Recommending TEE-based Functions Using a Deep Learning Model

Steven Lim

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science and Applications

Eli Tilevich, Chair
Na Meng
Francisco Servant

August 23, 2021
Blacksburg, Virginia

Keywords: Trusted Execution Environment, Recommendation System, Machine Learning,
Function Classification, Abstract Syntax Tree Classification
Copyright 2021, Steven Lim

Recommending TEE-based Functions Using a Deep Learning Model

Steven Lim

(ABSTRACT)

Trusted execution environments (TEEs) are an emerging technology that provides a protected hardware environment for processing and storing sensitive information. By using TEEs, developers can bolster the security of software systems. However, incorporating TEE into existing software systems can be a costly and labor-intensive endeavor. Software maintenance—changing software after its initial release—is known to contribute the majority of the cost in the software development lifecycle. The first step of making use of a TEE requires that developers accurately identify which pieces of code would benefit from being protected in a TEE. For large code bases, this identification process can be quite tedious and time-consuming. To help reduce the software maintenance costs associated with introducing a TEE into existing software, this thesis introduces ML-TEE, a recommendation tool that uses a deep learning model to classify whether an input function handles sensitive information or sensitive code. By applying ML-TEE, developers can reduce the burden of manual code inspection and analysis. ML-TEE's model was trained and tested on functions from GitHub repositories that use Intel SGX and on an imbalanced dataset. The accuracy of the final model used in the recommendation system has an accuracy of 98.86% and an F1 score of 80.00%. In addition, we conducted a pilot study, in which participants were asked to identify functions that needed to be placed inside a TEE in a third-party project. The study found that on average, participants who had access to the recommendation system's output had a 4% higher accuracy and completed the task 21% faster.

Recommending TEE-based Functions Using a Deep Learning Model

Steven Lim

(GENERAL AUDIENCE ABSTRACT)

Improving the security of software systems has become critically important. A trusted execution environment (TEE) is an emerging technology that can help secure software that uses or stores confidential information. To make use of this technology, developers need to identify which pieces of code handle confidential information and should thus be placed in a TEE. However, this process is costly and laborious because it requires the developers to understand the code well enough to make the appropriate changes in order to incorporate a TEE. This process can become challenging for large software that contains millions of lines of code. To help reduce the cost incurred in the process of identifying which pieces of code should be placed within a TEE, this thesis presents ML-TEE, a recommendation system that uses a deep learning model to help reduce the number of lines of code a developer needs to inspect. Our results show that the recommendation system achieves high accuracy as well as a good balance between precision and recall. In addition, we conducted a pilot study and found that participants from the intervention group who used the output from the recommendation system managed to achieve a higher average accuracy and perform the assigned task faster than the participants in the control group.

Acknowledgments

I would like to thank my advisor Dr. Tilevich for his guidance and support through my research. Your expertise and insightful feedback helped shape my research and steer it in the right direction.

I would also like to thank Dr. Liu for his guidance throughout the research process. You helped me formulate my research direction and your insightful feedback was invaluable to the completion of this thesis.

Contents

List of Figures

List of Tables

List of Abbreviations

AST Abstract Syntax Tree

CNN Convolution Neural Network

EDL Enclave Definition Language

GG-NN Gated graph neural network

GG-NN+CNN Gated graph neural network layer followed by convolution neural network layer.

GG-NN+GP Gated graph neural network layer followed by global pooling layer.

GG-NN+LSTM Gated graph neural network layer followed by long short term memory layer.

GNN Graph Neural Network

GRU Gated Recurrent Unit

LSTM Long Short Term Memory

NLP Natural Language Processing

SGX Software Guard Extension

TCB Trusted Computing Base

TEE Trusted Execution Environment

Chapter 1

Introduction

The security of a software system is a major concern for both the provider and the end user, especially if the software stores or processes sensitive information, such as credit card information. A potential solution to help better protect this data from being leaked during a cyber attack is a trusted execution environment (TEE). A TEE provides an isolated region of memory on which code can execute and data accessed in this region of memory is protected in terms of confidentiality and integrity [44]. However, to incorporate a TEE into existing software, it would require the developers of the software to first identify which pieces of code to place within the TEE. This process is laborious and error-prone, resulting in code not needing TEE protection being placed within a TEE, which increases the trusted computing base (TCB) of the software and thus, causes security issues. In addition, the additional unnecessary code within the TEE can also further reduce the overall performance of the software.

The process of incorporating TEE into an existing software is done in the software maintenance part of the software life cycle, which contributes to at least 60% of the total software's life cost [23][26][46]. Software maintenance is any change made to the software after its initial release. The cost of maintenance will also increase as the size of complexity of the software increases [46], which in turn makes it a costly endeavor to incorporate new, emerging technologies, such as a TEE, into existing software with millions of lines of code. In addition, the security of an application can often be neglected or underfunded due to the fact it is an

investment, a cost, with no financial return, in other words, it does not generate revenues for the company [29]. This would mean that companies might be reluctant to invest time into refactoring existing code bases to make use of TEEs, even with the existing automated refactoring tools because of the manual labor involved with identifying what functions to refactor. Thus, with these factors,it is difficult to incorporate TEEs into existing software and applications to help improve the security of their software products.

Much of the work has been focused on automating the refactoring process to make use of a TEE, such as TZSlicer [54], which requires as input a list of variables that are the source of sensitive information. Another example is RT-Trust [37], which requires the user to identify and annotate sensitive functions. There has been very little work done specifically regarding automated tools to assist developers in finding input for these refactoring tools or finding pieces of code to place within a TEE in general. The authors of TEE-DRUP introduces a recommendation system that returns a list of variables with their sensitivity level using NLP techniques, which is then used to identify non-sensitive functions. In contrast to TEE-DRUP's recommendation's focus on identifying non-sensitive function, the focus of this thesis is on identifying sensitive functions. While their approach can potentially be used to identify sensitive functions, their approach relied heavily on the variable names being correctly spelled, and does not take into consideration other information within a function, such as the parameter names and the names of the functions invoked within this function [36].

To help reduce the cost of incorporating a TEE into existing software, this thesis presents ML-TEE, a recommendation tool that assists developers in the process of identifying which functions should be placed within a TEE. ML-TEE uses a deep learning model to output the probability for a given function to belong inside a TEE as well as a recommendation based on a heuristically determined threshold value for the probability. Developers can use the recommended functions from ML-TEE to reduce the total number of functions they

would need to inspect. ML-TEE first extracts the AST of a C/C++ function or method and preprocesses it before passing it to a deep learning model that uses a graph neural network to learn features from the AST. The model then produces a probability for the function belonging inside a TEE as well as a recommendation. Like the approach taken in the recommendation system used by the authors of TEE-DRUP, we also make use of the function name and variable name [36]; however, we also take into consideration the structure of the function and the other variables within the function as well as the names of the invoked functions.

The contributions of this thesis are as follows:

1. A novel approach to determining whether a given C/C++ function or method belongs inside a TEE, which is in the form of a recommendation system, which uses a deep learning model.

2. An evaluation of the recommendation system's impact on a developer's accuracy of identifying sensitive functions and time taken to complete this task.

3. An analysis of the features important to the deep learning model, which helps provide an explanation for why and how our model works.

We will first introduce some background information in Chapter 2 followed by a literature review in Chapter 3. In Chapter 4, we discuss the design of the recommendation system as well as the different machine learning models we considered. In Chapter 5, we introduce the datasets used to train and evaluate the models, and in Chapter 6, we present and discuss our results. In Chapter 7, we discuss both the internal and external threats to validity. Finally, we conclude our work and discuss directions for future work in Chapter 8.

Chapter 2

Background Information

This chapter introduces some of the technical background needed to understand the research presented in this thesis.

2.1 TEE

A TEE is an isolated environment for executing code with its own secure memory and storage that guarantees the confidentiality of the code, data, and runtime states stored, and the integrity of the code executed in the TEE [49]. The provided isolation allows a TEE to ignore threats present in its external environment. The code inside a TEE is called trusted code, and the code executing outside the TEE is called untrusted code. We want to only isolate sensitive code and data inside the TEE to help minimize an application's overall execution time as well as its trusted computing base.

2.2 Intel's Software Guard Extensions (SGX)

Intel SGX is Intel's implementation of a TEE and is the TEE that we use in this thesis. It provides a set of hardcoded security instructions that allow developers to encrypt sections of memory, which are called enclaves, that can be used to satisfy the confidentiality and integrity guarantees of a TEE [15]. To call a function inside the enclave from the untrusted

code, or to call a function outside the enclave from the trusted code, the function's prototype must be placed in either the trusted or untrusted section of an EDL file respectively [39].

```
enclave {
  trusted {
      public void ve_lock ();
        public int ve_unlock ([in, string] char *password);
  };

  untrusted {
  };
};
```

Figure 2.1: Example EDL file [40]

Figure 2.1 shows an example of the content in an EDL file. The file contains an enclave with a body that contains a trusted and untrusted section. Within the trusted section of the EDL file, the developer lists function prototypes for the functions that are placed inside the TEE and will be invoked by a function running outside the TEE. The function prototypes placed inside the untrusted section of the EDL file are functions placed outside the TEE but will be invoked by a function running inside the TEE. For this research, our focus is on what functions should be placed in the trusted section of the EDL file, and in the context of this work, functions that should be placed within a TEE are those that should have their function prototype placed in the trusted section of the EDL file.

A function that is placed within the enclave cannot be directly called, but instead, a proxy function is called that then executes the desired function within the enclave [50]. Thus, a proxy function is a wrapper function for the real function that we want to call [50]. Each function prototype in an EDL file will have 2 proxy functions: a trusted proxy function and an untrusted proxy function [50]. The proxy functions are generated by using the Edger8r tool that is part of the SGX SDK, and this tool will generate 2 pairs of *.c* and *.h* files with one pair of files containing the trusted proxy functions and the other pair containing the

untrusted proxy functions [50]. Typically, the former file names end in $\_t$ and the latter file names end in $\_u$.

2.3 Sensitive and Non-Sensitive Code and Data

In the context of code, we define sensitive data to be variables that store security related objects, such as passwords and keys, and we define sensitive code to be functions and methods and handle sensitive data or perform security related operations, such as encryption and decryption. We used the use cases for Intel SGX given in an online Intel article [3] and in a HackerNoon article [32] to help define security related operations.

2.4 Abstract Syntax Tree(AST)

An AST is an ordered tree representation of source code. We will define a node in an AST to contain both the node type and the node value where the node value is a token in the source code and the node type is a compiler defined value that identifies the node value.

2.5 Supervised Machine Learning Training Process

Supervised machine learning is where all the data used for training and testing is labeled. A machine learning model is composed of layers where a layer is a function where the input are weighted inputs [16].

