An approach for predicting multiple-type overflow vulnerabilities based on combination features and a time series neural network algorithm

Naisargi Savaliya – 202211007

Reema Parikh - 202211066

Introduction

- Overflow vulnerability is a common and dangerous software vulnerability that can lead to various hazards. However, previous studies on predicting software overflow vulnerability have only focused on binary classification problems and failed to analyze factors and features that can lead to each type of overflow vulnerability. Therefore, this paper proposes a multiple-type overflow vulnerability prediction method based on a combination of features and a time series neural network algorithm.
- The main contributions of this paper are as follows.

  (1) The causes of eight kinds of software overflow vulnerability were analyzed: **Stack-based Buffer Overflow, Heap-based Buffer Overflow, Buffer Underwrite, Buffer Overread, Buffer Underread, Integer Overflow or Wraparound, Integer Underflow and Free of Pointer – not at Start of Buffer were analyzed.** According to these eight kinds of overflow vulnerability, a method for extracting internal features of the source code was proposed.

  (2) Symbolic transformation and vectorization of the extracted vulnerability features were carried out. According to the software overflow vulnerability features, a bidirectional recurrent neural network based on time series was analyzed, and a multipletype software overflow vulnerability prediction model was built using a bidirectional neural network.

  (3) Our proposed method achieved superior accuracy, precision, recall, and F1 value when predicting overflow vulnerabilities in the Juliet dataset and successfully predicted the corresponding type of vulnerability in four real applications: Linux Kernel, GNU LibreDWG, Espruino, and PDFResurrect.

Vulnerabilities

1. **Stack-based Buffer Overflow:** Occurs when a program writes more data to a buffer than it can hold, causing the excess data to overwrite adjacent memory locations.

2. **Heap-based Buffer Overflow:** Occurs when a program writes more data to a heap-allocated buffer than it can hold, causing the excess data to overwrite adjacent memory locations.

3. **Buffer Underwrite:** Occurs when a program writes less data to a buffer than it can hold, leaving uninitialized or sensitive data in the buffer.

4. **Buffer Overread:** Occurs when a program reads more data from a buffer than it contains, causing the program to access uninitialized or sensitive memory locations.

5. **Buffer Underread:** Occurs when a program reads less data from a buffer than it contains, leaving some of the buffer unread and potentially missing important information.

6. **Integer Overflow or Wraparound:** Occurs when an arithmetic operation on an integer value results in a value that is too large or too small for its intended use, leading to unexpected behavior or security vulnerabilities.

7. **Integer Underflow:** Occurs when an arithmetic operation on an integer value results in a value that is smaller than its intended use, leading to unexpected behavior or security vulnerabilities.

8. **Free of Pointer – not at Start of Buffer:** Occurs when a pointer is freed but not set back to NULL, allowing it to be used again and potentially leading to security vulnerabilities.

Model Architecture

1. **Feature Extraction:**
   - Internal Feature Set(IFS): It refer to the critical code of a software program that contains sensitive functions related to overflow vulnerability in the language library. The critical code consists of several program statements that include all parameters of the sensitive functions, including the feature code in the feature set IFS. If this function calls another function in the program, the critical code includes the related code in the called function.
   - External Feature Set(EFS): It refers to the features of a software program that are not directly related to its internal functions but can still affect its vulnerability to overflow attacks. The external vulnerability feature set (EFS) includes features such as the number of function calls, the number of loops, and the number of conditional statements in the source code. These features are transformed into a fixed dimension embedding vector with overflow vulnerability feature information and joined into an embedding layer containing external vulnerability features.

