DePaul University

Final Report

Software Vulnerability Detection with Neural Networks

Amy Aumpansub

Professor Zhen Huang

May 10th , 2020

Table of Contents

Abstract

The neural networks have been widely used in image processing and speech recognition as it can provide the high accuracy rate of prediction while insignificantly relying on human experts in feature extraction. As software vulnerabilities can be caused by a wide range of reasons, the neural networks can be served as a useful tool in detecting vulnerabilities. Unlike pattern-based methods, the neural network eliminates influence of human bias in feature extractions. A main focus of this study was to develop the strong predictive model using Neural Networks to detect the vulnerability of C/C++ programs automatically. The sample programs were originally collected from the National Vulnerability Database (NVD) and Software Assurance Reference Dataset (SARD). They include both vulnerability and non-vulnerability programs, divided into 4 different types as follows: API Function Call (API), Array Usage (AU), Pointer Usage (PU), and Arithmetic Expression (AE). The programs were preprocessed and transformed to vectors to feed the neural networks. This study found that the model built from a combined dataset which included all vulnerability types outperformed the models fitted with dataset of separated vulnerability types as there was no clear line to differentiate among vulnerability types. To develop a strong predictive model to detect the vulnerabilities in program, the neural networks were trained with different datasets, hyperparameters such as optimizer, gating mechanism, and activation functions in each phase of model development. The various models were developed from phase I to phase IV which in the first phase, four models were built with the datasets separated by the vulnerability types with tanh function for hidden layers and sigmoid function applied to the output layer. For the second phase, the models were fitted with the same datasets with phase I, but the activation functions including RELU and SIGMOID, as well as optimizers such as SGD, ADAM, and ADAMAX were utilized and compared to improve models. From Phase II, the ADAM optimizer was found to outperform ADAMAX optimizer. In Phase III, the models were built from the combined dataset which included all vulnerability types. The models from combined types were found to perform better than the models fitted with dataset of separated types. In Phase IV, the Bidirectional Recurrent Neural Networks were found to be more effective than regular Recurrent Neural Networks. Holding other hyperparameters constan, BGRU can converge faster and have lower loss rates and have higher accuracy rates than those of BLSTM models. The BGRU model's accuracy rate is as high as 94.6% for a training set and 92.4% for a test set. In the last step, the full dataset of 420, 067 program slices were spitted and trained. The model has the highest accuracy rate of 94.89%. For model validation, the model was evaluated with a test set and showed that it performs well with a high balanced accuracy rate of 93% which is comparable to that of a training set. The high sensitivity and specificity imply a good ability of model in explaining both vulnerability and non-vulnerability classes.

Dataset

The original dataset contains 15,592 programs wrote in C/C++ languages, retrieved from https://github.com/SySeVR. The dataset contains 1,592 programs collected from the National Vulnerability Database (NVD) and 14,000 programs collected from Software Assurance Reference Dataset (SARD). The original sample programs were pre-processed and transformed to 420,627 slices called semantic vulnerability candidates (SeVC) which include:

- 56,395 vulnerable slices (13.5 % of total slices)
- 364,232 non-vulnerable slices (86.5 % of total slices)

According to Li et al. (2018), slices of programs were divided into 4 main types regarding the vulnerability syntax analysis obtained from Checkmarx as follows:

1. Library or API Function Call

   This vulnerability type is associated with library or API functions calls in program which contain 811 C/C++ library/API function calls. This type represents 15.3% of total sample slices, comprising 13,603 vulnerable slices and 50,800 non-vulnerable slices.

2. Array Usage

   This vulnerability is related to the use of arrays such as improper uses of array element access, array, and arithmetic, accounting for 10% of total slices which contains 10,926 vulnerable slices and 31,303 non-vulnerable slices.

3. Pointer Usage

   This vulnerability type is correlated to inappropriate uses of pointer arithmetic and references which are a main type of sample slices comprising 69.4% of total slices which includes 28,391 vulnerable slices and 263,450 non-vulnerable slices.

4. Arithmetic Expression

   This vulnerability type is related to improper arithmetic expressions such as integer overflow which represents 5.3% of total slices, comprising 3,475 vulnerable slices and 18,679 non-vulnerable slices.

Dependent Variable:

A binary variable (0 is vulnerable slice and 1 is non-vulnerable slice)

Independent Variables:

Array of vectors transformed from tokens in each program slice using Word2Vector Model.
The details were discussed in the preprocessing section.

Methodology

Deep learning algorithms were utilized to solve this supervised learning problem to predict target (binary) attribute whether the program slice is vulnerable or not. The models were trained using Bidirectional Long Short-Term Memory (BLSTM) and Bidirectional Gated Recurrent Unit (BGRU). Several Python packages include Pandas, Matplotlib, NumPy, Sklearn, Genism, Keras, and Tensorflow were utilized in this study. The following steps were implemented for this project:

- Extracted Syntax-based Vulnerability Candidates (SyVCs) from C/C++ programs by creating a Program Dependency Graph for each function
- Generated program slices from SyVCs for both forward and backward slices
- Extracted program slices to Semantics-based Vulnerability Candidates (SeVCs)
- Transformed symbolic representations (tokens) into vectors using Word2Vector Model
- Adjusted the array of vectors of each slice with the mean vector lengths of all slices
- Split data into a training set (80%) for fitting model and a test set (20%) for evaluation
- Downsampled the program slices in training set to ensure the target classes were balanced
- Trained and developed several neural networks with different set of vector inputs and hyperparameters to compare the models and activations outputs
- Performed model evaluation with a test set using several metrics
- Analyzed activation outputs from neural networks to observe distributions, patterns, and outputs with various thresholds for prediction

Data Preprocessing

To preprocess source codes, the whole process was divided into 3 parts. The first part is to generate semantic vulnerability candidates (program slices) from source codes as shown in figure 1, implemented by Li et al. (2018). The second part is to transform slices to the array of vectors to feed the neural networks (figure2.1) which was reimplemented this study. The last part is to split data into training and test sets and resampling data to ensure the training set has balanced target classes (figure3.2).

Part 1

- The source codes were parsed using Joern package in Python
- Created CFG graphs from outputs from parser and used them to create PDG graphs
- Saved the SyVCs as separated files regarding the four vulnerability types
- Extracted program slices from SyVCs
- The process in details can be found from SySeVR paper
- The slicing outputs generated from part 1 were further preprocessed in Part 2.