Figure 2.2 shows an example of a simple fully connected neural network machine learning model where $i1$ and $i2$ are the input neurons, or inputs to the model, $h1$ and $h2$ are the

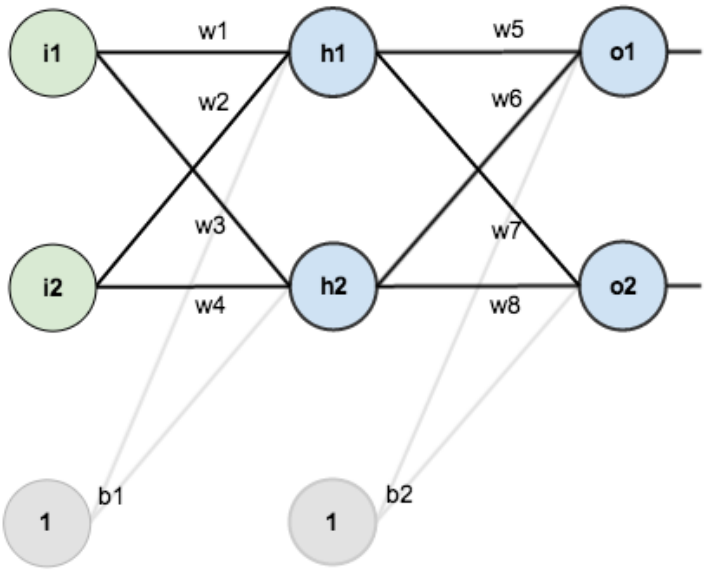


Figure 2.2: Example of a fully connected neural network [38]

hidden neurons, and $o1$ and $o2$ are the output neurons. The 2 input neurons compose the input layer, the 2 hidden neurals compose a hidden layer, and the 2 output neurons compose the output layer, where neurons in the same layer usually have the same function. Each neuron takes as input a single weighted value, except the input neuron which is an input value, and the weights in Figure 2.2 are prefixed by w followed by a number. The gray neurons with a constant value of 1 are the bias terms with $b1$ and $b2$ being the bias terms. As shown in Figure 2.2, each neuron is connected to another neuron with an edge that contains a weight, and the output from a neuron is multiplied by this weight before being passed to the connected neuron.

In general, there are 2 main steps during each iteration of the training process: a forward pass and a backward pass, which is sometimes referred to as backpropagation. During the forward pass, the input is given to the model, which then produces an output. The output is then compared to the correct output, which is the label assigned to the training data, and

a loss function is applied to compute the difference between the predicted output and the actual output, which is the loss. The goal during the training process is to minimize this loss; thus, the backpropagation step involves computing the gradient of the loss function with respect to the weights for each layer, which is accomplished using chain rule.

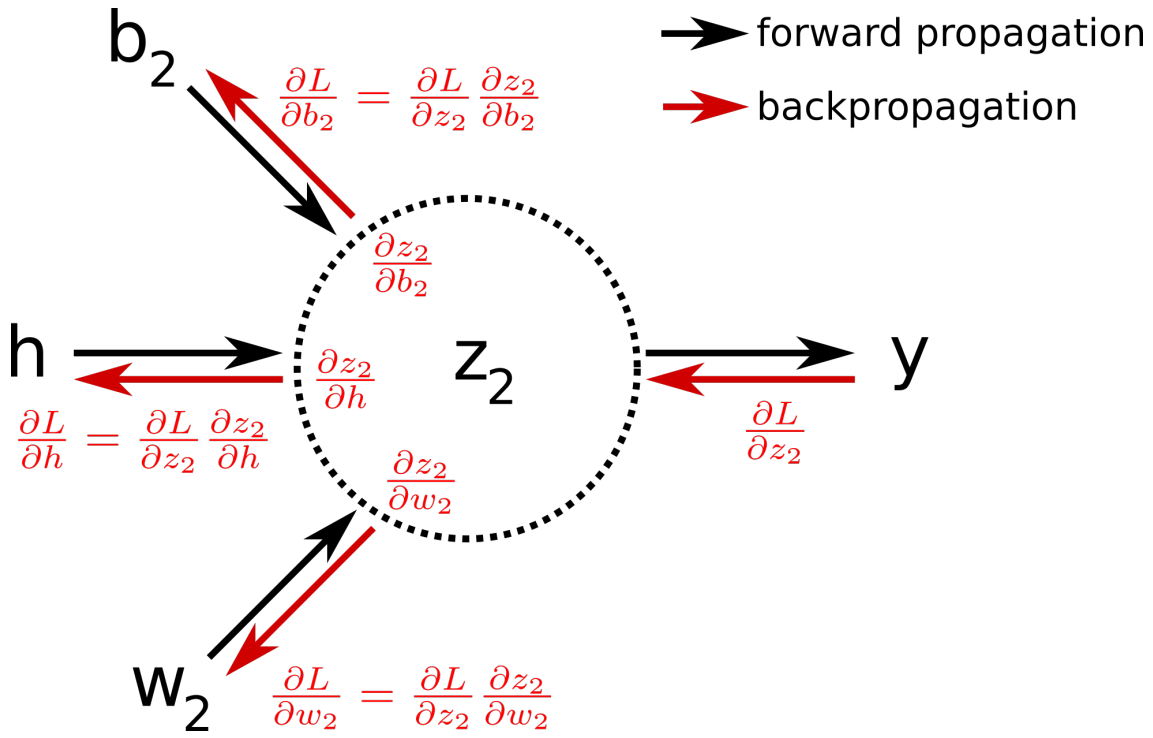


Figure 2.3: Backpropagation at a neuron [4]

Figure 2.3 shows an example of backpropagating through a neuron with function z_2 and loss function L. The partial derivative of the loss function L is taken with respect to each of the weights, h and w_2, and the bias, b, using the chain rule. The partial derivatives are then used to update their respective weight and with this change in weights, ideally for the next iteration of training, the output from the model is closer to the ground truth label and the loss decreases.

2.6 Graph Neural Network(GNN)

A graph neural network (GNN) takes as input a graph, which consists of a set of nodes and edges that connect pairs of nodes. In this thesis, we will use the term GNN to refer exclusively to spatial GNNs. These graphs can either be directed or undirected, but we used directed edges in this thesis because there exists a clear hierarchical structure in an abstract syntax tree (AST). Each node in the input to the GNN is represented by a node vector, or node embedding, and the set of edges can be represented using an adjacency matrix. A GNN learns node representations for each node through information being propagated through the edge connections [33].

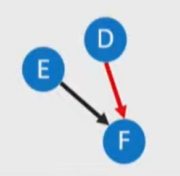
GNNs in general have the following 3 steps for each layer, which is shown in Figure 2.4b: (1) a message passing step, (2) an aggregation step, and (3) an update step:

1. **Message passing:** For each outgoing edge in a node v, a function is applied that takes as input node v, an outgoing edge to node u, and node u, and computes a vector, or message, that is passed to node u. This process occurs simultaneously for all nodes [6].

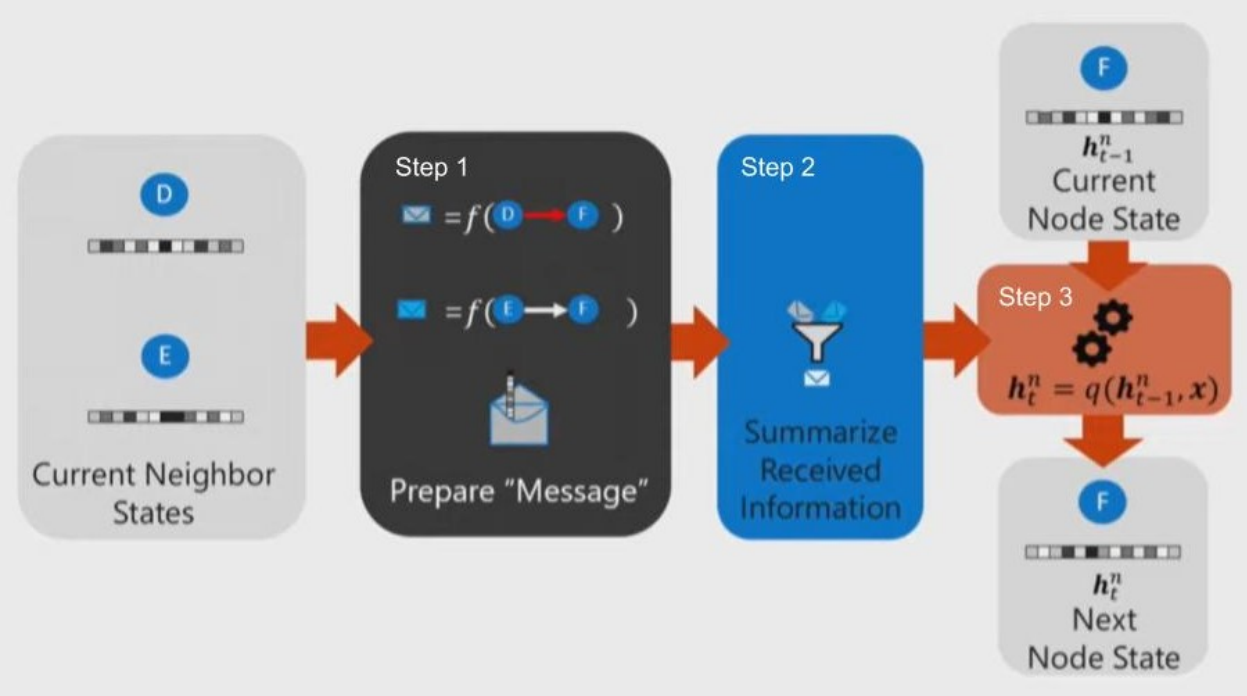
2. **Aggregation:** Each node applies an order-invariant function to aggregate all incoming messages, which outputs a vector that summarizes the messages [6].

3. **Update:** Each node applies a function that takes as input the current node's node embedding and the summarized messages vector from the aggregation step to generate a new node embedding for the current node [6].

From these steps, we can see that the new node embedding after the update step will contain some information about its neighbors, which are the nodes that passed a message to them [6]. A GNN layer is a single iteration of the above algorithm with all nodes updating their



(a) An example graph showing nodes E and D connected to node F, but with different types of edges [6].



(b) General algorithm of a spatial GNN for node F in example graph on the left. **Step 1:** Nodes D and E generate messages based on their node embeddings, their edge type connecting to node F, and node F's node embedding. **Step 2:** Node F receives a message from node D and E, and summarizes the messages into a single vector using an order-invariant function. **Step 3:** Node F updates its embedding using a function that takes as input Node F's current embedding and the summarized messages [6]

Figure 2.4: General algorithm for spatial GNN

embedding simultaneously for each layer before the start of the next layer. Thus, after the first layer, all node embeddings will have some information about their neighbors, and in the second layer, this information will be passed along and each node will now have information about its neighbor's neighbors. For example, given nodes u, v, and w where u has a direct edge to v, and v has a direct edge to w, in the first layer, v will gain information about u's node embedding and w will gain information about v's node embedding, which means after the update function, node v' will contain information of the original node v and u, and node w' will contain information of the original node w and v. In the second layer for node w', node w' will receive a message from v', which contains information about u, thus, after the update function, w'' will contain information of node w, v, and u.

