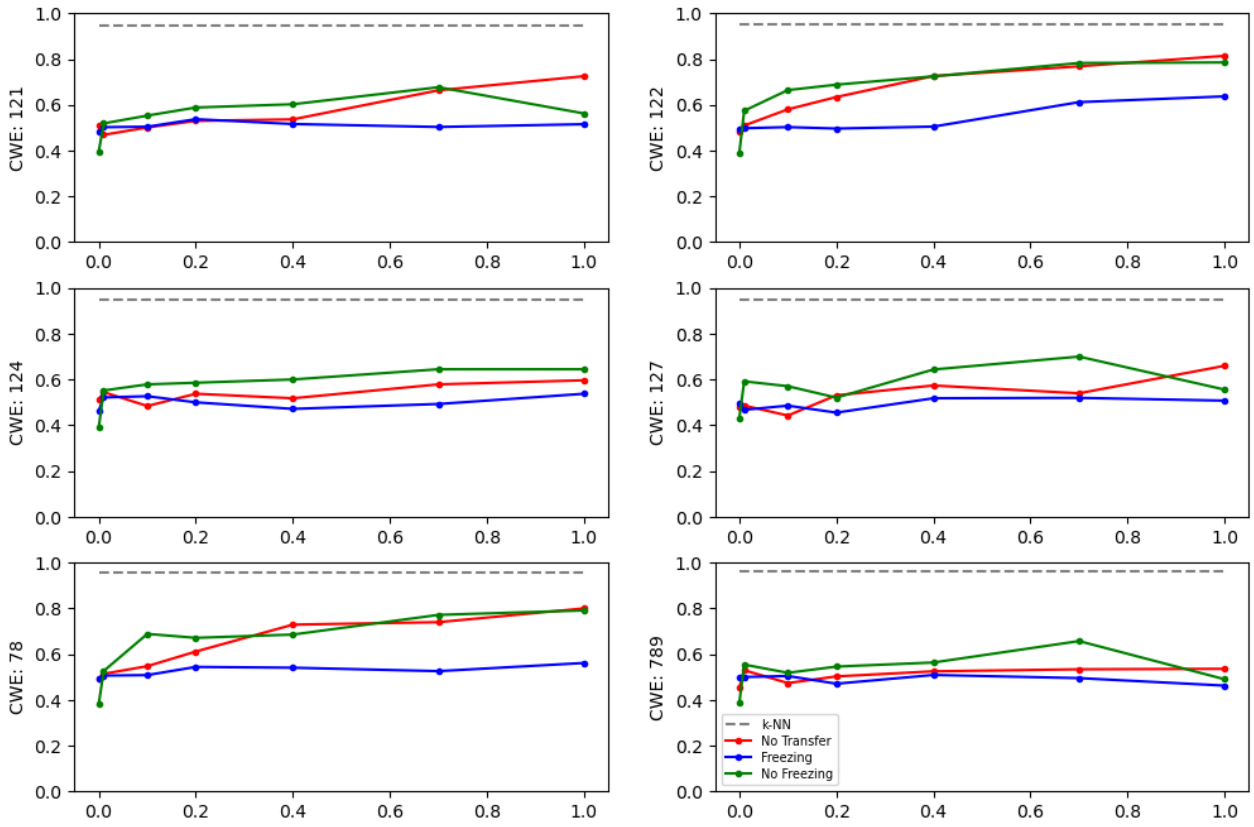

Figure 18: The transfer learning results on the LSTM model with freezing and without freezing for all six CWEs. We compare the accuracy (y-axis) against the percentage of the data set presented to the model (x-axis). The k-NN accuracy represents the k-value with the best results (in this case, 10).

4.4 Assembly Instruction Counts - Doc2Vec DBOW Embedding

We did not observe any non-random results in the doc2vec DBOW experiments. The doc2vec embeddings cluster poorly as observed in Figure 7, which explains the poor results. We present k-NN results for the doc2vec embeddings in Table 8, feed-forward results in Figure 19, CNN results in Figure 20, and LSTM results in Figure 21.

4.4.1 k-NN

	121	**122**	**124**	**127**	**78**	**789**
$k = 3$	0.508	0.509	0.466	0.480	0.498	0.520
5	0.508	0.514	0.486	0.484	0.504	0.507
10	0.499	0.505	0.466	0.460	0.508	0.496
25	0.477	0.495	0.457	0.511	0.501	0.455
50	0.511	0.512	0.518	0.495	0.492	0.491
100	0.508	0.511	0.523	0.495	0.495	0.504
200	0.490	0.487	0.514	0.505	0.488	0.498

Table 8: The accuracy using k-NN to transfer information from the source data set to the target data set for the doc2vec DBOW data representation.