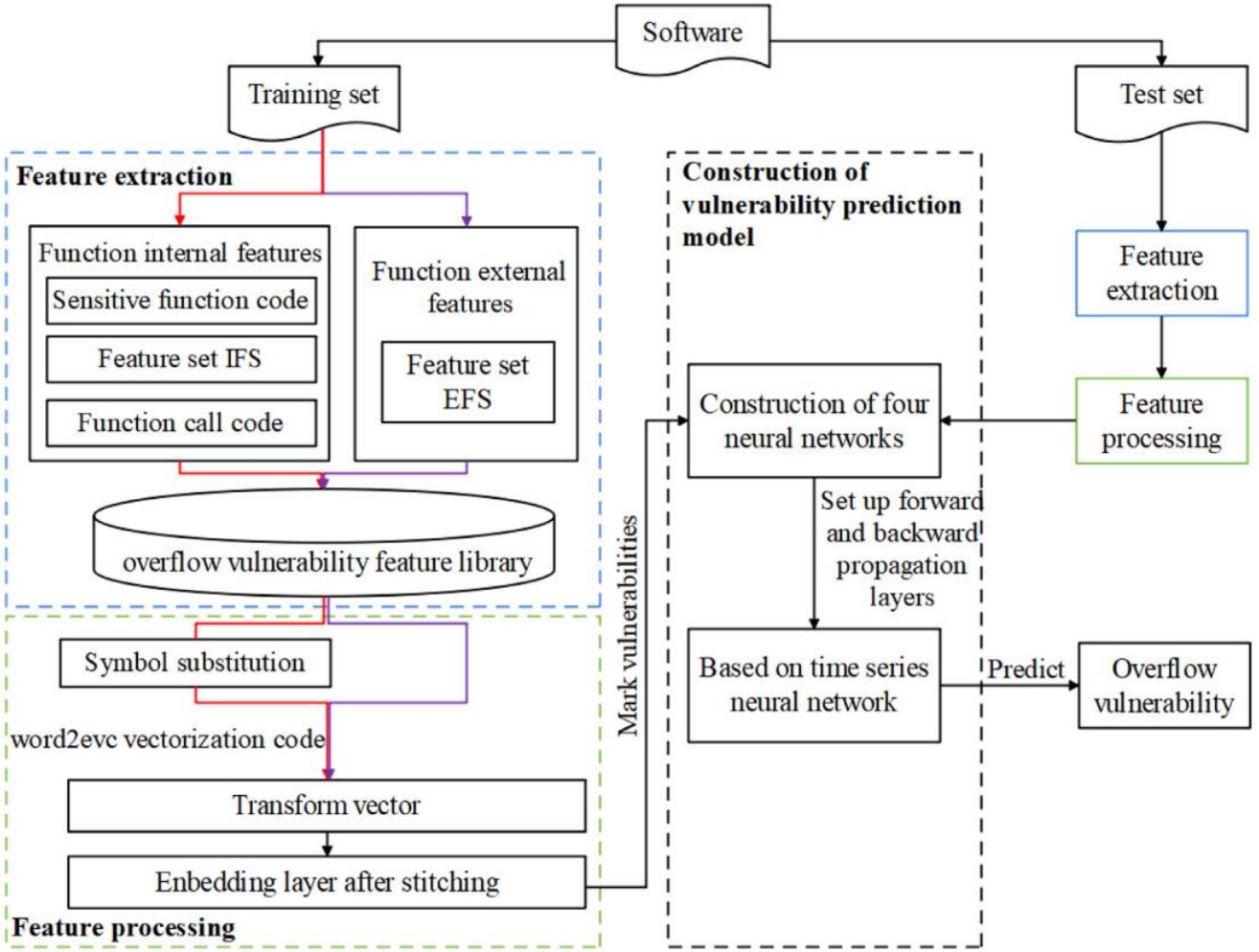

Fig. 1. The framework of the multiple-type overflow vulnerability prediction.

Model Architecture

2. **Feature processing of overflow vulnerability:**
   - **Symbol Substitution:** Symbol substitution is a process of replacing symbols in the source code with their corresponding tokens. After symbol substitution, the internal features are transformed into vector representations to enable neural network models to better learn these features and mine the connection of code context information.
   - **Vector Transformation:** vector transformation is used to transform internal features of software overflow vulnerability extracted from source code after symbol substitution into fixed dimension embedding vectors with overflow vulnerability feature information. These vectors are then joined into an embedding layer containing external vulnerability features to enable neural network models to better learn these features and mine the connection of code context information.
   - **Embedding Layer:** Splicing the embedding layer is a method proposed in the paper to simultaneously feed multiple classes of embedding into neural networks when predicting overflow vulnerability. The method involves using Word2vec to transform all the tokens in the internal vulnerability features of the source code into an embedding vector with information about overflow vulnerability features. All the embedding vectors with vulnerability feature information are then stitched together into an embedding layer containing internal vulnerability features. The dimension of internal vulnerability embedding is set according to the maximum number of internal features.

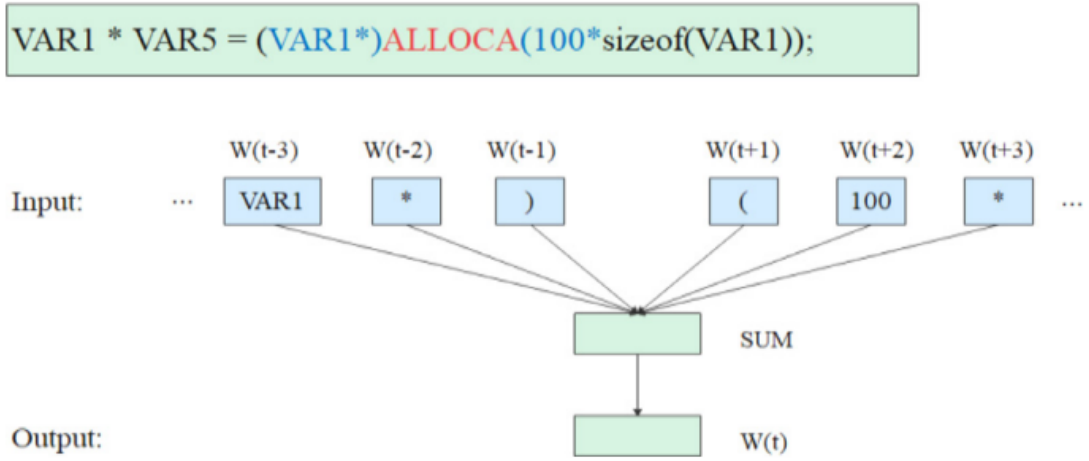

Fig. 2. Vector transformation process.