Figure 2.4a shows an example graph with nodes E and D connected to node F with different types of edges, which is denoted by the difference in the edge color. Figure 2.4b shows the previously described GNN algorithm for updating node F when the GNN contains a single layer. In the figure, $(t-1)$ is the current training iteration, or epoch, and x is the vector storing the summarized messages.

2.7 Gated Graph Neural Network(GG-NN)

In this thesis, we use the gated graph neural network (GG-NN) [33]. In the message passing step, for each node, it takes its node embedding and appends 0's to the end depending on the user defined length for the hidden state hyperparameter, and passes it along its edges [33]. For its aggregation step, it uses the summation function to aggregate all incoming messages, and for its update step, it uses a GRU cell to compute the new node embedding [6].

2.8 Graph Classification using GNN

A deep learning model for a graph classification task takes as input a graph and outputs a class label for the input graph. The general architecture of a model for this task starts with a GNN layer followed by a multilayer perceptron layer and a global pooling layer, which aggregates all node embeddings into a single output vector [55]. The final vector is then given as input into either a sigmoid function for binary classification or a softmax function for multi-classification. Zhou et al. points out that this method hinders graph classification because it does not make full use of the learned node embeddings, and instead proposes using traditional neural networks, like a 1-D convolution neural network (CNN), to extract relevant node embeddings before applying a pooling and fully connected layer; however, this approach requires the nodes to have a consistent node order [55].

2.9 Embeddings

An embedding maps an item to a relatively low dimensional vector such that similar items have vectors are closer to each other, meaning it would have a smaller Euclidean distance compared to 2 items that are more dissimilar [22]. A word embedding is a relatively low dimensional vector used to represent a word where the distance between vectors is ideally an indication of their semantic similarity with similar words being closer together. Similarly, code embedding is a vector that represents a piece of code, and a node embedding is a vector that represents a node in a graph. These embeddings are necessary because machine learning models cannot directly process raw text or any input that is not a matrix of numbers. Embeddings differ from other ways of representing text as a vector, such as one-hot encoding, in that each row in the column vector represents a feature. Thus, embeddings describe a

word and the words relationship with other words as shown in Figure 2.5. An embedding only has meaning in the context of other embeddings, otherwise it is a normal vector with values, and these other embeddings have to have been learned along with this embedding; in other words, 2 sets of embeddings that were not learned together cannot be intermingled, even if the words the embeddings represent are the same in both sets because the same word might have a different vector value.

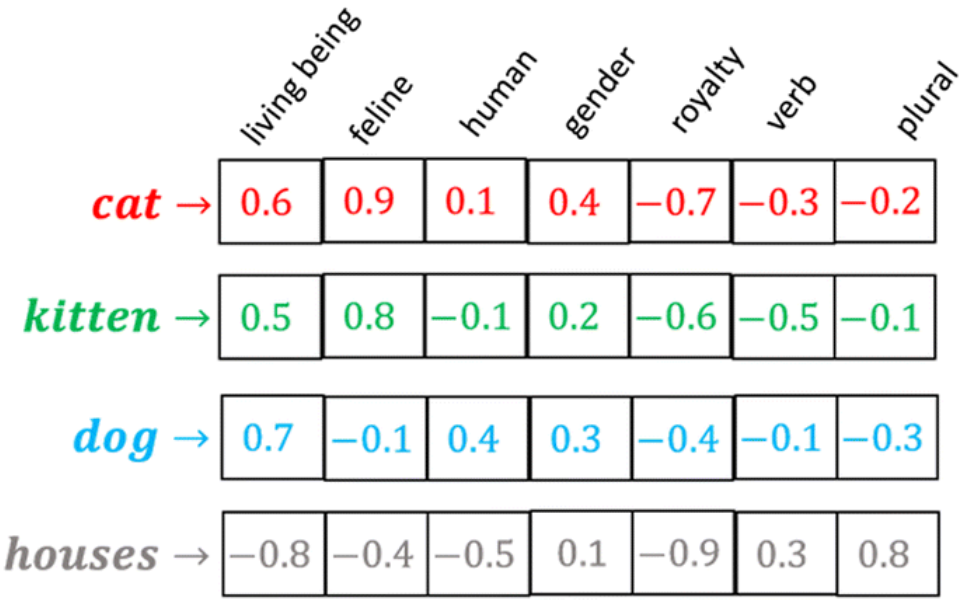


Figure 2.5: Example word embeddings [20].

Figure 2.5 shows an example of word embeddings with features for the words, cat, kitten, dog, and houses. Words that share a similar feature will have a similar value for that feature, such as the living feature for cat, kitten, and dog. On the other hand, as a difference in that feature will result in a very large difference in value for that feature, such as the living feature for cat and house. As we would expect, because cat and kitten are very similar to each other, their corresponding word embedding vectors are also very similar compared to the dog and house word embedding. In addition, because of the many dissimilar features of house compared to cat, kitten, and dog, we can see that the house word embedding is far

away from the other 3 words in terms of its euclidean distance.

In this research, we use the AST of a function, which contains words from the original source code as its node value; thus, code embeddings can be used to generate embeddings for the node embeddings, which are the nodes in the AST. To generate code embeddings, we flatten the function in the source code onto a single line and treated the flattened function as a sentence, which allows us to use word embedding models (See Section 3.3). The earliest word embedding was word2vec [41], which uses neural networks to compute word embeddings based on the words' context [11]. However, it ignores how often some context words appear in a word's context, which is known as the co-occurrence of a word, and this deficiency was addressed in the GloVe model [43], which computes embeddings that also contain information about a word's co-occurrences with other words [11]. Another issue with word2vec was that it does the address the problem on how to create embeddings for words that are not in the vocabulary because word2vec learns an embedding for a word, thus any changes to the word, such as a tense change, will result in a different embedding that word2vec needs to learn [11]. To overcome this issue, fastText [8] was introduced, which learns embeddings for subwords, or parts of a word, and these subwords are combined to compute the embedding of a word [11].

2.10 Stratified K-fold Cross Validation

Stratified k-fold cross validation is a technique that can be used to tune the hyperparameters of a model. K-fold cross validation is a process where the training dataset is randomly split without replacement into k equal groups, known as folds, and k models are trained with each model selecting a different dataset as the validation dataset and the rest of the $k-1$ datasets as the training dataset.

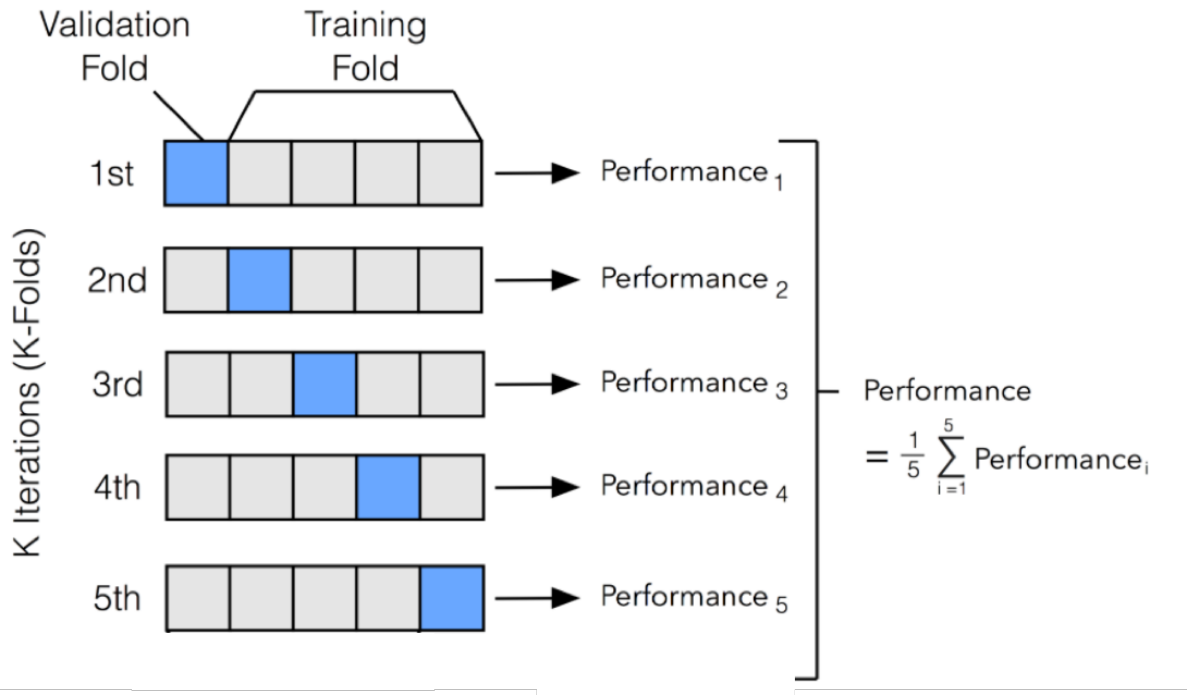


Figure 2.6: Example of k-fold cross validation where k=5 [2]

Figure 2.6 shows an example of 5-fold cross validation where each block in the figure is a fold and combining all 5 folds gives the original dataset. In stratified k-fold cross validation, each of the k-folds have a relatively similar ratio of data points for each class label. As shown in Figure 2.6, each iteration of the cross validation algorithm produces a different trained model with its own performance measure, which is then averaged to obtain the final performance for the overall model with the chosen hyperparameters. Sometimes k-fold cross validation is performed multiple times and the performance across all cross validation instances are averaged. The main reason for this is because how the data is split can directly impact its final performance. For example, hypothetically, when randomly splitting, all the images of a cat laying down is in fold and all the images of a cat standing are in another fold. Then, when the model is trained on the images of the former images and tested on the latter images, the model's performance will likely be relatively low because it only learned to identify cats

laying down. Thus, to reduce the effect this has on the averaged performance, the k-fold cross validation procedure is repeated multiple times, and this is known as repeated k-fold cross validation.

2.11 Handling Imbalanced Dataset

As explained in Section 2.5, the machine learning model learns through backpropagating the loss computed where the loss function typically assumes a relatively equal proportion of data samples for each class [9]. Thus, the more data samples a class label has, the more the model weights will be updated to correctly predict this class label. Therefore, in an imbalanced dataset where there exists class labels with very few data samples compared to other class labels, it will have minimal impact on the weights because the changes will be neutralized and changed to better suit predicting the majority class labels. From a broader perspective, the goal of a machine learning model is to minimize the total loss, and because the data samples in the majority class will have a larger impact on the loss, the weight updates will be more impacted by the majority class. Thus, the model might learn features specific to the data samples in the majority, which decreases the accuracy of correctly labeling data samples in the minority class.