4.4.2 Feedforward

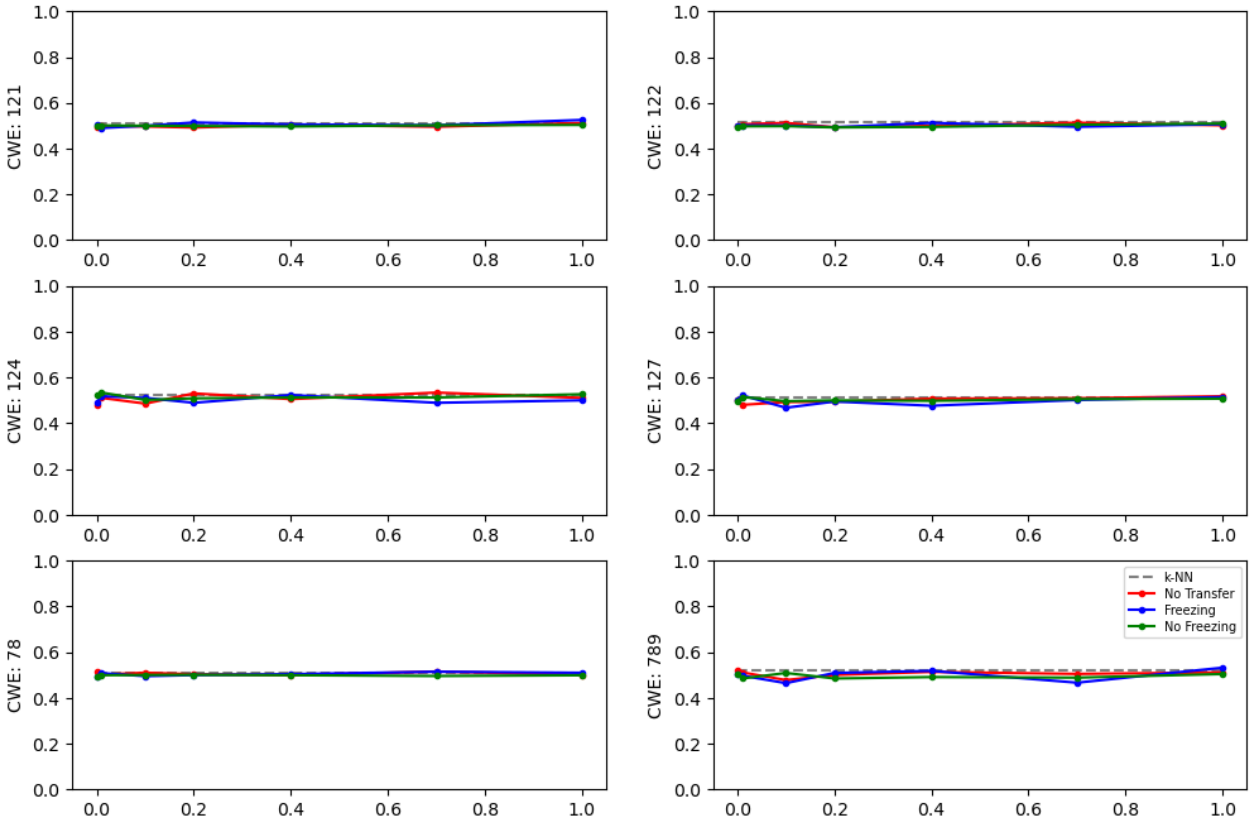

Figure 19: The transfer learning results on the feed-forward model with freezing and without freezing for all six CWEs. We compare the accuracy (y-axis) against the percentage of the data set presented to the model (x-axis). The k-NN accuracy represents the best k-value for that CWE for any k-value tested.

4.4.3 CNN

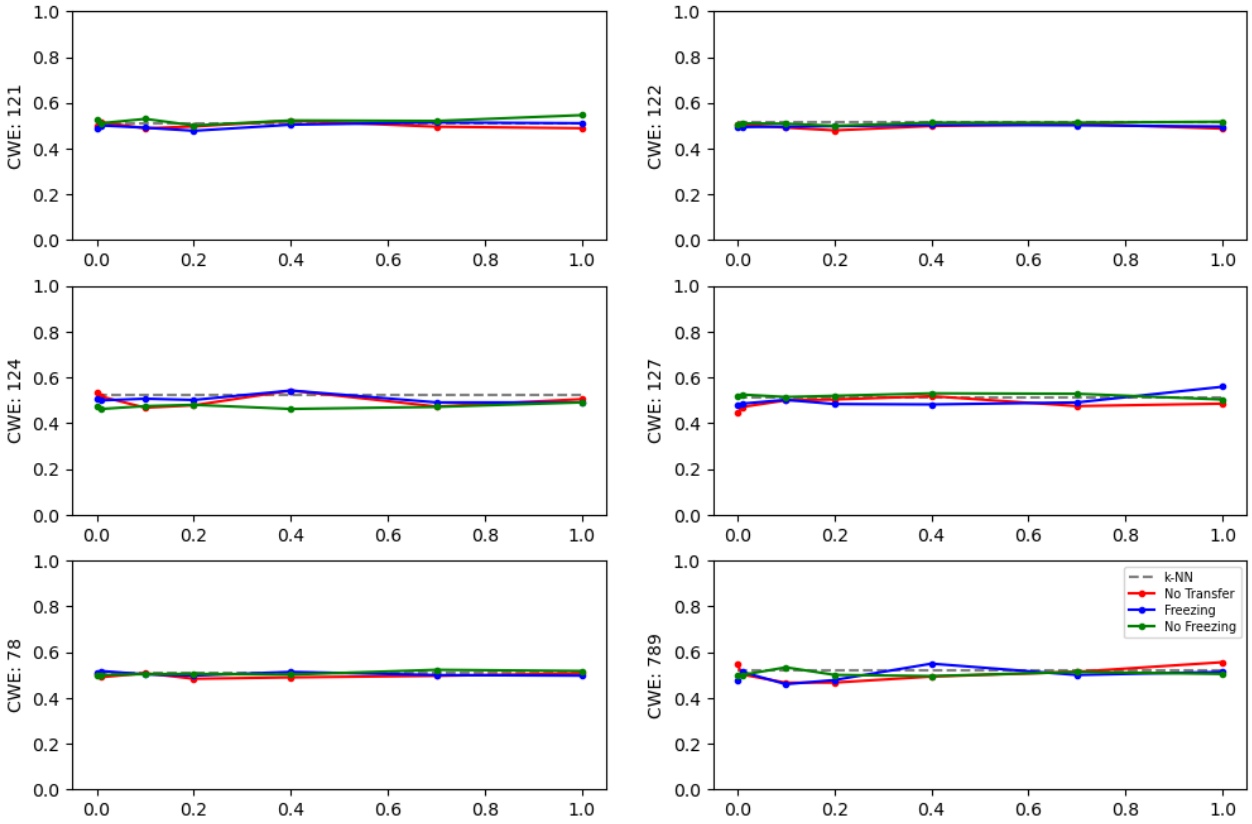

Figure 20: The transfer learning results on the CNN model with freezing and without freezing for all six CWEs. We compare the accuracy (y-axis) against the percentage of the data set presented to the model (x-axis). The k-NN accuracy represents the best k-value for that CWE for any k-value tested.