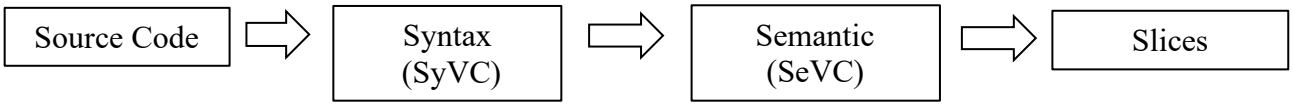


Figure 1: Processing Source Codes

Part 2

- Each slice was transformed to an array of tokens in which all comments and whitespaces were also removed before transformation and tokens were mapped with list of function names (Figure 2.1).
- For each slice, the tokenized outputs were stored in pickle file and labeled with the unique sample ID (no duplicated ID).
- Each pickle file contains an array of 5 elements including a list of tokens, a target label (0/1), list of functions, vulnerability type, and sample ID with a main function name.
- The tokens from each pickle file were converted to vectors using Word2Vector model built from Gensim package.
- The Word2Vector model converted tokens to vector based on cosine similarity distance which measures the angle between vectors in which the high similarity score indicates high similarity and a closer distance between tokens (Mikolov et al., 2013). The visualization of words in W2V model is shown in figure 2.2. The cosine similarity is computed as follows:

$$sim(X,Y) = \frac{X \bullet Y}{\|X\| \times \|y\|} = \frac{\sum_i (x_i \times y_i)}{\sqrt{\sum_i x_i^2} \times \sqrt{\sum_i y_i^2}}$$

- The Word2Vector model was trained with all vectors with input parameters as follows: min-size = 0 for frequency count and size = 30 (number of vector dimensions that Gensim Word2Vec maps the words onto). More details about inputs can be found inside createW2VModel function in tokensToVectors.py.
- Then, the tokens from each slices file were converted using the trained Word2Vector model. The output vector for each program slice is an array of vectors with a dimension of 30 columns and different number of rows depending on the number of tokens in each slice.
- The vector output from Word2Vector model for each tokenized program was stored into a separate pickle file contains an array of 5 elements: an array of vectors, a target label (0/1), list of functions in slice, vulnerability type, and sample ID with a function name.

Figure 2.1: Process of Tokenization and Vectorization

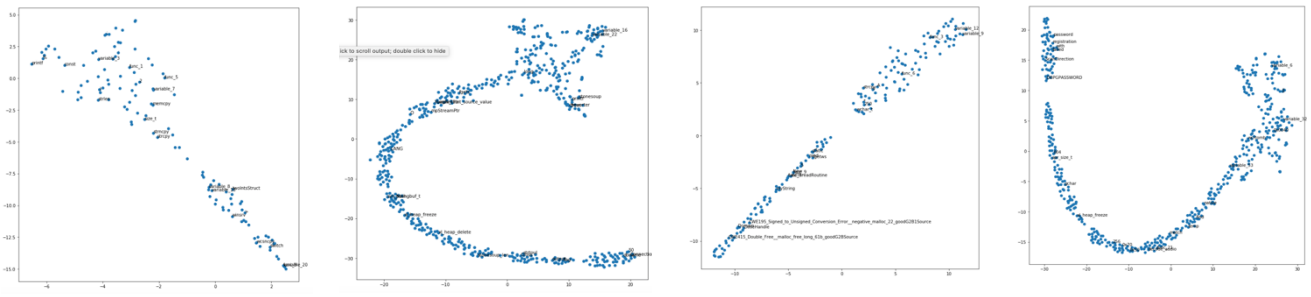


Figure 2.2: Visualized Words in W2V model for Each Vulnerability Type

Part 3

- The vector arrays of program slices were spitted into a training set (80%) for fitting model and a test set (20%) for model evaluation using a random seed.
- The original training set had an imbalanced class of target label in which the class label 1 (Vulnerability) is a minority in dataset. Class 1 only accounts for 15.6% of total program slices. Thus, the model fitted with imbalanced class samples without adjustment may lead to inaccurate evaluation and reduce ability to predict class label 1 which is a minority class and is a major focus in this study.
- The confusion matrix (figure3.1) shows the model evaluation of the model fitted with imbalanced class sample (75% of class 0 and 25% of class 1). The model has more ability to predict class 0 as its specificity and negative prediction are both higher than its sensitivity and precision. Thus, the accuracy rate does not represent the true performance of model as it does not take an imbalanced class problem into account.

Predicted Class

	Positive	Negative	Rate	
Positive	8.0	48.0	0.1428571492433548	Sensitivity
Negative	14.0	130.0	0.9027777910232544	specificity
	0.3636363744735718	0.7303370833396912	0.6899999976158142	Accuracy
	Precision	NegPrediction		

Figure 3.1: Confusion Matrix of an Imbalanced-class Samples

- In order to solve an imbalanced class issue, a training set was resampled using down-sampling method to randomly extract samples from a majority class (label 0) from a training set. The new sample set has a balanced class label, comprising 50% Vulnerability vector arrays and 50% Non-Vulnerability vector arrays.
- For the last step in figure 4, all vector arrays for each slice program needed to be adjusted to have a same length (same number of rows) as each array of vectors have different number of rows, but the neural networks required all vector input to have a same dimension (same number of columns and same number of rows).
- The mean of vector lengths for all slice programs was calculated and use as the threshold to adjust the vector lengths. If an array of vectors has a shorter length than the mean, it will embed with the zero array. For the array of vectors with longer length than the mean length, the vectors were truncated. The algorithms used to adjust the vectors lengths can be found in adjustVectorLen.py.

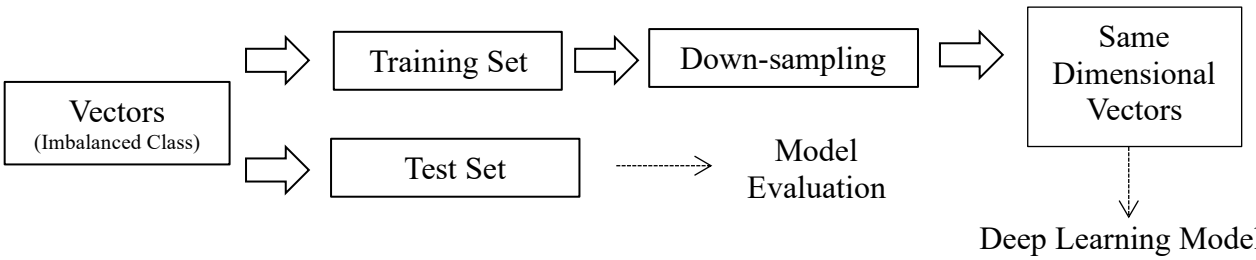


Figure 3.2: Downsampling and Vector Adjustment

Model Development

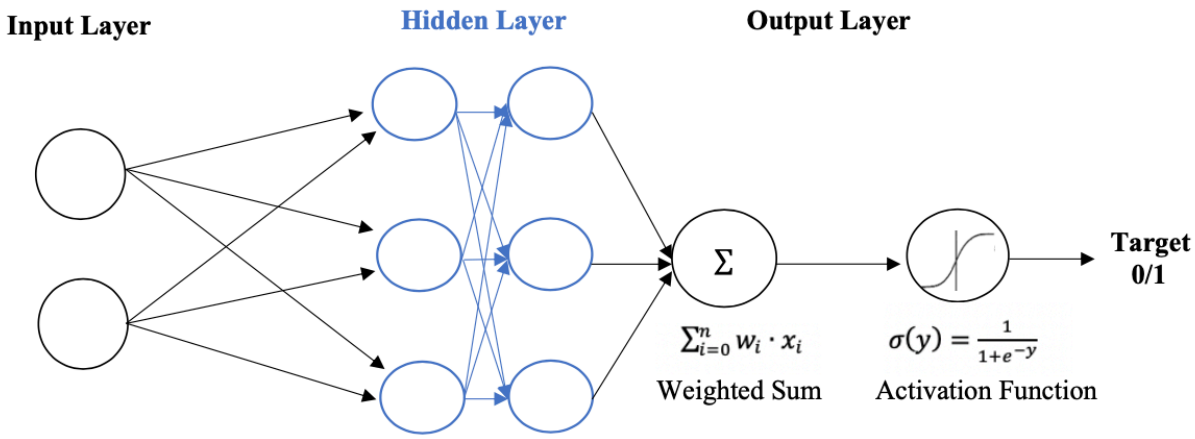
Phase I

The 30,000 vector arrays in which each array represents tokens in one program slice were randomly spitted into a training set and a test set as discussed in the previous section. The four models in the first phase were fit separately with a training set for each vulnerability type: API Function Call (API), Array Usage (AU), Pointer Usage (PU), Arithmetic Expression (AE). Additionally, the fifth model was trained with a training set of a combined vulnerability type.