One technique used to handle imbalanced dataset is weight balancing. Weight balancing balances the weight of each of the classes so that when computing the loss during back-propagation, the minority classes will have a greater impact on what the model learns [51]. Another technique used is random oversampling, which random duplicate examples in the minority class with replacement [10]. This technique attempts to balance out the number of samples in each class; however, models trained using this technique tend to overfit the training dataset because of the duplicates [10]. There exists other techniques, such as

SMOTE [12] and its variants; however, these techniques generate synthetic data using the vector representation of the data, which is not easily applicable to text based data [21], which is the type of data used in this research.

2.12 Evaluation Metrics

We used accuracy, precision, recall, and F1-score to evaluate the performance of the deep learning models with the latter 3 metrics used for skewed datasets. The following are the equations for the latter 3 given metrics.

$$Precision = \frac{True\ Positive}{True\ Positive + False\ Positive} \tag{2.1}$$

$$Recall = \frac{True\ Positive}{True\ Positive + False\ Negative} \tag{2.2}$$

$$F1-score = 2 \times \frac{Precision \cdot Recall}{Precision + Recall} \tag{2.3}$$

Chapter 3

Review of Literature

3.1 TEE Refactoring Tools

Prior work has mainly focused on developing tools to facilitate the process of moving functions into a TEE with little focus on automating the process of finding inputs to the developed tools. In the approach used by Rubinov et al, developers are required to use program annotations to indicate methods that read and return sensitive data [47]. Glamdring [35] and RT-Trust [37] both use a similar approach with RT-Trust requires the developer to indicate functions that contain sensitive data using the RT-Trust domain-specific annotation, and Glamdring requiring developers to annotate the source and sink variables that handle sensitive information. TEEShift requires the developer to specify the functions to place inside a TEE [31]. TZSlicer requires the developer to provide a list of program variables that are the source of sensitive data, which is then fed into a tool along with some other developer generated input to output functions, blocks of code, or lines of code that handle sensitive data [54]. Compared to these works that focuses on building refactoring tools to refactor pieces of code that has been determined by a developer to require refactoring, the work presented in this thesis focuses on recommending functions to help the developer decide what functions to place within a TEE.

However, there are some existing TEE tools that are or include a recommendation system. TEE-DRUP uses NLP techniques to find a list of variables that could potentially be storing

sensitive data and requires the developer to verify the correctness of this list. Then, based on the verified list, TEE-DRUP uses Taint Analyzer to determine which functions do not operate on sensitive data, and then requires the developer to verify and confirm which functions to move to the outside world [36]. While TEE-DRUP also includes a recommendation system, its focus is solely on recommending sensitive variables within the functions that are placed within the TEE [36], whereas ML-TEE recommends sensitive functions that are placed inside or outside of a TEE. Dhar introduces a recommendation system that classifies whether a given software requirement is relevant or not relevant to TEE [17], which also has a different focus compared to ML-TEE's.

3.2 Automated Detection of Vulnerable Functions

The task of detecting vulnerable functions is similar to the task the recommendation system introduced in this thesis performs, which is determining whether a function should be placed within a TEE or not. Russell et al. used NLP techniques to automate the approach for vulnerability detection in source code using CNN and RNN based models. The key idea behind this approach is the similarities between code and writing [48]. Chernis et al. used a supervised machine learning approach where they used handpicked features, such as function length and n-grams of functions as input for models such as the Naive Bayes classifier to classify if a function is vulnerable or not [13].

Deep learning models are used instead of traditional machine learning techniques because traditional machine learning models require creating handcrafted features, which is a time consuming and error prone process, and can lead to task specific features [34]. On the other hand, neural networks allow features to be extracted automatically and with multiple layers of abstraction, which can lead to the model discovering latent features that a human expert

might not consider, and thus, increasing the feature search space [34]. Sonnekalb introduces a machine learning approach to vulnerability detection using tokens, control flow graphs and ASTs as input into deep learning models such as LSTMs [52]. Feng et al. flattens the AST using the preorder traversal algorithm and then uses word2vec to embed each of the nodes in the resulting node sequence, which is then fed into a Bi-GRU to develop an automated method for vulnerability detection [18]. Zhou et al. combines ASTs, control flow graphs, data flow graphs, and natural code sequences into a joint graph and uses a GG-NN to learn node embeddings before feeding the output graph into a Conv layer that they designed to classify whether the the input function is vulnerable or not [55]. Different from these approaches, we focus on using deep learning techniques and the ASTs to recommend functions to be placed within a TEE.

3.3 Using Embeddings to Represent Code

There are 2 main ways of converting code into vectors, which is necessary for deep learning models: using code embeddings to embed source code or using node embeddings to embed nodes in a graph representation of source code. Earlier works flattened the source code and treated it as a sentence and used traditional NLP techniques to process it. Harer et al. used both a bag-of-words approach as well as word2vec to generate a vector for each of the tokens in a source code function, and found that using word2vec embeddings worked better compared to the bag-of-words approach [24]. Allamanis et al. represents the program as a graph by adding additional information to the program's AST where the node embeddings in the graph are computed by subtokenizing tokens and averaging the subtoken's embeddings before concatenating it with the node type embedding to form a single embedding for the node [5]. Zhou et al. uses a similar approach of embedding

the node values using a pre-trained word2vec model and using label encoding to embed the node type before concatenating the 2 embeddings to form a single node embedding, except, Zhou et al. does not subtokenize the tokens in the node value [55]. Code2vec learns continuous distributed vector representations for code using a neural network that is trained on a collection of AST paths that are decomposed from the code to allow the network to learn atomic representations of each path while simultaneously learning how to aggregate a set of them [7]. Zhou et al. also represents the code as a graph and uses GG-NN to learn node embeddings [55]. Kanade et al. investigated the use of using pre-trained word embeddings to improve models in prior works by using the state-of-the-art NLP model at that time, which was BERT, and modified it to create CuBERT to generate code embeddings given tokenized Python source code [27]. The deep learning model used by ML-TEE takes as input the AST representation of a function where the nodes in the AST are embedded using an approach similar to one used by Allamanis et al. [5] and Zhou et al. [55]. We subtokenize the node values before embedding each subtoken and averaging the embeddings, and instead of using label encoding to encode the node types like Zhou et al. [55], we used one-hot encoding because there does not exist an absolute unique ordering for the node types, which would be implied by using label encoding.

Chapter 4

Design of ML-TEE

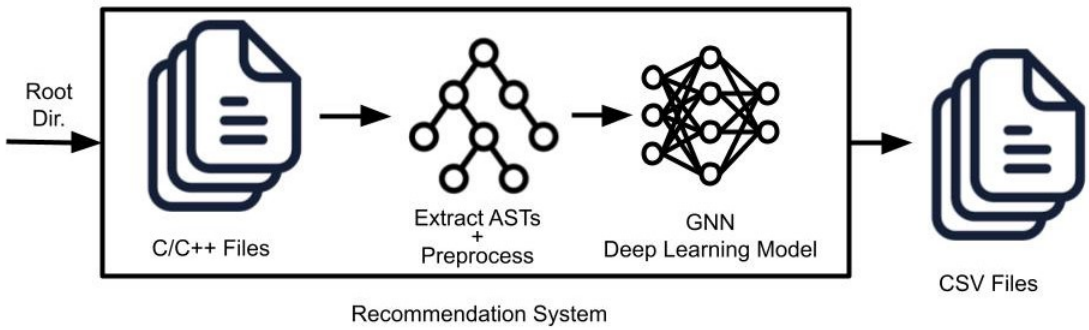


Figure 4.1: Outline of Recommendation System. The developer inputs the root directory of the software/application, and the recommendation system finds all the C/C++ files within the project, extracts the ASTs of the functions within these files and preprocesses them, then passes them into the trained machine learning model to output a probability and prediction, which is written to a the c/c++ file's corresponding CSV file along with the function name and line information.

4.1 Outline of ML-TEE

Figure 4.1 shows an outline of the recommendation system. ML-TEE takes as input the root directory of a C/C++ based application and finds all of the C/C++ files. It then extracts the AST of each function and method located within the file. The nodes in the ASTs are then preprocessed and converted into node embeddings using a word2vec model before being processed by a machine learning model, which returns both the probability of

a function belonging inside a TEE as well as ML-TEE's recommendation on whether the function should be placed inside the TEE based on a heuristically determined threshold value for the probability. The output of the recommendation system is a CSV file with each file corresponding to a C/C++ file. Table 4.1 shows an example of a line of output in the CSV file. We will define functions to be placed inside a TEE in the context of Intel SGX, which are function prototypes to be placed in the trusted section of an EDL file.

Table 4.1: Formatted example of a line of output with headers in the CSV file generated by the recommendation system. The start line is the file line of the first function line, and the end line is the file line of the last function line. For the prediction, yes means the function should be placed inside the TEE and no means it should not.

Function Name	Start Line:End Line	Probability	Prediction
main	32:232	0.31%	no

4.2 AST Extraction and Preprocessing

We assume that all necessary EDL files are already generated. Functions from the LibClang library were used to generate and traverse the AST of each of the C/C++ files. We extracted the functions and methods by searching for nodes with a node type of FunctionDecl or CXXMethod respectively, and stored node pairs that are connected by an edge while traversing through the subtrees to extract the AST. Figure 4.2 shows an example of a function that was placed inside the TEE, and Figure 4.3 shows its corresponding AST before preprocessing it. Each node in a node pair consists of a LibClang node type and a node value, which can also include the node value's type information if has one: for example, in the AST shown in Figure 4.3, for the node type FunctionDecl, it has the node value *ecall_create_wallet* with the type *int*. During this process, we also stored the function

and method names as well as their starting and ending line number, which will be in the recommendation system's output CSV file for this function.

```
1  int ecall_create_wallet(const char* master_password)
2  {
3      //
4      // OVERVIEW:
5      //  1. check password policy
6      //  2. [ocall] abort if wallet already exist
7      //  3. create wallet
8      //  4. seal wallet
9      //  5. [ocall] save wallet
10     //  6. exit enclave
11     //
12     //
13     sgx_status_t ocall_status, sealing_status;
14     int ocall_ret;
15
16     ...
17
18     // 1. check passaword policy
19     if (strlen(master_password) < 8 || strlen(master_password)+1 >
           MAX_ITEM_SIZE) {
20         return ERR_PASSWORD_OUT_OF_RANGE;
21     }
22
23     ...
24 }
```

Figure 4.2: Example code fragment for a function inside the enclave. The ellipsis represent missing code that was taken out of the original function to simplify it.