4.4.4 LSTM

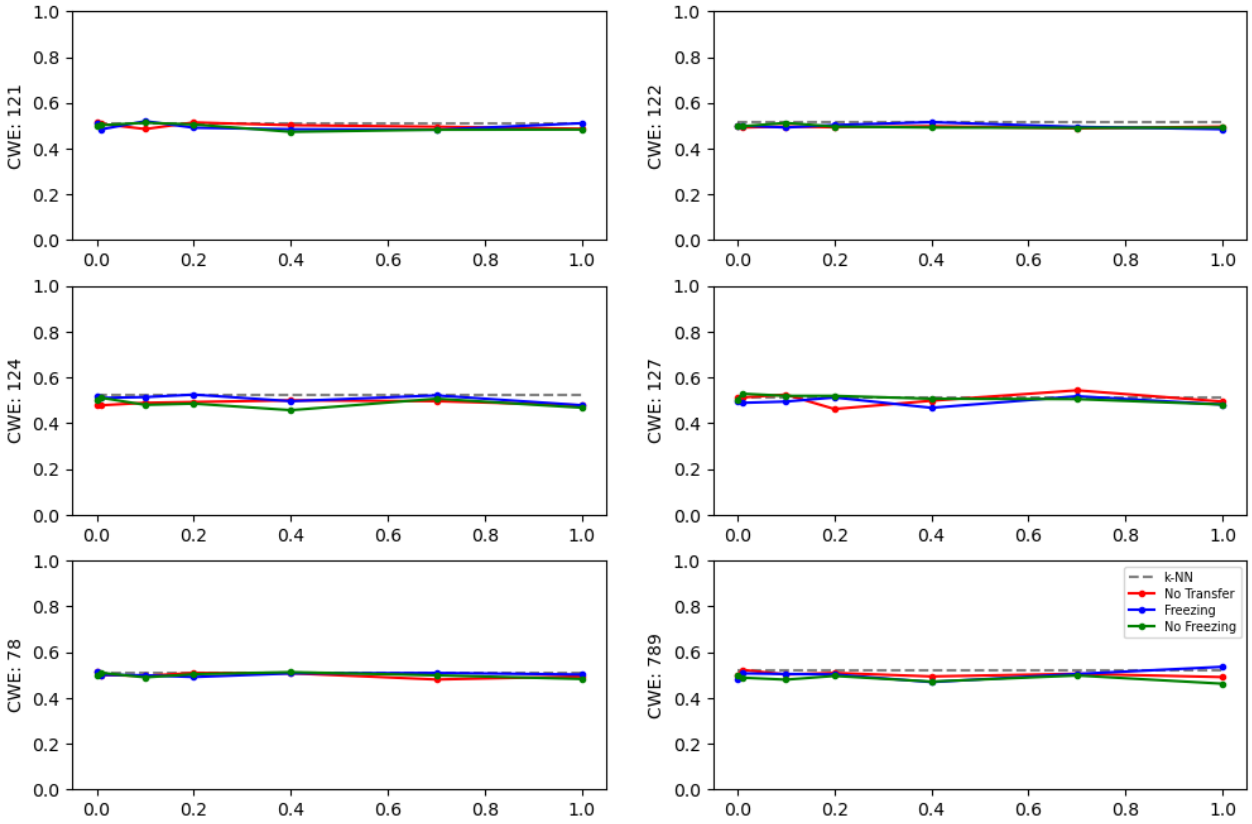

Figure 21: The transfer learning results on the LSTM model with freezing and without freezing for all six CWEs. We compare the accuracy (y-axis) against the percentage of the data set presented to the model (x-axis). The k-NN accuracy represents the best k-value for that CWE for any k-value tested.

4.5 Graph Embeddings

4.5.1 k-NN

The graph embedding representation produced successful transfer learning using a k-NN model. The most successful k-NN model had a k-value ($k = 50$), where the accuracies ranged from 0.881 to 0.963. We present all k-NN results for the graph embedding in Table 9.

	121	**122**	**124**	**127**	**78**	**789**
$k = 1$	0.482	0.741	0.549	0.990	0.584	0.625
25	0.702	0.742	0.780	0.952	0.688	0.714
40	0.868	0.915	0.963	0.952	0.860	0.929
45	0.904	0.919	0.963	0.942	0.880	0.963
50	0.895	0.918	0.963	0.942	0.881	0.938
100	0.855	0.889	0.939	0.913	0.876	0.929
200	0.820	0.866	0.890	0.875	0.851	0.884

Table 9: The accuracy of using k-NN to transfer information from the source data set to the target data set for the graph embedding data representation.

4.5.2 Feedforward

Only CWE 789 produced a feed-forward positive transfer, in both the freezing and no-freezing case. Refer to Figure 22 for all graph embedding feed-forward results.

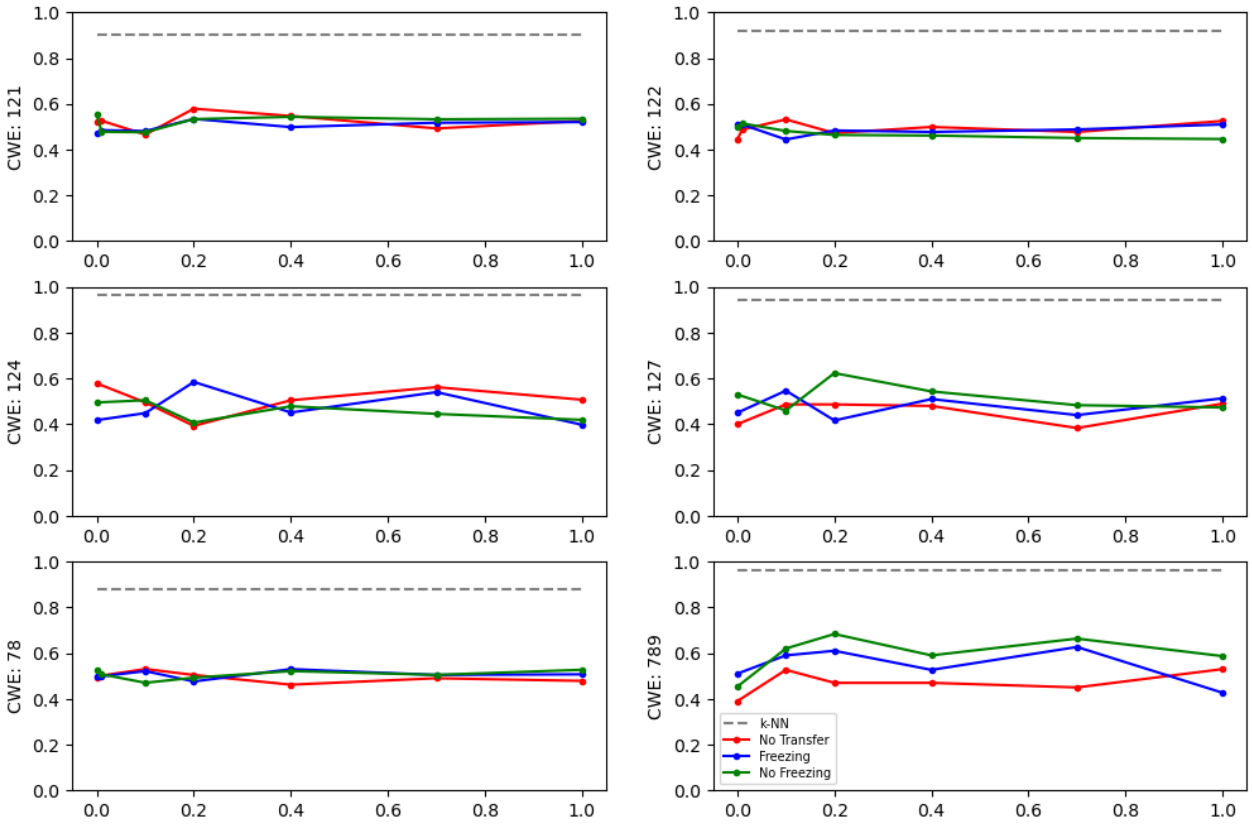

Figure 22: The transfer learning results on the feed-forward model with freezing and without freezing for all six CWEs. We compare the accuracy (y-axis) against the percentage of the data set presented to the model (x-axis). The k-NN accuracy represents the k-value with the best results (in this case, 50).

4.5.3 CNN

The CNN graph embedding experiments produced no non-random accuracies. Refer to Figure 23 for all graph embedding CNN results.