The architecture of Neural Network for this phase was set as the same inputs from SySe paper. The input parameters of sequential model from Keras package including Bidirectional gated recurrent unit (BGRU) of 256 neuron units with 2 hidden layers. The learning rate is 0.01 with batch size of 32, vector inputs with same shape [mean vector lengths x 30] fitted for 10 epochs. The binary cross-entropy loss is selected as loss function as it can speed up the learning process and convergence. The optimizer is "Adamax" (Appendix A).

Figure 5.1: Model and Network Architecture (Phase I)

```
Build model...
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
masking_1 (Masking)          (None, 127, 30)           0
_________________________________________________________________
bidirectional_1 (Bidirection (None, 127, 512)          440832
_________________________________________________________________
dropout_1 (Dropout)          (None, 127, 512)          0
_________________________________________________________________
bidirectional_2 (Bidirection (None, 512)               1181184
_________________________________________________________________
dropout_2 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 513
=================================================================
```



The figure of network with sigmoid threshold unit

Activation Functions:

The "Tanh" function is utilized to compute output which is the weighted sum of input and biases for the hidden layers (Calin, 2020):

$$o = tanh(\sum_{i=0}^{n} w_i \cdot x_i)$$

$$\text{where } tanh(y) = \frac{e^y - e^{-y}}{e^y + e^{-y}}$$

The weight update formula: $\Delta w_i = \Delta w_i + n(t_d - o_d)\,(1 - {o_d}^2)\; x_i$ where *n* is learning rate

The "Sigmoid" function is appropriate to predict target labels of binary classification in this study. It was implemented in the last activation layer to compute the outputs as follows (Mitchell, 1997):

$$o = \sigma(\quad \sum_{i=0}^{n} w_i \cdot x_i \quad)$$

$$\text{where } \sigma(y) = \frac{1}{1+e^{-y}}$$

The weight update formula: $\Delta w_i = \Delta w_i + n(t_d - o_d) o_d\,(1 - o_d)\, x_i$ where *n* is learning rate

Model Fitting (Phase I)

The four models in the first phase were fit separately with a training set for each vulnerability type: API Function Call (API), Array Usage (AU), Pointer Usage (PU), Arithmetic Expression (AE). The accuracy rates for all types are shown below. Among 4 types: Arithmetic Expression model has the highest accuracy rate, however, the rate for test set is significantly less than that of the training set, implying the overfitting program from AE model as errors for training data are more minimized than test set. On the other hands, models from API and AU has comparable accuracy rates for training and test set which implies a less effect of overfitting network.

The model provides the activation outputs range from 0 to 1 because it is computed using sigmoid function which is continuous and differentiable function. The prediction threshold of 0.5 is used to decide whether each output falls into class 1 or class 0 where any outputs which greater than 0.5 are considered to be class 1 (vulnerability), vice versa

	Accuracy Rate	
	Train	**Test**
API	89.8%	86.7%
AU	88.1%	86.0%
PU	87.2%	82.4%
AE	90.3%	83.1%

Figure 5.2: Accuracy Rates (Phase I)

Model Validation (Phase I)

Models were evaluated with a test set, comprising 6000 arrays of vectors (each array represent one program slice). The confusions matrix (figure 5.3) provides more details for model performance and validation. The different decision threshold has been used to construct the confusion matrix to explore a sensitivity of model performance across different thresholds (figure 5.4). The F1 scores were computed as weighted average of Precision and Recall which takes both false positives and false negatives into account. The balanced accuracy rates were calculated as the average of the proportion corrects of each class individually to provide unbiased accuracy rate regardless the more frequent class in a test set (Brodersen et al., 2010).

API

Predicted Class

	Positive	Negative	Rate	
Positive	1186.0	163.0	0.879169762134552	Sensitivity
Negative	632.0	4018.0	0.8640860319137573	specificity
	0.6523652076721191	0.9610140919685364	0.8674778938293457	Accuracy
	Precision	NegPrediction		

AU

Predicted Class

	Positive	Negative	Rate	
Positive	1790.0	158.0	0.918891191482544	Sensitivity
Negative	680.0	3371.0	0.8321402072906494	specificity
	0.7246963381767273	0.9552280902862549	0.8603100776672363	Accuracy
	Precision	NegPrediction		

PU

Predicted Class

	Positive	Negative	Rate	
Positive	1170.0	216.0	0.8441558480262756	Sensitivity
Negative	838.0	3775.0	0.818339467048645	specificity
	0.5826693177223206	0.9458782076835632	0.8243040442466736	Accuracy
	Precision	NegPrediction		

AE

Predicted Class

	Positive	Negative	Rate	
Positive	641.0	50.0	0.9276410937309265	Sensitivity
Negative	697.0	3042.0	0.8135865330696106	specificity
	0.4790732562541961 7	0.9838292598724365	0.8313769698143005	Accuracy
	Precision	NegPrediction		

Figure 5.3: Confusion Matrix (Phase I)

API Function Call (API)

Predicted Class

thresdArray	recall	precision	specificity	F1	Accuracy	balanceAccuracy
0.4	0.906597	0.617677	0.837204	0.734755	0.852809	0.871901
0.45	0.889548	0.630252	0.848602	0.737781	0.85781	0.869075
0.5	0.87917	0.652365	0.864086	0.748974	0.867478	0.871628
0.53	0.873981	0.663105	0.871183	0.754077	0.871812	0.872582
0.55	0.870274	0.668184	0.874624	0.755956	0.873646	0.872449
0.58	0.862861	0.674001	0.878925	0.756827	0.875313	0.870893
0.6	0.858414	0.678781	0.882151	0.758101	0.876813	0.870282
0.65	0.841364	0.690809	0.890753	0.75869	0.879647	0.866058
0.7	0.811712	0.702824	0.90043	0.753354	0.88048	0.856071
0.8	0.754633	0.749632	0.926882	0.752124	0.888148	0.840757

Figure 5.4: API Metrics across Prediction Thresholds

Threshold 0.53 is better than 0.5 which has higher F1(75%) and higher balanced accuracy (87%) as shown in table The peak points of F1 is at 0.65 and balanced accuracy is at 0.53. F1 is utilized to blend precision and specificity for model evaluation. Balanced accuracy is more accurate as accuracy is biased with the increasing threshold.