The ASTs are then preprocessed by first filling in some node values for specific node types that were unable to automatically find its node value, such as the node value case for CaseStmt in a switch/case block. In addition, we replaced all string and character literals with the <STR> token [28], and replaced all integer and float literals with the <NUM> token [14], to help reduce the vocabulary size. We assumed that sensitive information is not hard-coded, and from inspecting the source code, we found that the majority of the string literals are logging statements, thus, we can regard this as extraneous information that is not useful for our specific task. For nodes that we were unable to automatically extract a

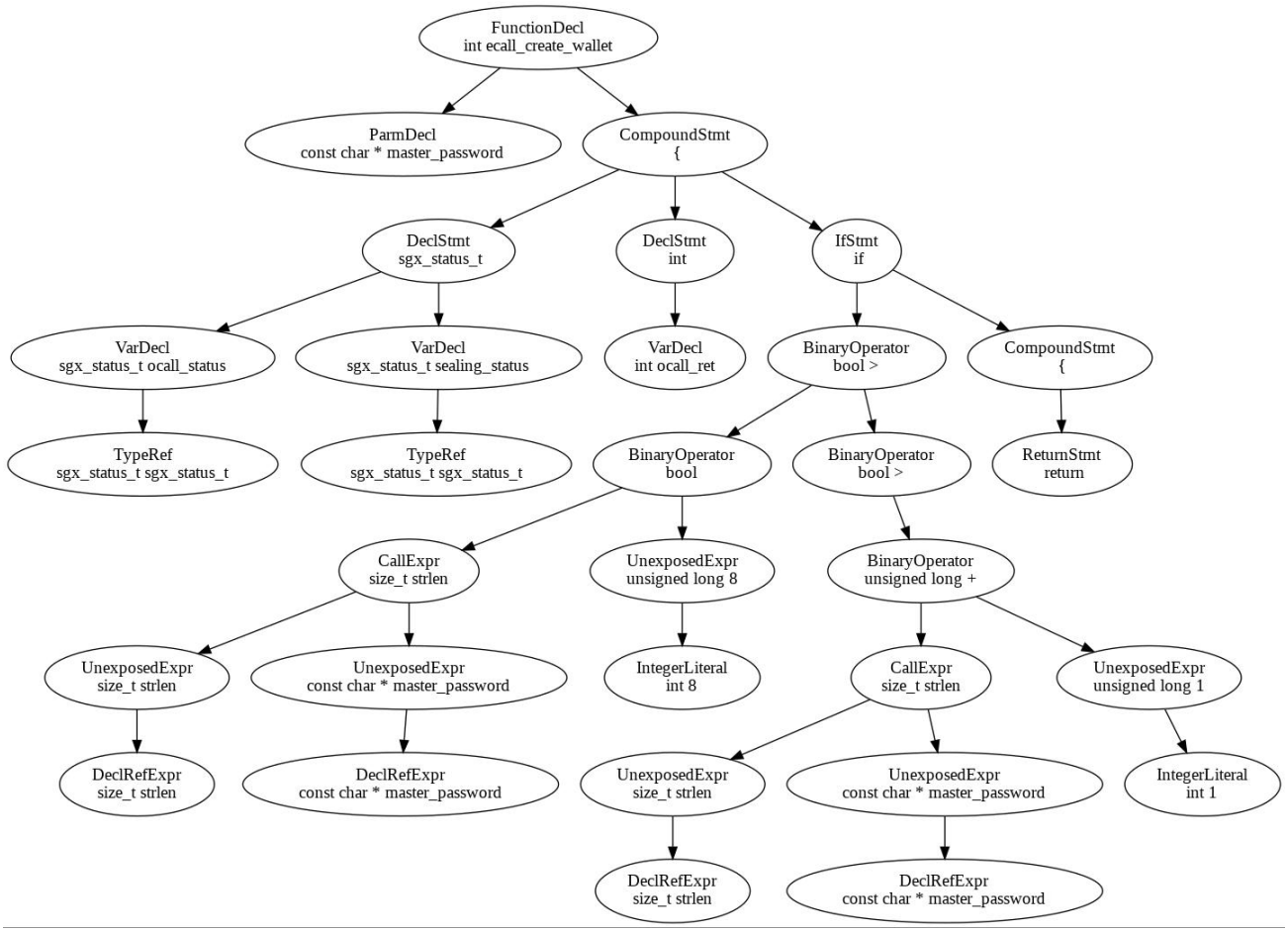


Figure 4.3: The extracted AST of the code fragment shown in Figure 4.2 before preprocessing it.

node value for, we set its node value to be <UNK>, the unknown token, which was set to be the zero vector [14].

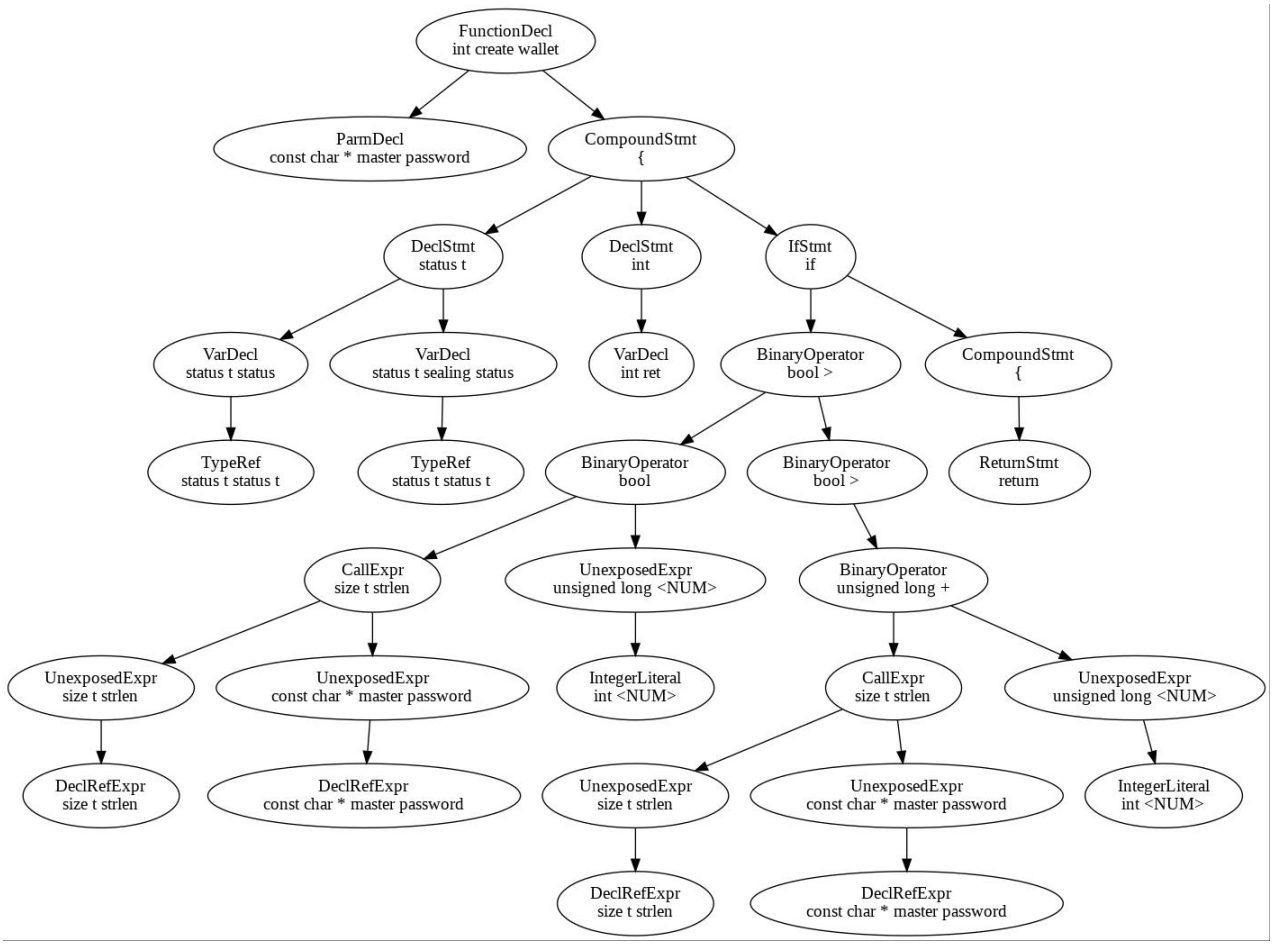


Figure 4.4: The final AST after preprocessing it, which will then be passed to the machine learning model to be classified.

The node values are then split on camelCase, snake_case, and PascalCase to further reduce the vocabulary size [53]. More importantly, this helps mitigate the closed vocabulary problem by reducing the out-of-vocabulary rate because in general, the subtokens are more common compared to the complete token [28]. We also removed Intel SGX specific subtoken words such as ecall and ocall from variable/function names so that the final model does not rely on these features to classify functions. Figure 4.4 shows the final AST after preprocessing. The

node value embedding is generated by first embedding the subtokens, and then averaging the vectors to get a single embedding for the node value. If a subtoken is not within our vocabulary, then it is represented by the <UNK> token, which is the zero vector. The node type is embedded using one-hot encoding, and the complete node embedding is created by concatenating the node value vector and the node type vector [55].

We followed the design used by Zhou et al. and fixed the total number of nodes to be the first 400 nodes encountered in the traversal of the AST [55]. However, we did not exclude ASTs that contained nodes exceeding this limit and instead followed traditional NLP techniques to remove extra nodes and add padding vectors to ASTs with fewer than 400 nodes [25]. We removed all edges connecting a node not in the first 400 and in the case an AST does not contain 400 nodes, we added in padding nodes, which are zero vectors with no edges connecting to or from these nodes to ensure all ASTs have exactly 400 nodes.

4.3 Model Architecture

The initial input of node embeddings, with an initial embedding size of 286, and their edge information is first given as input into a gated graph recurrent layer. We set the number of output channels to be 300 and the sequence length to be 2. We tested out 3 different prediction layers, a method adapted from Kung-Hsiang's [30] model, which uses global pooling layers to aggregate the output before passing it to a fully connected layer, and the methods proposed by Zhou et al., which uses either a CNN or LSTM layer instead of the global pooling layers [55]. LibClang traverses all ASTs using depth-first search method, which means all nodes in each AST are ordered consistently. In addition, GG-NN does not change the ordering of the nodes; thus, we can use the latter 2 methods proposed by Zhou et al. [55].

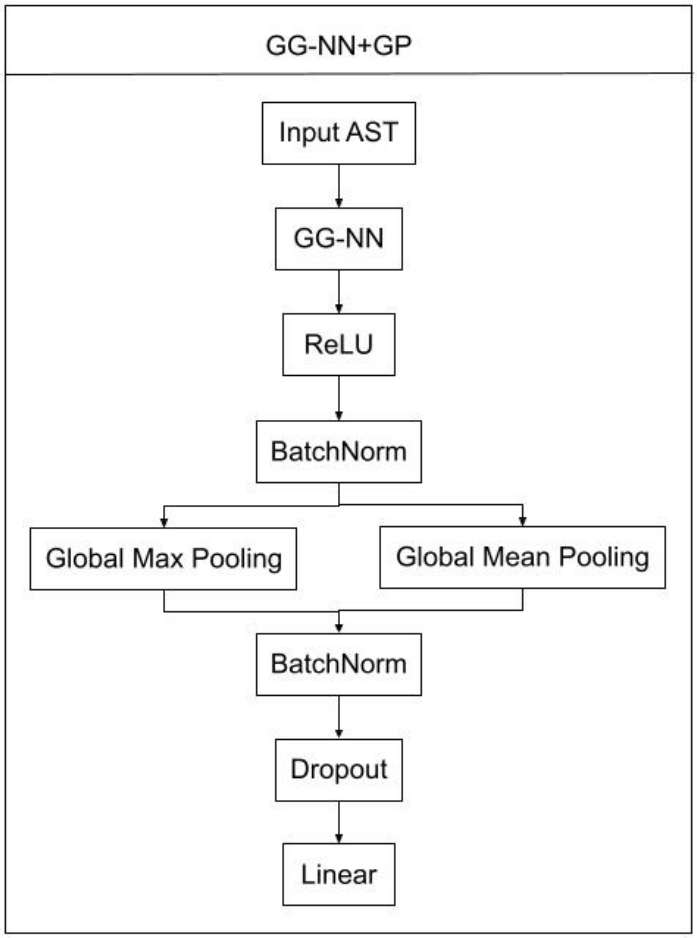


Figure 4.5: GNN model that uses a pooling layer after the GNN layer, which is then followed by a fully connected layer.