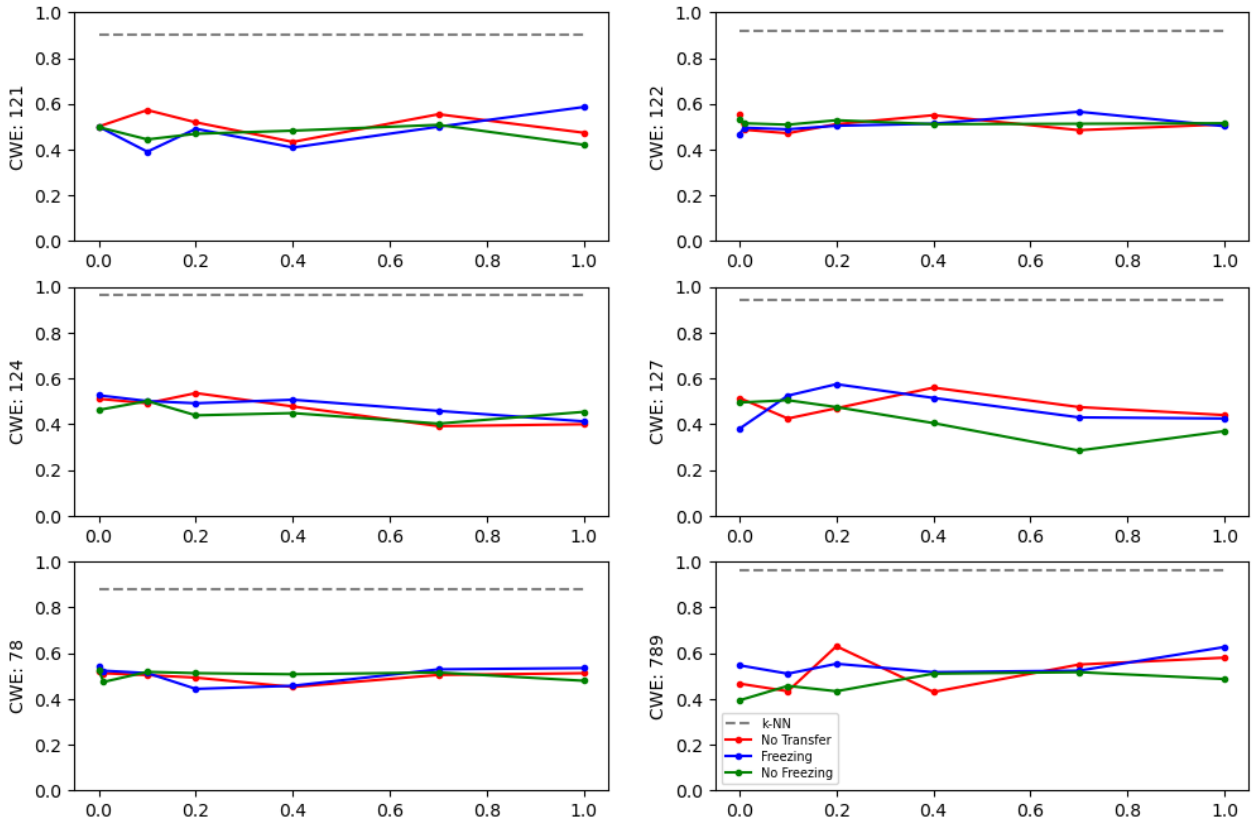

Figure 23: The transfer learning results on the CNN model with freezing and without freezing for all six CWEs. We compare the accuracy (y-axis) against the percentage of the data set presented to the model (x-axis). The k-NN accuracy represents the k-value with the best results (in this case, 50).

4.5.4 LSTM

Three LSTM experiments showed a positive transfer. The CWE 124 non-freezing case produced a transfer accuracy (0.556) compared to a no-transfer (0.418). The CWE 121 non-freezing case produced a transfer accuracy (0.565) compared to a no-transfer accuracy (0.476). The CWE 789 freezing case produced a transfer accuracy (0.520) compared to a no-transfer accuracy (0.420). Refer to Figure 24 for all graph embedding LSTM results.

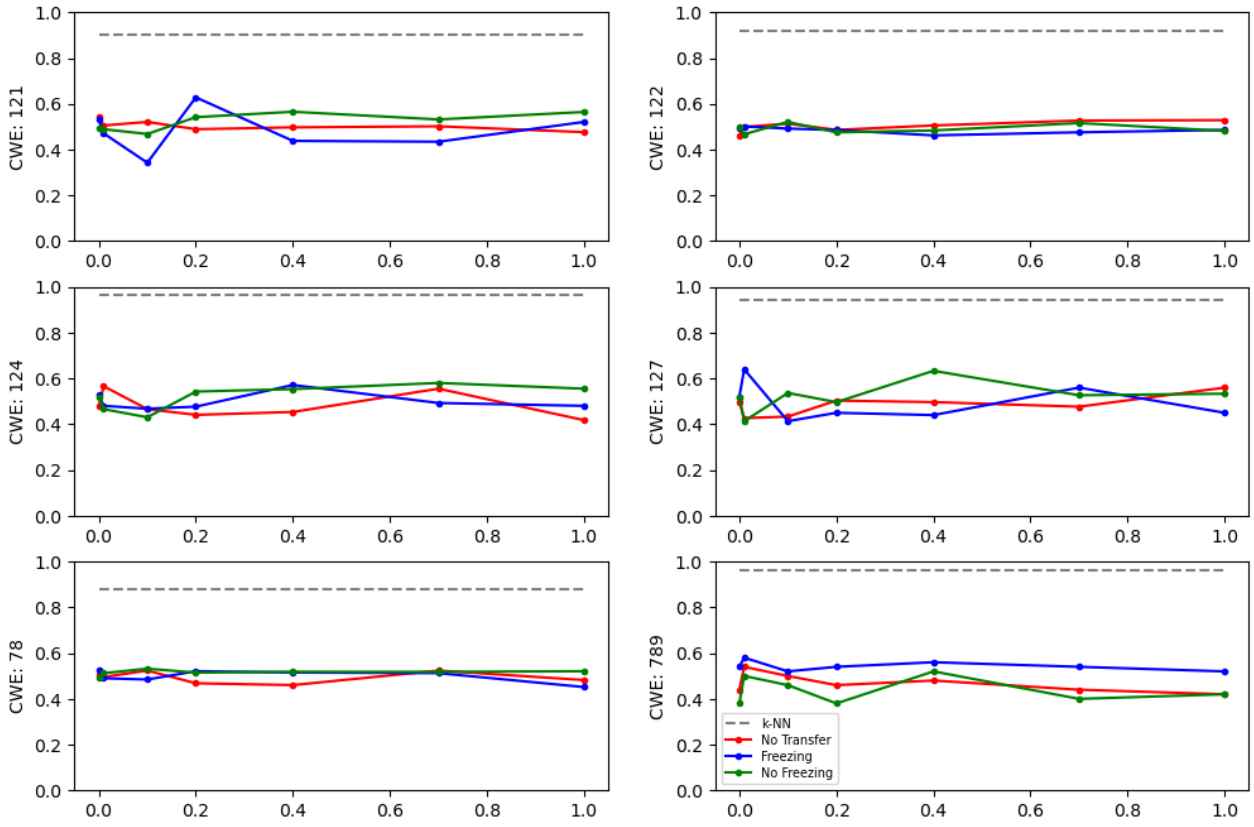

Figure 24: The transfer learning results on the LSTM model with freezing and without freezing for all six CWEs. We compare the accuracy (y-axis) against the percentage of the data set presented to the model (x-axis). The k-NN accuracy represents the k-value with the best results (in this case, 50).