Array Usage (AU)

Predicted Class

thresdArray	recall	precision	specificity	F1	Accuracy	balanceAccuracy
0.4	0.946099	0.704511	0.809183	0.807625	0.853642	0.877641
0.45	0.931725	0.713724	0.820291	0.808283	0.856476	0.876008
0.5	0.918891	0.724696	0.83214	0.810321	0.86031	0.875516
0.53	0.908624	0.732616	0.840533	0.811182	0.862644	0.874579
0.55	0.904517	0.737238	0.844977	0.812356	0.864311	0.874747
0.58	0.88655	0.740566	0.850654	0.807009	0.86231	0.868602
0.6	0.86961	0.745599	0.857319	0.802844	0.86131	0.863465
0.65	0.826489	0.770704	0.881758	0.797622	0.863811	0.854123
0.7	0.766427	0.800966	0.908418	0.783316	0.86231	0.837422
0.8	0.649897	0.868909	0.952851	0.743612	0.854476	0.801374

Figure 5.5: AU Metrics across Prediction Thresholds

Threshold 0.55 performs better than 0.5 which has higher F1(81%) and higher balanced accuracy (87.4%) as shown in table The peak points of F1 is at 0.58 and balanced accuracy is at 0.55.
F1 and BA keeps decreasing as the threshold increases

Pointer Usage (PU)

Predicted Class

thresdArray	recall	precision	specificity	F1	Accuracy	balanceAccuracy
0.4	0.885281	0.556715	0.788207	0.683565	0.810635	0.836744
0.45	0.872294	0.568139	0.80078	0.688105	0.817303	0.836537
0.5	0.844156	0.582669	0.818339	0.689452	0.824304	0.831248
0.53	0.832612	0.594539	0.829395	0.693718	0.830138	0.831004
0.55	0.822511	0.599054	0.834598	0.69322	0.831805	0.828554
0.58	0.807359	0.608483	0.843919	0.693953	0.835473	0.825639
0.6	0.798701	0.612618	0.848255	0.693392	0.836806	0.823478
0.65	0.771284	0.624051	0.860395	0.6899	0.839807	0.815839
0.7	0.731602	0.641366	0.877086	0.683519	0.843474	0.804344
0.8	0.640693	0.688372	0.912855	0.663677	0.849975	0.776774

Figure 5.6: AU Metrics across Prediction Thresholds

Threshold 0.58 performs better than 0.5 which has higher F1(69%) and higher balanced accuracy (82.5%) as shown in table The peak points of F1 is at 0.65 and balanced accuracy is at 0.55.The BA keeps decreasing as the threshold increases beyond 0.55.

Arithmetic Expression (AE)

Predicted Class

thresdArray	recall	precision	specificity	F1	Accuracy	balanceAccuracy
0.4	0.936324	0.444979	0.784167	0.603263	0.807901	0.860246
0.45	0.930535	0.46259	0.800214	0.617972	0.820542	0.865375
0.5	0.927641	0.479073	0.813587	0.631838	0.831377	0.870614
0.53	0.921852	0.48626	0.820005	0.636682	0.835892	0.870929
0.55	0.920405	0.496487	0.827494	0.64503	0.841986	0.87395
0.58	0.917511	0.510056	0.837122	0.655636	0.849661	0.877317
0.6	0.914616	0.514239	0.840332	0.658333	0.851919	0.877474
0.65	0.901592	0.53339	0.854239	0.670253	0.861625	0.877915
0.7	0.885673	0.548387	0.865205	0.677366	0.868397	0.875439
0.8	0.81042	0.605405	0.90238	0.693069	0.888036	0.8564

Figure 5.7: AE Metrics across Prediction Thresholds

Threshold 0.65 performs better than 0.5 which has higher F1(67%) and higher balanced accuracy (87.7%) as shown in table The peak points of F1 is at 0.7 and balanced accuracy is at 0.65. The BA keeps decreasing as the threshold increases beyond 0.65.

Model Development

Phase II

For the second phase, the improvement of neural networks is a main focus, so the architecture of neural networks such as optimizer and model units of sequential model were adjusted. Some inputs remain the same as Phase I but the research was further conducted to examine whether the selected parameters are appropriate for our network and study (Appendix B).

Loss Function

- Loss Function: The binary cross-entropy loss (BCE) is still applied in this phase as it fit our dataset which has binary target class 0 or 1. Each output unit from neural network model is a separate random binary variable, so total loss for the entire outputs in each step is computed as product of the loss of single binary variables.
- Cross-entropy loss is computed as follows (Nielsen, 2019):

$$C = -\frac{1}{n} \sum_{x} [y \ln a + (1 - y) \ln(1 - a)]$$

- Comparing to Mean Squared Error, Cross-entropy loss can speed up the learning process of neuron network as process will converge faster than Mean Squared Error which is based on a quadratic function. Cross entropy with sigmoid activation is recommended to use in a classification problem (Dufourq el al., 2017). It is found to speed up the learning process and its curve decreases faster in the earlier epochs.

Activation Function

- There are various activation functions that can be used for computer neural network outputs such as Linear, Gaussian, Sigmoid, Tanh, Softmax, and Relu activation functions.
- The activation function for the last layer (output layer) produces the activation outputs from the neural network.
- Softmax activation is usually applied with Loglikelihood loss function and appropriate for multiclass classification and applied only to the output layer (the last layer).
- Sigmoid activation is continuous and differentiable based on logistic function. Outputs range between 0 to 1, implying probability of each binary class. Thus, it is proper for our binary classification problem with a binary target class (0 non-vulnerability and 1 vulnerability). Sigmoid is commonly used as it does not destroy the activation, but it may vanish gradient.
- Relu activation can compute outputs faster than sigmoid function and have benefits of sparsity and a reduced likelihood of vanishing gradient. However, ReLu can create Dying Relu problem for negative inputs (Castaneda, 2019).

- When the ReLu activation is applied to compute the outputs, it creates dead neurons that never get activated (Dying ReLu Problem). ReLu neurons output zero and have zero derivatives for all negative inputs. In our case, outputs from previous layers are negative and later served as negative inputs into a ReLu neuron, so ReLu ends up in the dying state. as shown on figure 6.1.

```
start
Epoch 1/10
17/17 [==============================] - 17s 983ms/step - loss: 7.1564 - accuracy: 0.0349
Epoch 2/10
17/17 [==============================] - 14s 820ms/step - loss: 7.6384 - accuracy: 0.0000e+00
Epoch 3/10
17/17 [==============================] - 14s 834ms/step - loss: 7.6384 - accuracy: 0.0000e+00
Epoch 4/10
17/17 [==============================] - 15s 900ms/step - loss: 7.6384 - accuracy: 0.0000e+00
Epoch 5/10
17/17 [==============================] - 14s 819ms/step - loss: 7.6384 - accuracy: 0.0000e+00
Epoch 6/10
17/17 [==============================] - 14s 849ms/step - loss: 7.6384 - accuracy: 0.0000e+00
Epoch 7/10
17/17 [==============================] - 13s 786ms/step - loss: 7.6384 - accuracy: 0.0000e+00
Epoch 8/10
17/17 [==============================] - 13s 789ms/step - loss: 7.6384 - accuracy: 0.0000e+00
Epoch 9/10
17/17 [==============================] - 14s 813ms/step - loss: 7.6384 - accuracy: 0.0000e+00
Epoch 10/10
17/17 [==============================] - 13s 763ms/step - loss: 7.6384 - accuracy: 0.0000e+00
```

Figure 6.1: Model fitted with ReLu

Optimizer

- There are several optimizers that can be applied to optimize neural network such as Stochastic Gradient Descent (SGD) by default, Adamax (SySe Paper), RMS Prop, and Adam.
- Adam optimizer is a combination of RMSprop and SGD with momentum, which is an adaptive learning rate method and computationally efficient as it computes individual learning rates for different parameters.
- According to Kingma et al. (2015), ADAM method is appropriate for problems with large dataset and/or parameters, with non-stationary objectives, and for problems with very noisy and/or sparse gradients. Given the nature of software vulnerabilities which contain various causes, the ADAM method is an appropriate method to further examine and apply towards the model development in this study.

- ADAM optimizer will improve our neural network when a network has weak signals which is not sufficient to tune its weights effectively.