Figure 4.5, 4.6, 4.7 shows the architecture of the 3 models tested: GG-NN+GP, GG-NN+CNN, GG-NN+LSTM respectively. For the GG-NN layer, we used a hidden layer size of 286 for all 3 models.

GG-NN+GP The output of the GNN layer is given as input into both a global mean pooling and global max pooling layer with the 2 output vectors being concatenated to form a single vector. Then, this is given as input to a fully connected layer followed by a sigmoid function layer to form a prediction.

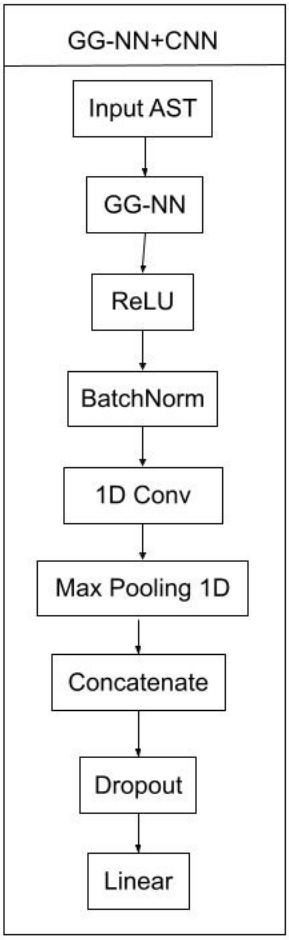


Figure 4.6: A GNN model that uses a 1-D convolution layer immediately following the GNN layer before aggregating the learned features using a pooling layer into a fully connected layer.

GG-NN+CNN The output of the GNN layer is given as input into a 1-D convolution layer. In the 1D convolution layer used, we used a kernel size of (1, 2) and a filter size of 128. The output from this layer is then given as input to the ReLU activation layer, which is then followed by a global 1D max pooling layer. The output from the max pooling layer is then given as input to a fully connected layer.

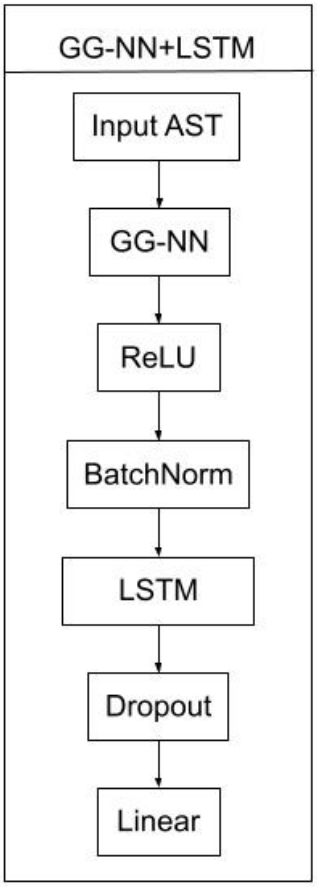


Figure 4.7: A GNN model that uses an LSTM layer and uses the final hidden layer of the LSTM as input into the fully connected layer.

GG-NN+LSTM The output of the GNN layer is given as input into a LSTM layer. In the LSTM layer used, we set the number of layers to be 1 and used a hidden dimension of size 128. The final hidden layer of the LSTM layer was then given as input to a fully connected layer.

Chapter 5

Dataset

The initial list of Intel SGX projects from GitHub was curated by Dhar [17]. We downloaded projects that use Intel SGX from GitHub. We focused on classifying C/C++ functions; thus, we also removed projects that did not contain functions written in these programming languages. We then extracted and preprocessed the ASTs free of compilation error using the method described in the previous section, and removed all duplicate ASTs after the final preprocessing step, which resulted in 16,177 unique ASTs from 270 different GitHub projects.

5.1 Data Labeling and Assumptions

We assumed that the developers of the GitHub projects we used for training and testing correctly identified and placed only functions that should be placed inside the TEE within the TEE. We used the Edger8r tool to first generate the 4 files containing the trusted and untrusted proxy functions. Then, we labeled each of the unique functions based on whether the developer of the application that uses this function placed it inside the TEE. Thus, all functions were placed into 1 of 2 categories: the function or method belongs inside a TEE or it does not, with the former being the positive case and the latter being the negative case. In the case where a conflict occurred, a function was not placed inside a TEE for one application but was placed inside a TEE for another project, we resolved it by labeling the

function as belonging inside a TEE.

To facilitate the labeling process, we first created a parser to parse the EDL files to extract all the function prototypes within the trusted section of the enclave. The file that defines the body of a trusted function in an EDL file must contain a *#include* line pointing to the location of the file containing the trusted proxy functions, which is generated by the Edger8r tool when executed on the EDL file. We then wrote a script to automatically label functions as the positive case if it has the same function name and parameter type information as a function prototype in the trusted section of an EDL file, and the *.c* or *.h* file that contains this function's body has a *#include* statement that points to the trusted proxy function file for this EDL file. We then manually labeled the functions that the script did not label as the positive case, but had the same function name and parameter type information as a function prototype in the trusted section of an EDL file. Then, all of the remaining unlabeled functions were labeled as the negative case.

5.2 Experimental Setup

The total number of positive cases was 512 and the total number of negative cases was 15,665. The dataset was then randomly split into a 80/20 train and test dataset using a random stratified split to preserve the ratio of positive to negative cases, which is approximately 1:31 and shows that this is an imbalanced dataset. In the training dataset, the total number of positive cases was 410, and the total number of negative cases was 12,532. In the testing dataset, the total number of positive cases was 102, and the total number of negative cases was 3,133.

A word2vec model was trained on the source code of the functions and methods in the training dataset to learn embeddings for the node values. Because C/C++ code uses semicolons to

denote the end of a statement, a function or method can be collapsed into a single line, which we did. We first preprocessed each of the functions and methods in the training dataset by removing the comments before tokenizing it using functions from the Pygments[1] library. The tokens were then placed onto a single line with a space separating each token and then we replaced all character and string literals with the <STR> token and all integer and float literals with the <NUM> token. Then, we subtokenized each token by camelCase, PascalCase, and snake_case into spaced separated subtokens. The same Intel SGX words that we removed during the preprocessing of the AST described in Section 4.2 were also removed from the subtokenized source code. The flattened functions and methods were then used to train a word2vec model from the Gensim [45] library with an embedding dimension of size 200 and the default parameters. The model was trained for 10 epochs and ignored words that occurred less than 5 times. The word2vec model as well as all other machine learning models were all trained on a Tesla P100 GPU.

We one-hot encoded the node types, which resulted in a vector of length 86. The final embedding size for a node was 286.

[1] https://pygments.org/

Chapter 6

Evaluation

We want to answer the following evaluation questions: **Q1. Model Performance** How well do the 3 models given in the previous section perform and which one is the best? **Q2. Performance of Models Using Imbalanced Dataset Techniques** What is the performance of the model after applying techniques to handle training on an imbalanced dataset? **Q3. Pilot Study** Does this recommendation system improve a software developer's accuracy and reduce their time spent on identifying functions that should be placed inside a TEE? **Q4. Explaining Model Features** Why and how does the trained deep learning model work, i.e. what does it learn?

6.1 Model Performance

We introduced 3 different models in Section 4.3, GG-NN+GP, GG-NN+CNN, and GG-NN+LSTM, and we evaluated each of them in this section.

6.1.1 Experimental Design

We fine tuned the hyperparameters of our model using stratified k-fold cross validation. We trained each of the classification models identified in the previous section with the Adam optimizer with a learning rate of 0.00001, except for the GG-NN+GP model, which used

a learning rate of 0.0001. We used a batch size of 64 for all the models and used L2 regularization to avoid overfitting. In addition, we trained all the models using the binary cross-entropy loss function. We created the models using the PyTorch [42] library and PyTorch Geometric library [19]. We computed a threshold value for each of the models by averaging the optimal threshold values for each of the stratified k-fold cross validation models. The averaged threshold values were then used as the threshold value for determining whether the output probability from the model on the test dataset means the function is a positive or negative case. For the GG-NN+GP model, the threshold value used was 27.78%, for the GG-NN+CNN model, it was 29.26%, and for the GG-NN+LSTM model, it was 33.98%.

6.1.2 Results

The results of the 3 models tested are summarized in Table 6.1. From Table 6.1, we can see that the GG-NN+GP model has the highest accuracy, F1-Score, precision, and recall compared to the other models with the GG-NN+CNN model performing the second best in all 4 metrics compared to GG-NN+LSTM, which performed the worst.

Table 6.1: Classification accuracy, F1 score, precision, and recall of the models tested on the test dataset in percentages based on their individually computed threshold value. For the GG-NN+GP model, the threshold value used was 27.78%, for the GG-NN+CNN model, it was 29.26%, and for the GG-NN+LSTM model, it was 33.98%.

Models	Accuracy (%)	F1 Score (%)	Precision (%)	Recall (%)
GG-NN+GP	98.86	80.00	89.16	72.55
GG-NN+CNN	98.64	76.60	83.72	70.59
GG-NN+LSTM	98.45	73.40	80.23	67.65

We also plotted the change in F1-score, precision, and recall for the best performing model,

which was the GG-NN+GP, and the results are shown in Figure 6.1.