5 Discussion

5.1 Results Discussion

The no-freezing models had a positive transfer. The no-freezing single-channel image and tf-idf embedding models were particularly successful. Pre-training a model on malware detection provides a benefit on a later vulnerability detection task. The freezing models performed poorly, either producing no transfer or a negative transfer. The poor performance of the freezing models suggests that fine-tuning all weights is necessary. Below in Figures 28 and 25 we only present non-freezing results.

5.1.1 Models

A k-NN model with a well-chosen k-value provides excellent results for all data representations, aside from the single-channel image and doc2vec DBOW embeddings. The k-NN results illustrate the similarity between malware and vulnerability detection tasks. However, k-NN is often inefficient. The single-channel image k-NN experiments do not exceed 0.868 for any CWE or k-value. Neural networks are beneficial in the inefficient k-NN cases.

The CNN was the most successful neural network architecture, as Figure 25 demonstrates. We provide evidence that CNNs perform well when presented with malware and vulnerability detection tasks. Our results suggests that future transfer learning research on malware and vulnerabilities should pursue convolutional architectures.

The feed-forward architecture was second best neural network architecture, and also produced positive transfer results. The feed-forward and CNN results were often nearly identical (e.g., the tf-idf embedding for CWE 78 in Figure 25).

The LSTM architecture was unsuccessful, rarely producing non-random results and never exceeding the k-NN results. Our research suggests that vulnerability data should not be treated as sequential. In contexts where vulnerability and malware data are streaming, it is more useful to consider each case independently.

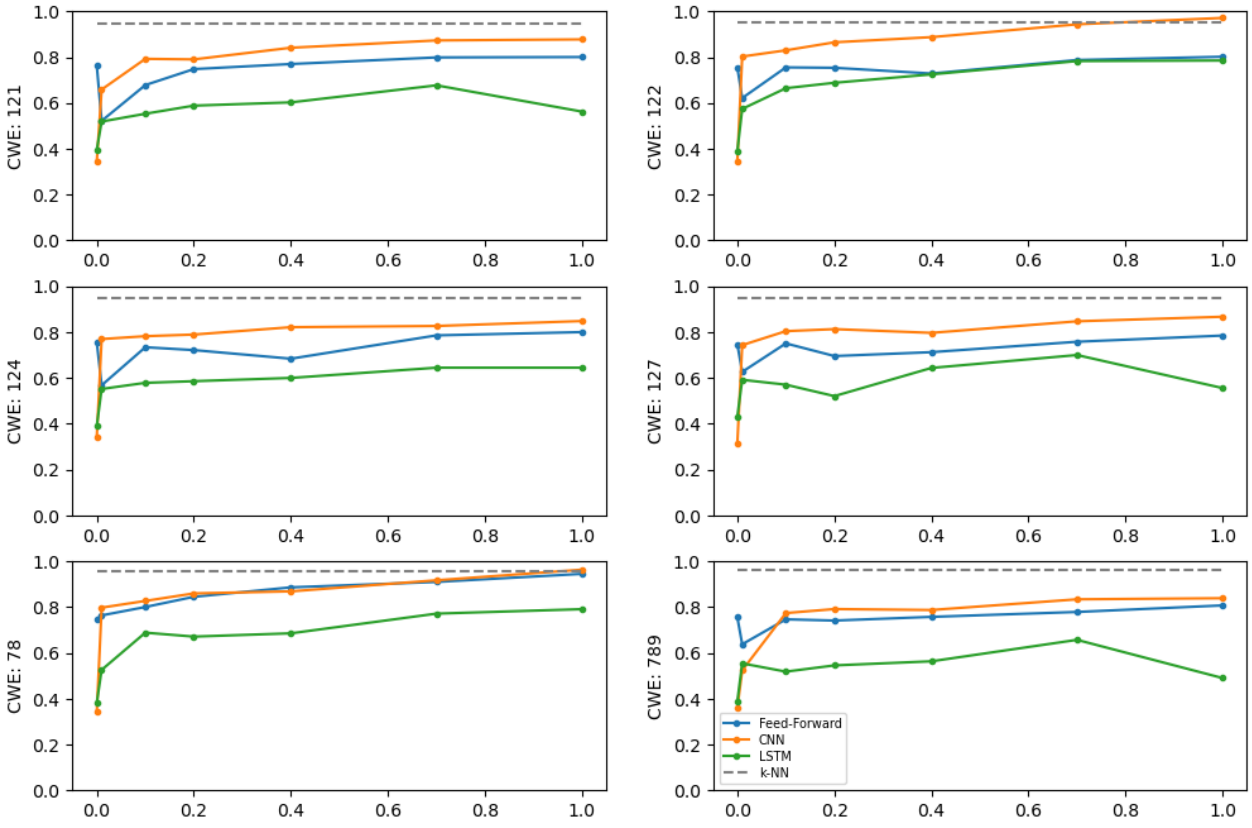

Figure 25: The non-freezing transfer tf-idf results for all architectures across all CWEs. We present tf-idf here, since tf-idf was successful among data representations, and all neural network architectures were ran on the tf-idf embedding.

5.1.2 Data Representations

The single-channel image was the most successful data representation. Our results agree with previous research that images are a good data representation for malware detection. We also provide evidence that image data representations are applicable to diverse problems in binary analysis. CNNs performed well on the single-channel images as expected, since they excel on image-based tasks.

The tf-idf embedding was the second most successful data representation. It reached slightly lower accuracies in all cases than the single-channel image data representation. The tf-idf embedding had a nearly identical accuracy to the single-channel image in some cases (e.g. CWE 78 with the feed-forward architecture in Figure 28 with a tf-idf (0.945) and image accuracy (0.963)).

The EMBER features data representation, for most CWEs, produced non-random results. The EMBER features perform well for CWE 78 and CWE 122 in the feed-forward experiments. However, they perform much worse than random for CWE 127 and CWE 124. The same occurs in the no-freezing CNN for CWEs 121, 124, and 127. The result remains worse than random even after repeating the experiment and varying hyperparameters such as the learning rate scheduler step size. All 1-D data representations were run on the same models, so this effect is likely due to the EMBER features data representation rather than faulty model setup. There is an interesting effect in the plots of these experiments in Figure 26. The accuracy is initially high or random and decays to a lower-than-random accuracy, often near zero. The effect does not occur with random weight initialization. When training a model to classify malware, there may be a specific location in parameter space that will learn to misclassify vulnerabilities exactly. Empirically, this location is reachable when training from a model with either initially high or random accuracies, as in Figure 26. Theoretically, a model should be able to learn an inverse function and reverse the learned values. However, this may not be the case if the training data of the model prefer the location in parameter space which results in a near-zero accuracy. We present some successful experiments for the EMBER feature representation, but our mixed results suggest that researchers should investigate EMBER further before using it in practice.