Model Fitting (Phase II)

The architecture of neural networks such as optimizer and model units of sequential model were adjusted to minimize its error rate. The four models in the second phase were fit separately with a training set for each vulnerability type: API Function Call (API), Array Usage (AU), Pointer Usage (PU), Arithmetic Expression (AE). The accuracy rates for all types are shown below.

The models in Phase II were trained with Binary Cross entropy function with tanh activation for hidden layers and sigmoid activation for the last (output) layer. Holding other parameters constant as Phase I, the only change from Phase I includes the use of different optimizers: ADAMAX, SGD, ADAM in order to find the best optimizer to develop our neural networks for each vulnerability type. We begin with the same training set and test set as Phase I. The summary of models with ADAMAX and SGD optimizer is presented in figure 6.2 in the following page.

	Accuracy Rate	
	ADAMAX	**SGD**
API	86.7%	63.1%
AU	86.0%	58.6%
PU	82.4%	62.3%
AE	83.1%	67.1%

Figure 6.2: Accuracy Rates of Models Fitted with ADAMAX vs SGD

All models with ADAMAX optimizers perform better than those fitted with Stochastic gradient descent (SGD) optimizer, so the SGD optimizer did not improve the models from Phase I. Thus, in order to choose the best optimizer for our dataset, the new model was trained with another optimizer, "ADAM."

Predicted Class

	Positive	Negative	Rate	
Positive	1186.0	163.0	0.879169762134552	Sensitivity
Negative	632.0	4018.0	0.8640860319137573	specificity
	0.6523652076721191	0.9610140919685364	0.8674778938293457	Accuracy
	Precision	NegPrediction		

Figure 6.3: Confusion Matrix with ADAMAX (Phase I)

Predicted Class

	Positive	Negative	Rate	
Positive	1204.0	145.0	0.8925129771232605	Sensitivity
Negative	484.0	4166.0	0.8959139585494995	specificity
	0.7132701277732849	0.9663650989532471	0.8951491713523865	Accuracy
	Precision	NegPrediction		

Figure 6.4: Confusion Matrix with ADAM (Phase II)

From the confusion matrices above, the model with ADAM optimizer in Phase II performs better than the model with ADAMAX from Phase I. The ADAM Optimizer increases accuracy rates by 3% and precision rates by 6% on test set. From Phase II, we used binary cross-entropy loss (BCE) with sigmoid function for the Last Activation Layer (output between 0 and 1). We changed the optimizer and refitted the models.

We further improved the models by training networks with all program slices for each vulnerability types with ADAM optimizer. API, AU, PU, and AE have 64,403, 42,229, 130,000, and 22,154 samples slices, respectively. The accuracy rates from the models are shown in the table below.

	Accuracy Rate	
	Train	**Test**
API	93.6%	89.5%
AU	91.7%	89.2%
PU	94.9%	90.9%
AE	92.5%	90.5%

Figure 6.5: Accuracy Rates of Models with ADAM Optimizer (Phase II)

Among four vulnerability type, the pointer usage model performs the best as it has the highest accuracy rates of 94.9% for training set and 90.9% for test set. The API model has the second highest accuracy rates of 93.6% for a training set and 89.5% for a test set. All models have a high accuracy rates which indicate a good performance in prediction. The validation for these models is discussed below.

Model Validation (Phase II)

Models were evaluated with a test set. The confusions matrix (figure 6.6) provides more details for model performance and validation. The decision threshold is set to 0.5 for constructing the confusion matrix. All four models have a high sensitivity which indicates the strong ability of model to learn from class 1 (vulnerability sample). All accuracy rates are also high as the rates from training set which displays a strong predictive power of all four models on the new data.

API

Predicted Class

	Positive	Negative	Rate	
Positive	2528.0	209.0	0.9236389994621277	Sensitivity
Negative	1141.0	9002.0	0.8875086307525635	specificity
	0.6890161037445068	0.9773097634315491	0.8951863646507263	Accuracy
	Precision	NegPrediction		

AU

Predicted Class

	Positive	Negative	Rate	
Positive	2113.0	149.0	0.9341291189193726	Sensitivity
Negative	763.0	5420.0	0.8765971064567566	specificity
	0.7347009778022766	0.9732447266578674	0.8920071125030518	Accuracy
	Precision	NegPrediction		

PU

Predicted Class

	Positive	Negative	Rate	
Positive	3214.0	218.0	0.9364801645278931	Sensitivity
Negative	2137.0	20430.0	0.9053041934967041	specificity
	0.6006354093551636	0.9894420504570007	0.909419596195221	Accuracy
	Precision	NegPrediction		

AE

Predicted Class

	Positive	Negative	Rate	
Positive	626.0	65.0	0.9059334397315979	Sensitivity
Negative	358.0	3381.0	0.9042524695396423	specificity
	0.6361788511276245	0.9811375737190247	0.9045146703720093	Accuracy
	Precision	NegPrediction		

Figure 6.6: Confusion Matrix (Phase II)

Model Development

Phase III

The models in Phase I and II were built separately corresponding to four vulnerability types: API Function Call (API), Array Usage (AU), Pointer Usage (PU), Arithmetic Expression (AE). However, the main goal of research is to detect the software vulnerability in which it may have some overlapping cases among four vulnerability types. There is no clear line or criteria to differentiate all types of vulnerability from the dataset. Therefore, the model development in this phase focused on the model fitting with a combined dataset which included all four vulnerability types. The model evaluation was conducted to examine the performance and predictive power of models built with the combined dataset compared to the models built with each dataset separated by vulnerability types.

In this phase, 1,000 vector arrays in which each array represents tokens in one program slice from combined dataset were randomly spitted into a training set and a test set as discussed in the previous section.

Model Fitting (Phase III)

The architecture of Neural Network for this phase was set as the same inputs from Phase II with ADAM optimizer. The hyperparameters of sequential model include Bidirectional gated recurrent unit (BGRU) of 256 neuron units with 2 hidden layers. The tanh function was applied to produce the outputs of 2 hidden layers and the sigmoid function was used to compute the activation outputs in the last layer. The learning rate is 0.01 with batch size of 32, vector inputs with same shape [mean vector lengths x 30] fitted for 10 epochs. The binary cross-entropy loss was chosen it can speed up the convergence of loss (Appendix B).

	Accuracy	Sensitivity	Specificity
COMBINED	61%	91%	53%
API	53%	69%	46%
AU	64%	79%	62%
PU	38%	81%	31%
AE	61%	61%	62%

Figure 6.7: 1000 Samples with ADAM Optimizer (Phase III)

According to the sensitivity rates shown in figure 6.7, the model fitted with the combined data types outperform all models fitted with dataset separated by types in explaining the target class 1 (vulnerability) as the sensitivity is as high as 91%. Compared to API and PU models, the combined model performs better in detecting the target class 0 (non- vulnerability). The overall performance indicates that the combined model performs similarly to AU and AE models. The combined dataset is more appropriate to utilized to predict vulnerability. Thus, the model was further developed to improve its ability to detect both vulnerability and non-vulnerability types based on combined dataset.

Model Development

Phase IV

The models in Phase III were built with a combined dataset which included all four vulnerability types. The model evaluation was conducted and showed that the performance and predictive power of models built with the combined dataset was better to explain the vulnerability class, compared to the models built with each dataset separated by vulnerability types.