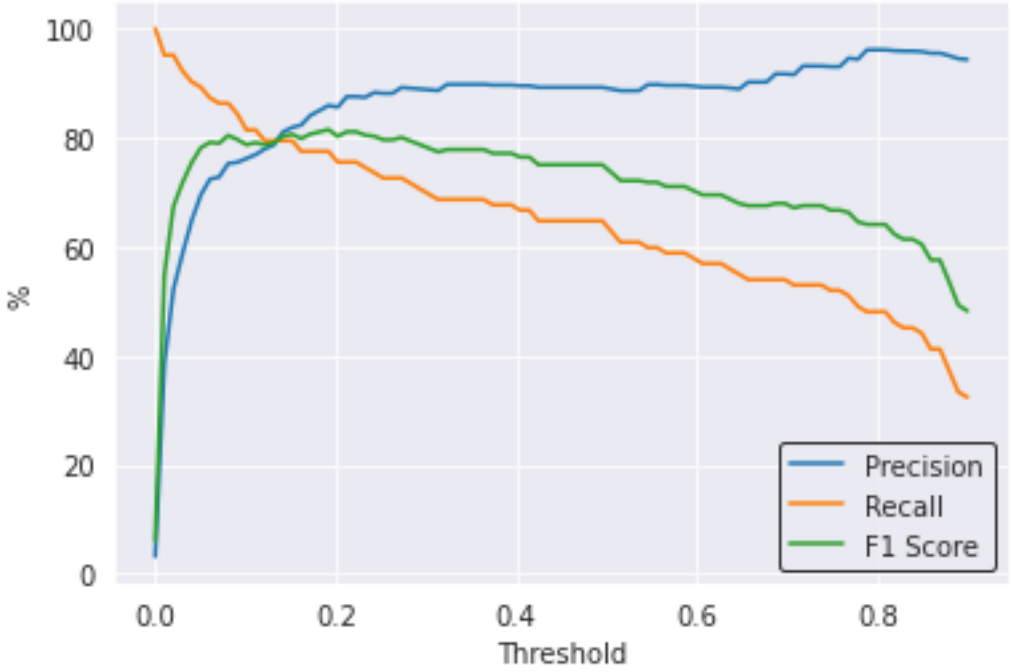


Figure 6.1: F1-score, precision, and recall for different threshold values for the GG-NN+GP model.

6.1.3 Discussion

From Table 6.1, we can see that the GG-NN+GP model was the best performing model, so we decided to use this model in our recommendation system in our subsequent experiments. This model has a very high precision of 89% and a moderately high recall of 73%, which means that while it is unable to find all the functions that should be placed inside a TEE, the ones it does identify are very likely to be correctly identified. Because the GG-NN+GP model was the best performing model of the 3, we used this in our recommendation system to make predictions and used it as the model for our subsequent experiments.

From Figure 6.1 we can see that as the threshold probability increases, the precision as

expected also increases with a threshold value of 0.8 having a precision of at least 90%. However, as the threshold value increases, the recall decreases; thus, with a higher threshold value, the model will predict fewer functions that need to be placed inside the TEE and will miss more functions that need it, but the functions the model does identify have a very high probability of being correctly identified as needing to be placed inside the TEE.

6.2 Performance of Models Using Imbalanced Dataset Techniques

As shown in Section 5.2, the dataset we used was imbalanced, so we used some existing techniques to try and improve GG-NN+GP's performance.

6.2.1 Experimental Design

We used the weight balancing and random oversampling techniques to try and improve the model's F1-score. In the experiment using the weight balancing technique, we increased the weight of the positive case samples by a factor of 31 because the ratio of positive to negative cases is 1:31. For the random oversampling experiment, we randomly duplicated samples from the positive case samples until the total number of positive case samples equals the total number of negative case samples. For all of these experiments, we only made these modifications to the training dataset, both during the cross validation process as well as training the final model. The only other change made to the training of the GG-NN+GP model, which was used in this part of the experiment, was that for the weight balancing experiment, the learning rate was changed to 0.00005. For the weight balancing model, the threshold value used was 82.04%, and for random oversampling, the threshold value was

61.08%,

6.2.2 Results

A summary of the results is shown in Table 6.2. From Table 6.2, we can see that the model trained using the random oversampling technique performed the best.

Table 6.2: Classification accuracy, F1 score, precision, and recall of the GG-NN+GP model with different modifications tested on the test dataset in percentages.

Methods	Accuracy (%)	F1 Score (%)	Precision (%)	Recall (%)
Weight Balancing	98.55	76.85	77.23	76.47
Random Oversampling	98.61	77.61	78.79	76.47

6.2.3 Discussion

By comparing the results from Table 6.2, with the base GG-NN+GP model from Table 6.1, we can see that using either technique results in a worse performing model in terms of accuracy, F1-score, and precision; however, the recall for both models trained with either technique is higher compared to the baseline model, which has a recall of 72.55%.

A potential reason for the decrease in accuracy, F1 score, and precision is because with the many duplicate positive cases in the training dataset, the model overfit the training dataset, meaning it learned too many features that are specific to the training dataset, which means it would not be able to generalize to new data very well.

6.3 Pilot Study

We wanted to understand whether the recommendation tool would be useful to software developers and whether it has any impact on the time taken to identify what functions should be placed within the TEE, so we conducted a pilot study.

6.3.1 Experimental Design

We conducted a pilot study involving a control and intervention group with participants in both groups given the same task of analyzing a GitHub project and identifying functions that should be placed inside a TEE. The chosen GitHub project was a simple wallet application and was not included in the list of GitHub projects that formed the train and test dataset. The chosen project was refactored to remove Intel SGX related functionality, and the ASTs of the functions were generated using the procedure given in the previous section. The refactoring process involved removing Intel SGX related libraries and functions, such as the sealing function, as well as renaming functions and variables that contained Intel SGX related words such as *ecall* and *ocall*.

We selected 6 participants who have graduated with a computer science degree, and we ensured that all 6 participants had no prior experience using a TEE by asking them prior to selecting them. The participants were split into 2 groups of 3, which were the control and intervention groups. The intervention group was given a spreadsheet where each sheet represented a C/C++ file in the project and contains information of the functions inside that file, which includes the function name, the start and end line number of the function within the file, and the output from the recommendation system, which contains the probability of the function being placed inside the TEE and its recommendation based on the heuristically determined threshold value. Table 4.1 shows an example of a line within the spreadsheet

given to the intervention group. For the control group, a similar spreadsheet was given, but instead only included that is not useful for determining whether a function should be placed within a TEE or not, and this information includes the function name, the start and end line number of the function within the file, the number of statements in the function, and the modified McCabe cyclomatic complexity value of the function. Table 6.3 shows an example of a line within the spreadsheet given to the control group.

Table 6.3: Example of a line in the spreadsheet given to the control group. The start line is the file line of the first function line, and the end line is the file line of the last function line. This information was obtained by using the *pmccabe* command.

Function Name	Start Line:End Line	# Statements in function	Modified McCabe Cyclomatic Complexity
main	32:232	93	34

Each participant was given a Google Form with a set of instructions introducing the task and a brief introduction to TEE and what functions would benefit from being placed in one. The participants were asked to complete the form in one sitting and to record their start and end times. The form contains a list of functions categorized by what file they are in and participants were asked to check the box next to the function that they believe should be placed inside a TEE. The ground truth labels of each of the functions was based on what functions the original developer placed inside the trusted section of the EDL file. The refactored project contained 18 functions in total with 5 functions that the developers placed inside the enclave.

6.3.2 Results

The results of the pilot study are summarized in Table 6.4. From Table 6.4, we can see that the intervention group, which was given the output from the recommendation system had a higher average accuracy and a lower average time taken.

Table 6.4: Pilot study results. The following results are the averaged results.

Group	Average Accuracy (%)	Average Time taken (minutes)
Control	62.96	14
Intervention	66.67	11

6.3.3 Discussion

From Table 6.4, we can see that by using the output from the recommendation group, the accuracy improved by about 4% and the participants in the intervention group were able to complete the task 21% faster. These results suggest that the recommendation system can help developers more accurately identify which functions should be placed inside a TEE as well as reduce the time taken to complete this task; however, a formal study will need to be conducted to obtain conclusive results.

6.4 Explaining Model Features

We wanted to investigate which nodes in the AST the GG-NN+GP model considers to be important in formulating its decision, specifically, which parts of the source code is important. Liu et al [36] considers the variable name, variable type, the enclosing function name, and the file path to the enclosing function to contain important information on a variable's sensitivity, i.e. how likely the variable accesses sensitive information. From the findings and results presented by Liu et al [36], we decided to investigate the impact function names, and variable names and their type information have on the model's performance. In addition, we tested out parameter names and their type information as well as the names of functions invoked within the enclosed function.

6.4.1 Experimental Design

We investigated the impact function names, parameter names and their type information, variable names and their type information, and the names of the functions that are called within the function have on the model's performance. Thus, we removed different combinations of these 4 pieces of information from all data in the test dataset and re-evaluated our trained model on the modified test dataset. The idea behind this approach is that if removing a specific node value results in a change in the model's prediction, then that node value is important in formulating the model's prediction; if removing it does not change the model's prediction, then that node value is not important to the model.

For variable names and parameter names, we removed all instances where a node value contains the variable/parameter name. We only removed the node values and not the node types. To find the variable names, we searched for node values with a node type of $VarDecl$ to extract the variable names and removed all nodes with a value containing this variable. A similar procedure was followed to remove the parameter names and the names of the functions invoked by the function, except the node type searched for was $ParmDecl$ and $CallExpr$ respectively. For the function name, we removed the node value of the node with a node type of either $FunctionDecl$ or $CXXMethod$.

6.4.2 Results

Table 6.5 shows the results of the trained model re-evaluated on the test dataset with certain node values removed: (1) Removing function names, (2) Removing variables names and their type information, (3) Removing the names of the functions called, and (4) Removing the function's parameters and their type information.

From Table 6.5, we can see by removing all 4 factors, the F1 score significantly decreases

Table 6.5: F1 score of the GG-NN+GP model with different node values removed from the test dataset. (1) Removing function names, (2) Removing variables names and type information, (3) Removing the names of the functions called, and (4) Removing the function's parameters and type information.

Node Values Removed	F1 Score (%)
Removing (1), (2), (3), and (4)	29.75
Removing (1)	71.08
Removing (2)	73.26
Removing (3)	74.71
Removing (4)	69.46

compared the baseline F1 score of 80% for the GG-NN+GP model. In addition, by removing a single factor, the F1 score also decreases by a significant amount, which means that all 4 components contribute significantly to the model's prediction.

6.4.3 Discussion

By comparing the results from Table 6.5 with the baseline F1-score of 80.00% for the model, this shows that the model considers all 4 identified factors to be important with the function's parameters to be the most important because removing function's parameters resulted in the lowest F1-score compared to removing the other 3 individually. In addition, as seen by the very low F1-score with all 4 factors removed from the data, we can conclude that the model works by making use of this information to formulate its output.

Chapter 7

Threats to Validity

In this chapter, we discuss a few internal and external threats to the validity of the results presented in this thesis.

7.1 Internal Threats to Validity

We observed that for some GitHub projects, the developers used a different naming convention for function and variable names within the enclave, such as being prefixed with *ecall_*, compared to those outside the enclave. During the preprocessing step described in Section 4.2, we removed subtokenized words associated with an enclave from all node values; however, we were unable to remove them all because we were not familiar with all of the naming conventions used. Thus, it is possible that the machine learning models were biased towards these words; however, because we subtokenized each node value and averaged the subtokenized embeddings, this helped mitigate the bias introduced by these enclave related words.