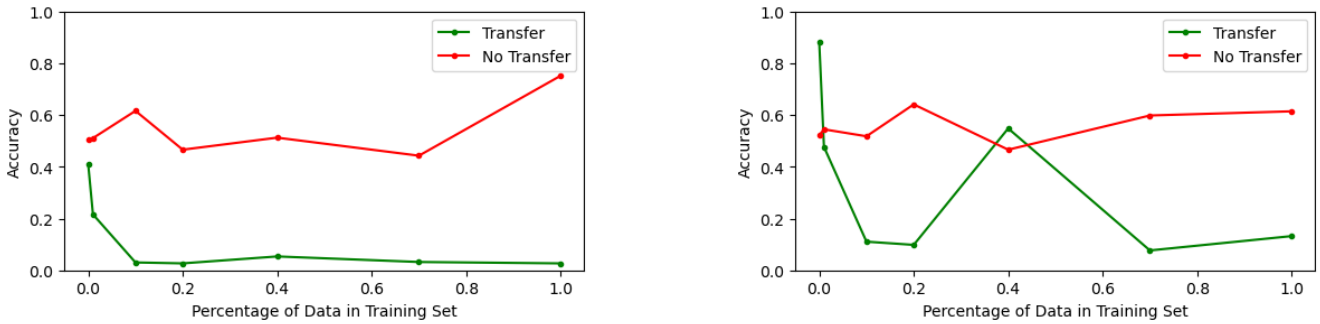

Figure 26: Examples of worse-than-random accuracies using the EMBER features data representation. Here we show CWE 127 with the non-freezing CNN (left) and CWE 124 with the non-freezing feed-forward network (right).

The doc2vec DBOW embedding was unsuccessful in performing either transfer learning or supervised learning. All model accuracies are near 0.5, which suggests a poor data representation. We present the inability of the doc2vec DBOW to cluster in the t-SNE in Figure 7. Practitioners should not use this data representation for malware detection, vulnerability detection, or transfer learning between the two tasks.

The graph embedding produced poor results, comparable to the doc2vec DBOW embedding. Unlike the doc2vec DBOW embedding, a few experiments have a positive transfer. The non-freezing feed-forward CWE 789 results which we present in Figure 28, the freezing LSTM model with CWE 789, and the non-freezing LSTM model with CWE 124 all have a positive transfer. We present the latter two LSTM results in Figure 27. Despite some promising results, the highest accuracy reached by a non-k-NN model with graph embeddings was 0.683. The k-NN results were successful, reaching an accuracy (0.963). The small data set size is likely the cause of the low neural network graph embedding accuracies.

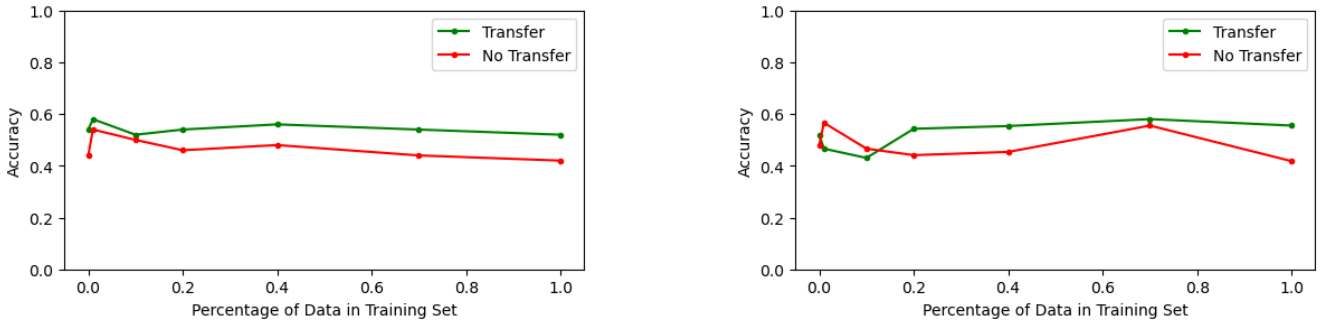

Figure 27: Examples of positive transfer using the freezing LSTM model with CWE 789 (left) and the non-freezing LSTM model with CWE 124 (right) with the graph embeddings.

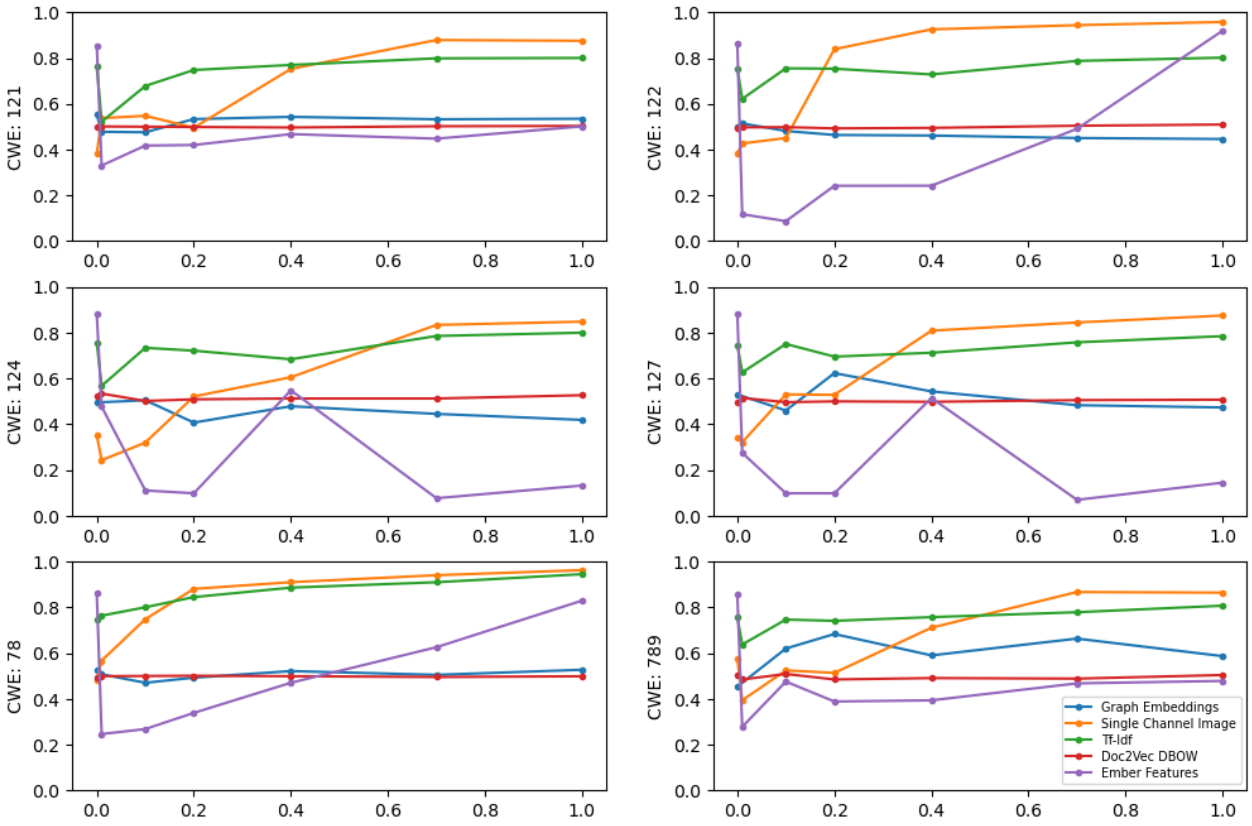

Figure 28: The non-freezing transfer feed-forward results for all data representations across all CWEs.