In this phase, the combined dataset was continually used to develop the model, but the study would further focus on the use of Bidirectional Recurrent Neural Networks (BGRU and BLSTM) with ADAM and ADAMAX optimizer to fit the models (Appendix C).

Gated Recurrent Unit (GRU):

- GRU has no explicit memory unit and no forget gate and update gate, hence it trains the model faster than LSTM, but may lead to a lower accuracy rate.
- Comparing to LSTM, the GRU has a simpler architecture which reduces the number of hyperparameters.

Long Short-Term Memory (LSTM):

- LSTM comprises both update gate and forget gate and remembers longer sequences than GRU. However, GRU is also found to be comparable to LSTM on the sequence modeling (Chung et al., 2014).

Bidirectional Recurrent Neural Networks:

- Bidirectional Recurrent Neural Networks provides the original input sequence to the first layer and a reversed copy of the input sequence to the second layer, so there are two layers side-by-side.
- Bidirectional RNNs are found to be more effective than regular RNNs. It has been widely used as it can overcome the limitations of a regular RNN (Schuster et al., 1997). The regular RNN model preserves only information of the past. Whereas, These Bidirectional networks have access to the past as well as the future information; therefore, the output is generated from both the past and future context and leads to a better prediction and classifying sequential patterns.

- The output from fitting LSTM and BLSTM models also indicates that the bidirectional unit outperforms the regular LSTM holding other hyperparameters constant (figure 7.1).

- The BLSTM model has lower loss rate of 0.58 compared to a loss rate of 0.60 from LSTM. The BLSTM also have a higher accuracy rate of 64.2% than that of LSTM model of 62.8%. Note: Both models were fit with same input parameter and dataset. The models were built with a small dataset of 1000 slices so the accuracy rate may not be high. The larger dataset would be used to fit the model in the later section.

BLSTM

```
-
start
Epoch 1/10
9/9 [==============================] - 14s 2s/step - loss: 0.7094 - accuracy: 0.5312
Epoch 2/10
9/9 [==============================] - 11s 1s/step - loss: 0.7036 - accuracy: 0.4549
Epoch 3/10
9/9 [==============================] - 13s 1s/step - loss: 0.6761 - accuracy: 0.6076
Epoch 4/10
9/9 [==============================] - 13s 1s/step - loss: 0.6690 - accuracy: 0.5938
Epoch 5/10
9/9 [==============================] - 13s 1s/step - loss: 0.6559 - accuracy: 0.6111
Epoch 6/10
9/9 [==============================] - 12s 1s/step - loss: 0.6463 - accuracy: 0.6250
Epoch 7/10
9/9 [==============================] - 11s 1s/step - loss: 0.6119 - accuracy: 0.6562
Epoch 8/10
9/9 [==============================] - 11s 1s/step - loss: 0.6092 - accuracy: 0.6632
Epoch 9/10
9/9 [==============================] - 11s 1s/step - loss: 0.5969 - accuracy: 0.6562
Epoch 10/10
9/9 [==============================] - 11s 1s/step - loss: 0.5844 - accuracy: 0.6424
```

LSTM

```
start
Epoch 1/10
9/9 [==============================] - 8s 909ms/step - loss: 0.7051 - accuracy: 0.4722
Epoch 2/10
9/9 [==============================] - 5s 591ms/step - loss: 0.6990 - accuracy: 0.4792
Epoch 3/10
9/9 [==============================] - 5s 537ms/step - loss: 0.6852 - accuracy: 0.5903
Epoch 4/10
9/9 [==============================] - 5s 540ms/step - loss: 0.6764 - accuracy: 0.5938
Epoch 5/10
9/9 [==============================] - 5s 533ms/step - loss: 0.6546 - accuracy: 0.6250
Epoch 6/10
9/9 [==============================] - 5s 534ms/step - loss: 0.6443 - accuracy: 0.5938
Epoch 7/10
9/9 [==============================] - 5s 531ms/step - loss: 0.6478 - accuracy: 0.6007
Epoch 8/10
9/9 [==============================] - 5s 536ms/step - loss: 0.6172 - accuracy: 0.6146
Epoch 9/10
9/9 [==============================] - 5s 533ms/step - loss: 0.6364 - accuracy: 0.5833
Epoch 10/10
9/9 [==============================] - 5s 572ms/step - loss: 0.6096 - accuracy: 0.6285
```

Figure 7.1: BSTM vs LSTM Models Fitted with 1000 Samples

Model Fitting (Phase IV)

The architecture of neural networks such as optimizer and model units of sequential model were same as those of phase III. To examine the performance of BLSTM and BGRU on the dataset, the models in the fourth phase were fit with both BGRU and BLSTM with a training set for combined vulnerability type of 4,000 program slices and evaluated with a test set of 1,000 program slices. The models were fit and evaluated to examine the model performance on dataset. BGRU performs better and train the DL model faster than BLSTM. The accuracy rates are shown in figure 7.2A.

	Accuracy Rate	
	Train	**Test**
BGRU	77.5%	66.4%
BLSTM	72.4%	63.8%

Figure 7.2A: Accuracy Rates (5000 samples)

	Accuracy Rate	
	Train	**Test**
BGRU	87.5%	81.7%
BLSTM	83.4%	79.3%

Figure 7.2B: Accuracy Rates (30,000 samples)

	Accuracy Rate	
	Train	**Test**
BGRU	94.6%	91.0%
BLSTM	92.4%	87.7%

Figure 7.3C: Accuracy Rates (100,000 samples)

As the BGRU model outperforms BLSTM when holding other parameters constant, the further study was conducted to build the models with larger size of dataset to examine whether BGRU still performs better than BLSTM on a large dataset including 30,000 and 100,000 samples. The outputs show the same trend that the BGRU performs very well and better than BLSTM. The accuracy rates of BGRU models for both training and test sets are higher than those of BLSTM. The more details of model performance were discussed in the next section.

Model Validation (Phase IV)

Models were evaluated with test sets. The confusions matrix (figure 7.3A) provides more details for model performance and validation of the models trained with 4,000 samples (80% of 5000 samples) in figure 7.2A. The decision threshold is set to 0.5 for validation. The BGRU model outperforms the BLSTM in most metrices expect the sensitivity. It has a higher accuracy, precision, and specificity which indicates the stronger ability of model to predict both vulnerability and non-vulnerability types. For a larger dataset of 30,000, the BGRU also outperforms BLSTM in every metrics. The obvious improvement is the sensitivity which the BGRU has 90% of sensitivity which is 8% higher than that of BLSTM, indicating that the BGRU model can explain the vulnerability class better than the BLSTM (figure 7.3B). The model from 100,000 datasets also shows the similar performance in which BGRU model performs better than BLSTM. Comparing to BLSTM, The BGRU network typically train, converge, and learn faster. Thus, the BLSTM models that fit with 10 epochs may not reach convergent which can explain why the models for BGRU perform better in 10 epochs.