For each function in a GitHub project, we labeled it based on whether the developers of the project placed it within the TEE or not and trusted their decision. However, it is possible that the developers might have incorrectly placed a function within the enclave and thus, the deep learning model would have learned incorrect information. We mitigated the impact of these incorrect labels by using functions from a large sample of 270 GitHub projects and

included as many functions as time permitted in our final dataset.

7.2 External Threats to Validity

The dataset used consists of only C/C++ functions from GitHub projects using Intel SGX. One external threat to validity that arises would be whether the results shown extends to other TEE implementations. We subtokenized and represented all words not in the vocabulary with the zero vector, which helped deal with other types of naming conventions used to help mitigate the issue of generalizing to C/C++ based software using other TEE implementations. However, because of the differences in the structure and nodes of the ASTs between different programming languages, our recommendation system that is trained only on C/C++ functions will not generalize well to functions written in a different language.

Another external threat to validity is the introduction of new use cases for TEE outside of the existing ones in our compiled dataset. The new use cases not represented in our dataset might lead to the introduction of new sensitive data and code, which means the recommendation system would not be able to correctly identify these new functions. However, because we used a supervised learning approach to train our model, the model can continue training on the new data instead of having to be completely retrained.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

The problem we addressed in this thesis was how to reduce the software maintenance cost associated with incorporating a TEE into an existing software. We focused on identifying and recommending C/C++ functions that should be placed within a TEE to help developers by reducing the number of functions they need to inspect, which in turn reduces the software maintenance cost. Our approach in the form of a recommendation system serves as the main contribution for this thesis.

We introduced a recommendation system that uses a deep learning model to help reduce the number of lines of code a developer would need to inspect. We tested out 3 different models, GG-NN+GP, GG-NN+LSTM, and GG-NN+CNN, and found the GG-NN+GP model have the best overall performance. We also tested out 2 different techniques, random oversampling and weight balancing, to improve the performance of the model trained on an imbalanced dataset and found neither to have a better overall performance compared to the GG-NN+GP model. A pilot study was conducted to test the recommendation system's performance in terms of accuracy and time taken to identify which functions within a chosen GitHub project should be placed inside a TEE, and discovered that the group using the recommendation system has a higher average accuracy and a shorter average time taken to identify functions that need to be placed inside a TEE. In addition, we also analyzed what the machine

learning model used by the recommendation system considers to be important in formulating its output and discovered that function names, parameter names, variable names, and the names of the functions invoked within the function were all important.

8.2 Future Work

One direction for future work is to conduct a formal study involving more participants to draw a more conclusive result because in this thesis, we only conducted a pilot involving 6 participants. In this thesis, we focused on applications that used Intel SGX, for future work, this can be expanded to other types of TEE implementations, such as OP-TEE [1]. Another direction for future work would be to expand the recommendation system to include other programming languages, such as Python or JavaScript, because many applications that use Intel SGX do not code their entire application in C/C++.

Bibliography

[1] Open portable trusted execution environment. https://www.op-tee.org/.

[2] Model selection. URL https://ethen8181.github.io/machine-learning/model_selection/model_selection.html.

[3] Intel sgx product brief, 2019. https://software.intel.com/content/dam/develop/public/us/en/documents/intel-sgx-product-brief-2019.pdf.

[4] Souss-Massa Agadir. Backpropagation, and what's doing. 2020. https://www.kaggle.com/learn-forum/187430.

[5] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740*, 2017.

[6] Miltos Allamanis. Msr cambridge lecture series: An introduction to graph neural networks: Models and applications. URL https://www.microsoft.com/en-us/research/video/msr-cambridge-lecture-series-an-introduction-to-graph-neural-networks-models-and-ap

[7] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[8] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146, 2017.

[9] Jason Brownlee. A gentle introduction to imbalanced classification. 2020. https://machinelearningmastery.com/what-is-imbalanced-classification/.

[10] Jason Brownlee. Random oversampling and undersampling for imbalanced classification, 2020. https://machinelearningmastery.com/random-oversampling-and-undersampling-for-imbalanced-classification/.

[11] Timo Böhm. The general ideas of word embeddings. 2018. https://towardsdatascience.com/the-three-main-branches-of-word-embeddings-7b90fa36dfb9.

[12] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[13] Boris Chernis and Rakesh Verma. Machine learning methods for software vulnerability detection. In *Proceedings of the Fourth ACM International Workshop on Security and Privacy Analytics*, pages 31–39, 2018.

[14] Hoa Khanh Dam, Truyen Tran, Trang Pham, Shien Wee Ng, John Grundy, and Aditya Ghose. Automatic feature learning for vulnerability prediction. *arXiv preprint arXiv:1708.02368*, 2017.

[15] Brett Daniel. What is intel sgx? 2021. https://www.trentonsystems.com/blog/what-is-intel-sgx.

[16] Tim Dettmers. Deep learning in a nutshell: Core concepts. 2015. https://developer.nvidia.com/blog/deep-learning-nutshell-core-concepts/.

[17] Siddharth Dhar. Optimizing tee protection by automatically augmenting requirements specifications. Master's thesis, Virginia Tech, 2020.

[18] Hantao Feng, Xiaotong Fu, Hongyu Sun, He Wang, and Yuqing Zhang. Efficient vulnerability detection based on abstract syntax tree and deep learning. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 722–727. IEEE, 2020.

[19] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[20] Hariom Gautam. Word embedding: Basics. 2020. https://medium.com/@hari4om/word-embedding-d816f643140.

[21] Anna Glazkova. A comparison of synthetic oversampling methods for multi-class text classification. *arXiv preprint arXiv:2008.04636*, 2020.

[22] Sally Goldman. Embeddings. 2020. https://developers.google.com/machine-learning/crash-course/embeddings/video-lecture.

[23] Aakriti Gupta and Shreta Sharma. Software maintenance: Challenges and issues. *Issues*, 1(1):23–25, 2015.

[24] Jacob A Harer, Louis Y Kim, Rebecca L Russell, Onur Ozdemir, Leonard R Kosta, Akshay Rangamani, Lei H Hamilton, Gabriel I Centeno, Jonathan R Key, Paul M Ellingwood, et al. Automated software vulnerability detection with machine learning. *arXiv preprint arXiv:1803.04497*, 2018.

[25] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pages 200–20010. IEEE, 2018.

[26] Bob Hunt, Bryn Turner, and Karen McRitchie. Software maintenance implications on cost and schedule. In *2008 IEEE Aerospace Conference*, pages 1–6. IEEE, 2008.

[27] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code. In *International Conference on Machine Learning*, pages 5110–5121. PMLR, 2020.

[28] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. Big code!= big vocabulary: Open-vocabulary models for source code. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 1073–1085. IEEE, 2020.

[29] Nishtha Kesswani and Sanjay Kumar. Maintaining cyber security: Implications, cost and returns. In *Proceedings of the 2015 ACM SIGMIS Conference on Computers and People Research*, pages 161–164, 2015.

[30] Huang Kung-Hsiang. Hands-on graph neural networks with pytorch & pytorch geometric. 2019. https://towardsdatascience.com/hands-on-graph-neural-networks-with-pytorch-pytorch-geometric-359487e221a8.

[31] Titouan Lazard, Johannes Götzfried, Tilo Müller, Gianni Santinelli, and Vincent Lefebvre. Teeshift: Protecting code confidentiality by selectively shifting functions into tees. SysTEX '18, page 14–19, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450359986. doi: 10.1145/3268935.3268938. URL https://doi-org.ezproxy.lib.vt.edu/10.1145/3268935.3268938.

[32] Leland Lee. Adventures of an enclave (sgx / tees), 2018. https://hackernoon.com/adventures-of-an-enclave-sgx-tees-9e7f8a975b0b.

[33] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

[34] Guanjun Lin, Sheng Wen, Qing-Long Han, Jun Zhang, and Yang Xiang. Software vulnerability detection using deep neural networks: a survey. *Proceedings of the IEEE*, 108(10):1825–1848, 2020.

[35] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, Christof Fetzer, and Peter Pietzuch. Glamdring: Automatic application partitioning for intel SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 285–298, Santa Clara, CA, July 2017. USENIX Association. ISBN 978-1-931971-38-6. URL https://www.usenix.org/conference/atc17/technical-sessions/presentation/lind.

[36] Y. Liu and E. Tilevich. Reducing the price of protection: Identifying and migrating non-sensitive code in tee. In *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 112–120, 2020. doi: 10.1109/TrustCom50675.2020.00028.

[37] Yin Liu, Kijin An, and Eli Tilevich. Rt-trust: Automated refactoring for trusted execution under real-time constraints. In *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2018, page 175–187, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450360456. doi: 10.1145/3278122.3278137. URL https://doi.org/10.1145/3278122.3278137.

[38] Matt Mazur. A step by step backpropagation example. 2015. https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/.

[39] John P. Mechalas. Intel® software guard extensions part 4: Design an enclave. 2016. https://software.intel.com/content/www/us/en/develop/articles/software-guard-extensions-tutorial-series-part-4.html.

[40] John P. Mechalas. Intel® software guard extensions part 7: Refine the enclave with proxy functions. 2016. https://software.intel.com/content/www/us/en/develop/articles/intel-software-guard-extensions-tutorial-part-7-refining-the-enclave.html.

[41] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[42] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[43] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[44] Sergio Prado. Introduction to trusted execution environ-

ment and arm's trustzone. 2021. https://embeddedbits.org/introduction-to-trusted-execution-environment-tee-arm-trustzone/.

[45] Radim Řehůřek and Petr Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta, May 2010. ELRA. http://is.muni.cz/publication/884893/en.

[46] Yongchang Ren, Tao Xing, Xiaoji Chen, and Xuguang Chai. Research on software maintenance cost of influence factor analysis and estimation method. In *2011 3rd International Workshop on Intelligent Systems and Applications*, pages 1–4. IEEE, 2011.

[47] Konstantin rubinov, Lucia Rosculete, Tulika Mitra, and Abhik Roychoudhury. Automated partitioning of android applications for trusted execution environments. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 923–934, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450339001. doi: 10.1145/2884781.2884817. URL https://doi.org/10.1145/2884781.2884817.

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

[49] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. Trusted execution environment: what it is, and what it is not. In *2015 IEEE Trustcom/BigDataSE/ISPA*, volume 1, pages 57–64. IEEE, 2015.

[50] Fan Sang. Enclave - sgx 101. 2019. https://sgx101.gitbook.io/sgx101/sgx-bootstrap/enclave.

[51] George Seif. Handling imbalanced datasets in deep learning, 2018. https://towardsdatascience.com/handling-imbalanced-datasets-in-deep-learning-f48407a0e758.

[52] Tim Sonnekalb. Machine-learning supported vulnerability detection in source code. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1180–1183, 2019.

[53] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1433–1443, 2020.

[54] M. Ye, J. Sherman, W. Srisa-an, and S. Wei. Tzslicer: Security-aware dynamic program slicing for hardware isolation. In *2018 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 17–24, 2018. doi: 10.1109/HST.2018.8383886.

[55] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *arXiv preprint arXiv:1909.03496*, 2019.