5.2 Future Work

5.2.1 Data Sets

The successful preliminary results in this thesis present a good opportunity for further research. Further experiments should increase the size of the source data set. Many widely known successful examples of transfer learning include millions of data points. Given that a data set of 8000 produced successful transfer learning, it is likely that a data set of hundreds of thousands or millions of data points will lead to more successful results.

The source of our target data set is SARD, a synthetic data set [11, 25]. SARD's source code is intentionally generated and does not represent actual cases of vulnerabilities found in the wild. Such data sets of actual vulnerability cases exist; however, they are difficult or impossible to compile into a binary. Given a data set of real-world vulnerable binaries, researchers can better understand the applicability of malware to vulnerability transfer models. A model could also be pre-trained on synthetic vulnerabilities to determine the feasibility of using synthetic data sets to detect real-world vulnerabilities.

The NIST SARD is also limited because there are no Common Vulnerabilities and Exposures (CVE) labels. CVE labels are only assigned to actual vulnerability cases [22]. Investigators rank vulnerabilities between 0 (least severe) and 10 (most severe) during CVE labelling. Known CVE data sets remain uncompilable. Generating a compilable CVE-labeled vulnerability data set would allow models to rank vulnerability severity.

5.2.2 Other Experiments

Day one vulnerabilities can be simulated in the target data set by setting aside a single CWE and pre-training on the rest of the CWEs. After pre-training we could then fine-tune on the CWE which had been left out. We understand how a model generalizes to new vulnerabilities that may emerge by setting aside a single CWE.

We can also investigate how well a model trained on antivirus-detected malware can classify undetected malware. Such an experiment can be performed with our source data set. Each binary records the number of antiviruses that detected it and the number of antiviruses that tested it. We have done preliminary research on this classification task, with promising results.

5.2.3 Other Features

There are other data representations we did not experiment with. Previous researchers have created 3-channel images and then performed further feature extraction. Models have successfully classified malware represented as a 3-channel image [34]. Opcode counts are also a commonly used metric for feature development [12, 2].

Malicious source code is another potential feature set. Most research on vulnerability detection involves uncompiled source code, which is the data representation of most vulnerability data sets [11, 24, 25, 22]. Source code is exceptionally well suited to language models since natural and programming languages are similar. Language models are successful when applied to vulnerability data sets [10, 11, 23, 24, 25, 22].

We encountered several issues with CFG generation. We used the CFGFast object within angr to generate the CFGs, which is less accurate than the CFGEmulated object. However, the CFGEmulated object is slower and therefore untenable for data set generation. Angr is

slow because it is written in Python. A decompiler written in a language such as C or C++, potentially with a Python binding, could generate control flow graphs much quicker.

5.2.4 Other Models

Future experiments can use graph neural networks (GNNs) with CFGs. We only used CFGs with the graph embedding graph2vec algorithm and counting assembly instructions. Graph2vec employs similar techniques as GNNs (e.g. using node neighborhoods). The goal of graph2vec is not to perform machine learning tasks but rather to create embedding vectors for further processing. A graph-dedicated model would likely perform better than graph2vec. GNNs also consider node information, which in this case includes the instructions for each code block. Therefore, a GNN may capture the information from graph2vec and the assembly instruction counts.

Future experiments can pre-train well established models, such as ResNet50 and Inceptionv3, on a large data set of malware for later transfer to vulnerabilities. Researchers have used other pre-trained networks (e.g. VGG and BERT) successfully in malware detection.

5.2.5 Ensembles and Transfer Methods

Our research shows that different models perform well on different data representations. Therefore, if we generated multiple data representations of the same data set, a model ensemble could be used. Each data representation would be used with an optimal model. The optimal models would vary by architecture and hyperparameter (e.g. initial learning rate, activation functions, which layers are frozen).

There are several other methods of transfer learning we did not explore. We could use a feature extractor pre-trained on the source data set with the target data set, augment the target data set with the source data set to create a combined feature mapping, or maintain portions of a pre-trained model while randomly resetting the rest of the weights, such as the predictor.

6 Conclusion

Overall our experiments showed a potential for pre-training a model on a malware detection task to use on a later vulnerability detection task. The model weights from the malware detection task were often successfully used as a starting point for a model on a vulnerability detection task. Using different representations of malware binaries was also successful, with large accuracy improvements seen with the single-channel image data representation and tf-idf embedding. The doc2vec DBOW embedding failed to produce any learning. The EMBER features data set did produce successful learning. However, several cases had a less than random, even near 0, accuracy. Before use in detecting malware by practitioners the EMBER features data representation needs further research. The graph embedding data set also showed promise. However, it reached low accuracies, likely due to the small size of the data set itself.

The CNN architecture succeeded, often exceeding the k-NN model, and producing high accuracy for most data representations. The feed-forward network also had high accuracies in many cases. The LSTM model was only promising with the graph embedding data representation. However, the LSTM graph embedding accuracies were still relatively low compared to

k-NN. Ultimately, no model utilizing freezing was successful. Larger networks and data sets are likely required for freezing models.

References

[1] Z. Chen, J. Niu, and Q. Tian, "Dropfilter: Dropout for convolutions," *CoRR*, vol. abs/1810.09849, 2018.

[2] Y. Ye, T. Li, D. Adjeroh, and S. S. Iyengar, "A survey on malware detection using data mining techniques," *ACM Comput. Surv.*, vol. 50, 2017.

[3] A. Bensaoud, N. Abudawaood, and J. Kalita, "Classifying malware images with convolutional neural network models," *CoRR*, vol. 22, 2020.

[4] K. Y. Han, B. Kang, and E. G. Im, "Malware classification using instruction frequencies," in *Proceedings of the 2011 ACM Symposium on Research in Applied Computation*, 2011, pp. 298–300.

[5] M. Hassen, M. M. Carvalho, and P. K. Chan, "Malware classification using static analysis based features," in *2017 IEEE Symposium Series on Computational Intelligence (SSCI)*, 2017, pp. 1–7.