BGRU

Predicted Class

	Positive	Negative	Rate	
Positive	185.0	41.0	0.8185840845108032	Sensitivity
Negative	295.0	478.0	0.618369996547699	specificity
	0.3854166567325592	0.9210019111633301	0.6636636853218079	Accuracy
	Precision	NegPrediction		

BLSTM

Predicted Class

	Positive	Negative	Rate	
Positive	185.0	41.0	0.8185840845108032	Sensitivity
Negative	321.0	452.0	0.5847347974777222	specificity
	0.365612655878067	0.9168357253074646	0.6376376152038574	Accuracy
	Precision	NegPrediction		

Figure 7.3A: BGRU vs BLSTM Confusion Matrix (5,000 samples)

Predicted Class

	Positive	Negative	Rate	
Positive	731.0	77.0	0.9047029614448547	Sensitivity
Negative	1022.0	4170.0	0.803158700466156	specificity
	0.4169994294643402	0.9818695783615112	0.8168333172798157	Accuracy
	Precision	NegPrediction		

Predicted Class

	Positive	Negative	Rate	
Positive	663.0	145.0	0.8205445408821106	Sensitivity
Negative	1095.0	4097.0	0.7890986204147339	specificity
	0.3771331012248993	0.9658179879188538	0.7933333516120911	Accuracy
	Precision	NegPrediction		

Figure 7.3B: BGRU vs BLSTM Confusion Matrix (30,000 samples)

Predicted Class

	Positive	Negative	Rate	
Positive	2383.0	287.0	0.8925093412399292	Sensitivity
Negative	1506.0	15824.0	0.9130986928939819	specificity
	0.6127539277076721	0.9821860790252686	0.9103500247001648	Accuracy
	Precision	NegPrediction		

Predicted Class

	Positive	Negative	Rate	
Positive	2408.0	262.0	0.9018726348876953	Sensitivity
Negative	2193.0	15137.0	0.8734564185142517	specificity
	0.5233644843101501	0.9829859137535095	0.8772500157356262	Accuracy
	Precision	NegPrediction		

Figure 7.3C: BGRU vs BLSTM Confusion Matrix (100,000 samples)

Model From Full Dataset

The model was fit with the full dataset to evaluate the performance, the overfitting problem, and the difference of network outputs for each vulnerability types. The total 420,067 programs slices were combined into one dataset, comprising 64,403, 42,229, 291,281, and 22,154 from API, AU, PU, and AE types, respectively. The combined dataset was spitted into a training set and a test set with 80/20 ratios. Then, the training set was down-sampling to ensure that both target classes (vulnerability and non- vulnerability) in dataset were balanced.

The model was built with the same hyperparameters from model development in Phase IV with ADAM optimizer. The hyperparameters of sequential model include Bidirectional gated recurrent unit (BGRU) of 256 neuron units with 2 hidden layers. Tanh function was applied to produce the outputs of 2 hidden layers and sigmoid function was applied to compute activation outputs in the last layer. The learning rate is 0.01 with batch size of 32, vector inputs of shape [mean vector lengths x 30]. The cross-entropy loss was chosen as it can speed up the convergence of loss.

```
Epoch 1/10
2824/2824 [==============================] - 13285s 5s/step - loss: 0.4097 - accuracy: 0.8209
Epoch 2/10
2824/2824 [==============================] - 14041s 5s/step - loss: 0.2432 - accuracy: 0.9039
Epoch 3/10
2824/2824 [==============================] - 14114s 5s/step - loss: 0.1904 - accuracy: 0.9267
Epoch 4/10
2824/2824 [==============================] - 14219s 5s/step - loss: 0.1652 - accuracy: 0.9376
Epoch 5/10
2824/2824 [==============================] - 14375s 5s/step - loss: 0.1532 - accuracy: 0.9419
Epoch 6/10
2824/2824 [==============================] - 14330s 5s/step - loss: 0.1444 - accuracy: 0.9459
Epoch 7/10
2824/2824 [==============================] - 14374s 5s/step - loss: 0.1398 - accuracy: 0.9472
Epoch 8/10
2824/2824 [==============================] - 14512s 5s/step - loss: 0.1373 - accuracy: 0.9480
Epoch 9/10
2824/2824 [==============================] - 14566s 5s/step - loss: 0.1371 - accuracy: 0.9489
Epoch 10/10
2824/2824 [==============================] - 14642s 5s/step - loss: 0.1371 - accuracy: 0.9482
```

Figure 8.1: Models Fitted with a Training set

According to figure 8.1, the learning process is faster in the beginning as the loss rates significantly decrease in epoch 1 to 3. The accuracy rates increase for as the training process goes from epoch 1 to 10. The model has the highest accuracy rate of 94.89% in epoch 9 and starts to decrease in epoch 10 as the error rate has no longer minimized. The network outputs range between 0 and 1 as a sigmoid function was applied the output layer. The confusion matrix was developed with a test set (figure 8.2). The model performs well to explain both target classes as the sensitivity and specificity are over 90%. However, the F1 score was further computed in the next section to blend the precision and sensitivity.

Predicted Class

	Positive	Negative	Rate	
Positive	10768.0	439.0	0.9608280658721924	Sensitivity
Negative	5898.0	67019.0	0.9191135168075562	specificity
	0.6461058259010315	0.9934922456741333	0.9246706962585449	Accuracy
	Precision	NegPrediction		

Figure 8.2: Confusion Matrix (Test Set)

Predicted Class

thresdArray	recall	precision	specificity	F1	Accuracy	balanceAccuracy
0.4	0.971803	0.619123	0.908115	0.756372	0.916599	0.939959
0.45	0.965914	0.632301	0.913669	0.764288	0.920629	0.939792
0.5	0.960828	0.646106	0.919114	0.772647	0.924671	0.939971
0.53	0.954939	0.654196	0.922419	0.776464	0.926751	0.938679
0.55	0.952084	0.659905	0.924585	0.779515	0.928249	0.938335
0.58	0.946819	0.669929	0.928302	0.784663	0.930769	0.93756
0.6	0.942714	0.676247	0.930633	0.787551	0.932243	0.936674
0.65	0.930579	0.69351	0.936791	0.794742	0.935964	0.933685
0.7	0.915588	0.712322	0.943168	0.801265	0.939494	0.929378
0.8	0.868564	0.760706	0.958007	0.811065	0.946091	0.913286

Figure 8.3: Metrics across Prediction Thresholds

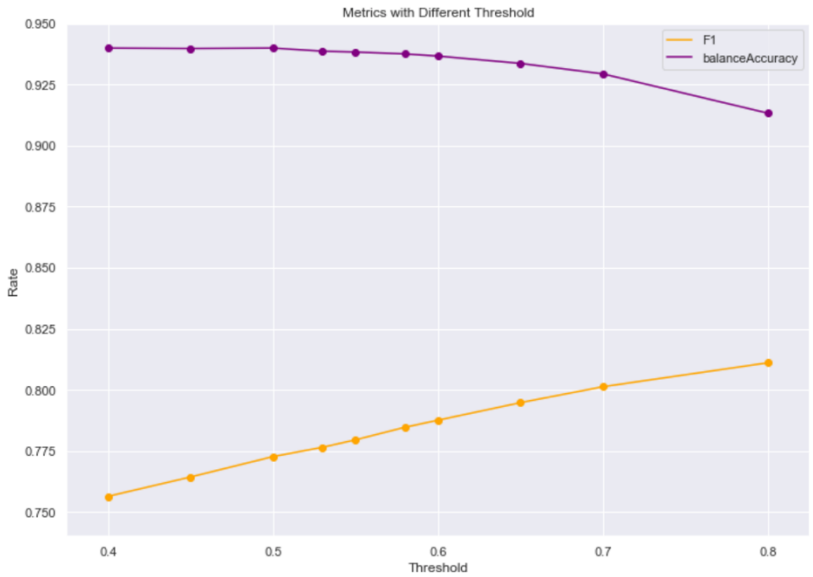


Figure 8.4: F1 vs Balanced Accuracy Rate across Different Thresholds

According to figure 8.4, the F1 scores increase, but balanced accuracy decreases as the threshold increases. The peak point of balanced accuracy is at 0.5. F1 is utilized to blend precision and specificity for model evaluation. The accuracy rate keeps increasing, hence balanced accuracy is more accurate as accuracy is biased with the increasing threshold.