[6] K. Shaukat, S. Luo, V. Varadharajan, I. A. Hameed, and M. Xu, "A survey on machine learning techniques for cyber security in the last decade," *IEEE Access*, vol. 8, pp. 222 310–222 354, 2020.

[7] N. Bhodia, P. Prajapati, F. D. Troia, and M. Stamp, "Transfer learning for image-based malware classification," *CoRR*, vol. abs/1903.11551, 2019.

[8] J. Chen, "A malware detection method based on rgb image," in *Proceedings of the 2020 6th International Conference on Computing and Artificial Intelligence*, 2020, p. 283–290.

[9] H. Benkraouda, J. Qian, H. Q. Tran, and B. Kaplan, "Attacks on visualization-based malware detection: Balancing effectiveness and executability," *CoRR*, vol. abs/2109.10417, 2021.

[10] Z. Chen, S. Kommrusch, and M. Monperrus, "Neural transfer learning for repairing security vulnerabilities in c code," *IEEE*, vol. 49, 2022.

[11] S. Chakranorty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet?" *IEEE*, vol. 48, 2021.

[12] H. Rathore, S. Agarwal, S. K. Sahay, and M. M. Sewak, "Malware detection using machine learning and deep learning," in *Big Data Analytics*. Springer International Publishing, 2018.

[13] K. Weiss, T. M. Khoshgoftaar, and D. Wang, "A survey of transfer learning," *Journal of Big Data*, vol. 3, 2016.

[14] F. Zhuang, Z. Qi, K. Duan, D. Xi, Y. Zhu, H. Zhu, H. Xiong, and Q. He, "A comprehensive survey on transfer learning," *CoRR*, vol. abs/1911.02685, 2019.

[15] E. S. Olivas, J. D. M. Guerrero, M. M. Sober, J. R. M. J. R. M. Benedito, and A. J. S. Lopez, *Handbook Of Research On Machine Learning Applications and Trends: Algorithms, Methods and Techniques*. Hershey, PA: Information Science Reference - Imprint of: IGI Publishing, 2009.

[16] H. S. Anderson and P. Roth, "EMBER: an open dataset for training static PE malware machine learning models," *CoRR*, vol. abs/1804.04637, 2018.

[17] F. Ö. Çatak and A. F. Yazi, "A benchmark API call dataset for windows PE malware classification," *CoRR*, vol. abs/1905.01999, 2019.

[18] L. Nataraj, S. Karthikeyan, G. Jacob, and B. S. Manjunath, "Malware images: Visualization and automatic classification," *CoRR*, vol. abs/1301.3781, 2013.

[19] V. Anandhi, P. Vinod, and V. G. Menon, "Malware visualization and detection using densenets," *Personal and Ubiquitous Computing*, 2021.

[20] N. Marastoni, R. Giacobazzi, and M. D. Preda, "Data augmentation and transfer learning to classify malware images in a deep learning context," *Journal of Computer Virology and Hacking Techniques*, vol. 17, 2021.

[21] P. Panda, O. K. C. U, S. Marappan, S. Ma, M. S, and D. V. Nandi, "Transfer learning for image-based malware detection for iot," *Sensors*, vol. 23, 2023.

[22] G. P. Bhandari, A. Naseer, and L. Moonen, "Cvefixes: Automated collection of vulnerabilities and their fixes from open-source software," *CoRR*, vol. abs/2107.08760, 2021.

[23] T. Marjanov, I. Pashchenko, and F. Massacci, "Machine learning for source code vulnerability detection: What works and what isn't there yet," *IEEE Security Privacy*, vol. 20, 2022.

[24] D. Grahn and J. Zhang, "An analysis of c/c++ datasets for machine learning-assisted software vulnerability detection."

[25] Y. Chen, Z. Ding, X. Chen, and D. Wagner, "Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection," 2023.

[26] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 709–724.

[27] A. Schaad and D. Binder, "Deep-learning-based vulnerability detection in binary executables," in *Foundations and Practice of Security*, 2023, pp. 453–460.

[28] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovre: Efficient cross-architecture identification of bugs in binary code," in *Network and Distributed System Security Symposium*, 2016.

[29] "Pe malware machine learning dataset," Practical Security Analytics, 2018. [Online]. Available: https://practicalsecurityanalytics.com/pe-malware-machine-learning-dataset/

[30] R. Ronen, M. Radu, C. Feuerstein, E. Yom-Tov, and M. Ahmadi, "Microsoft malware classification challenge," *CoRR*, vol. abs/1802.10135, 2018.

[31] "Nist software assurance reference dataset," National Institute of Standards and Technology, 2018. [Online]. Available: https://samate.nist.gov/SARD/

[32] "Common weakness enumeration," The Mitre Corporation, 2023. [Online]. Available: https://cwe.mitre.org/

[33] L. Nataraj, V. Yegneswaran, P. Porras, and J. Zhang, "A comparative assessment of malware classification using binary texture analysis and dynamic analysis," in *Proceedings of the 4th ACM Workshop on Security and Artificial Intelligence*, ser. AISec '11, 2011, p. 21–30.

[34] J. Fu, J. Xue, Y. Wang, Z. Liu, and C. Shan, "Malware visualization for fine-grained classification," *IEEE Access*, vol. 6, pp. 14 510–14 523, 2018.

[35] K. Q. Weinberger, A. Dasgupta, J. Attenberg, J. Langford, and A. J. Smola, "Feature hashing for large scale multitask learning," *CoRR*, vol. abs/0902.2206, 2009.

[36] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *IEEE Symposium on Security and Privacy*, 2016.

[37] E. Cheng, "Binary analysis and symbolic execution with angr selection for anomaly detection," Ph.D. dissertation, Worcester Polytechnic Institute, 2016. [Online]. Available: https://web.wpi.edu/Pubs/E-project/Available/E-project-101816-114710/unrestricted/echeng_mqp_angr.pdf

[38] Q. Lee and T. Mikolov, "Distributed representations of sentences and documents," *CoRR*, vol. abs/1405.4053, 2014.

[39] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *CoRR*, vol. abs/1301.3781, 2013.

[40] A. Narayanan, M. Chandramohan, R. Venkatesan, L. Chen, Y. Liu, and S. Jaiswal, "graph2vec: Learning distributed representations of graphs," *CoRR*, vol. abs/1707.05005, 2017.

[41] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, 1998.

[42] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, pp. 1735–1780, 1997.

[43] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *CoRR*, vol. abs/1502.03167, 2015.

[44] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *The Journal of Machine Learning Research*, vol. 15, 2014.

[45] C. Laurent, G. Pereyra, P. Brakel, Y. Zhang, and Y. Bengio, "Batch normalized recurrent neural networks," in *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2016, pp. 2657–2661.

[46] "Welcome to pytorch lightning," Lightning AI, 2023. [Online]. Available: https://lightning.ai/docs/pytorch/latest/