Overall, the model fitted with full dataset performs well with a high balanced accuracy rate of 93%. The high sensitivity and specificity imply a good ability of model in explaining both vulnerability and non-vulnerability classes. The model performs better in predicting non-vulnerability class as it has 99% in negative prediction. However, the predictive power for vulnerability class is still moderately strong as the F1 ranges between 75% to 80% across different thresholds.

Conclusion

The main focus of this study is to develop the strong predictive model using Neural Networks to detect the vulnerability of programs. The original C/C++ programs were originally collected from NVD and SARD. It includes both vulnerability and non-vulnerability programs. The programs were divided into 4 different types as follows: API Function Call (API), Array Usage (AU), Pointer Usage (PU), and Arithmetic Expression (AE). The dataset was generated by transforming the programs to the program slices. The target class is a binary class which 0 represents the non-vulnerability program and 1 represents vulnerability program. Each program slice was transformed to an array of vectors using Word to Vector model and the vector outputs were adjusted with the mean length before fitting the network. Through all phases of model development, the hyperparameters of the model were adjusted to examine the improvement in performance when changing the optimizer, recurrent units, activation functions. The study found that the model with cross-entropy loss, tanh activation for hidden layers, and sigmoid activation for output layer, paired with an ADAM optimizer have the strongest performance in vulnerability detection. The best model from last phase of model development has the highest accuracy rate of 94.89%. For model validation, it performs well on test set with a high balanced accuracy rate of 93% which is comparable to that of a training set. The high sensitivity of 96% and specificity of 91% indicate a good ability of model in explaining both vulnerability and non-vulnerability classes.

Future Studies

As the neural network model is a black-box model which is not required human interpretation. Future studies would be focused on the combination use of machine learning and deep learning in vulnerability detection to automatically extract the feature to eliminate influence of human bias and utilize machine learning to build the classifier for human interpretation. The CNN would be utilized in feature extraction as it can capture non-linear features. The learning would be transferred from Neural Network to Machine Learning by fitting the machine learning models such as SVM and KNN with feature vectors extracted from CNN. The use of deep learning and machine learning is expected to build on this study and improve the model to detect vulnerability.

References

Brodersen, K. H., Ong, C. S. , Stephan, K. E. & Buhmann, J. M. (2010). The Balanced Accuracy and Its Posterior Distribution. *20th International Conference on Pattern Recognition*, Istanbul, 3121-3124. Retrieved from https://ieeexplore.ieee.org/document /5597285

Calins, O. (2020). *Deep Learning Architectures A Mathematical Approach (pp. 28-30).* Springer.

Castaneda, G., Morris, P. & Khoshgoftaar, T.M. Evaluation of maxout activations in deep learning across several big data domains. *J Big Data* 6**,** 72 (2019). https://doi.org/10.1186/s40537-019-0233-0

Chung, J., Gulcehre, C., Cho, K., & Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. In *NIPS 2014 Workshop on Deep Learning, December 2014*

Dufourq, E., & Bassett, B. (2017). Automated Problem Identification: Regression vs Classification via Evolutionary Deep Networks. *Proceedings of the South African Institute of Computer Scientists and Information Technologists,* 2017. Article No.12, 1–9. Retrieved from https://doi.org/10.1145/3129416.3129429

Kingma, D., Ba J. (2015). Adam: A Method for Stochastic Optimization. *The 3rd International Conference for Learning Representations, May 2015.*

Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., & Chen, Z. (2018). SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. *IEEE Dataport.* Retrieved from http://dx.doi.org/10.21227/fhg0-1b35.

Mitchell, T. (1997). *Machine Learning (pp. 96-98).* McGraw-Hill Science.

Mikolov , T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient Estimation of Word Representations in Vector Space. *The International Conference on Learning Representations.* Retrieved from https://arxiv.org/abs/1301.3781v3.

Nielsen, M. (2019). *Neural Networks and Deep Learning* (Chapter 3). Retrieved from http://neuralnetworksanddeeplearning.com/chap3.html

Schuster, M., & Paliwal, K. (1997). Bidirectional Recurrent Neural Networks. *IEEE Transactions on Signal Processing, November 1997* Retrieved from: https://doi.org/10.1109/78.650093.

Appendix A

Model Development Phase I: Hyperparameters

Parameters	Meaning	Value
Input Shape	Shape of Vectors in the Input Layer	Mean Vector Length x 30
Units	Number of Network Units per Layer	256
Layers	Number of Hidden Layers	2
Dropout	Fraction of the units to drop for linear transformation of inputs	0.2
Eta	Learning Rate	0.01
Epoch	Number of Training Steps	10
Batch Size	Number of training examples utilized in one iteration	32
Loss	Cost Function	binary crossentropy
Activation	Activation function to compute outputs for hidden layers	tanh
Recurrent Activation	Activation function to use for the recurrent step	hard_sigmoid
Output Activation	Activation function to compute outputs in the last layer	sigmoid
Gating Mechanism	Internal mechanisms of Recurrent Neural Networks (RNNs)	GRU
Layer Wrapper	Wrapper for RNN	Bidirectional
Optimizer	Optimizer for Training Model	adamax

Appendix B

Model Development Phase II, III: Hyperparameters

Parameters	Meaning	Value
Input Shape	Shape of Vectors in the Input Layer	Mean Vector Length x 30
Units	Number of Network Units per Layer	256
Layers	Number of Hidden Layers	2
Dropout	Fraction of the units to drop for linear transformation of inputs	0.2
Eta	Learning Rate	0.01
Epoch	Number of Training Steps	10
Batch Size	Number of training examples utilized in one iteration	32
Loss	Cost Function	binary Crossentropy
Activation	Activation function to compute outputs for hidden layers	**tanh, relu, sigmoid**
Recurrent Activation	Activation function to use for the recurrent step	hard_sigmoid
Output Activation	Activation function to compute outputs in the last layer	sigmoid
Gating Mechanism	Internal mechanisms of Recurrent Neural Networks (RNNs)	GRU
Layer Wrapper	Wrapper for RNN	bidirectional
Optimizer	Optimizer for Training Model	**adamax, sgd, adam**

Appendix C

Model Development Phase IV: Hyperparameters

Parameters	Meaning	Value
Input Shape	Shape of Vectors in the Input Layer	Mean Vector Length x 30
Units	Number of Network Units per Layer	256
Layers	Number of Hidden Layers	2
Dropout	Fraction of the units to drop for linear transformation of inputs	0.2
Eta	Learning Rate	0.01
Epoch	Number of Training Steps	10
Batch Size	Number of training examples utilized in one iteration	32
Loss	Cost Function	binary_crossentropy
Activation	Activation function to compute outputs for hidden layers	tanh
Recurrent Activation	Activation function to use for the recurrent step	hard_sigmoid
Output Activation	Activation function to compute outputs in the last layer	sigmoid
Gating Mechanism	Internal mechanisms of Recurrent Neural Networks (RNNs)	**GRU, LSTM**
Layer Wrapper	Wrapper for RNN	**Bidirectional, Regular**
Optimizer	Optimizer for Training Model	**adam, adamax**