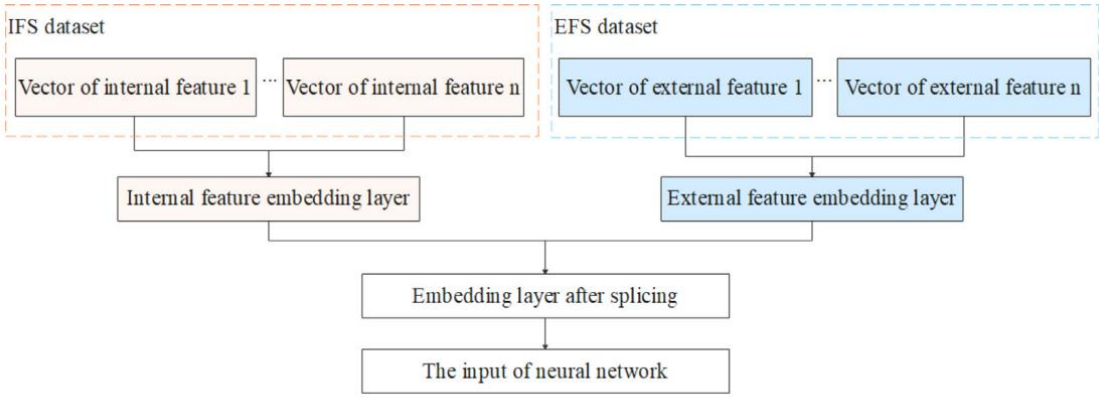

Fig. 3. Embedding processing method.

Model Architecture

3. **Prediction for overflow vulnerability:**
   - The method uses the embedding layer as the neural network input and employs a time series-based bidirectional neural network to predict overflow vulnerability in the program. Bidirectional RNN, bidirectional LSTM, and bidirectional GRU are selected in this section to predict the overflow vulnerability of software. The embedding layer contains sensitive functions, parameters, program control, function calls, and other information related to the overflow vulnerability of the source code.

1. Input layer: This is the first layer of the neural network, where the input data is fed into the model. In this case, the input data is the embedding layer containing internal vulnerability features.

2. Forward propagation layer: This layer processes the input data in a forward direction and generates a hidden state vector that captures contextual information from previous time steps.

3. Backward propagation layer: This layer processes the input data in a backward direction and generates a hidden state vector that captures contextual information from future time steps.

4. Output layer: This is the final layer of the neural network, where the output prediction is generated based on the hidden state vectors generated by both forward and backward propagation layers.

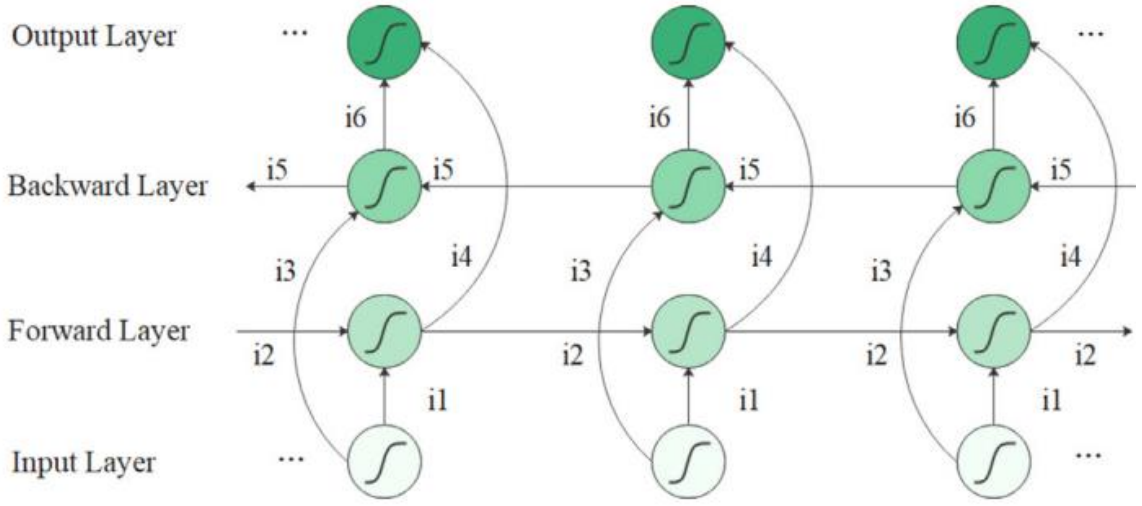

Fig. 4. Bidirectional recurrent neural network based on time expansion.

Dataset

- Distribution of different categories.
- Total data – 34,634
- Train data – 27,707
- Test data – 6,927

```
Heap-based Buffer Overflow                  9028
Stack-based Buffer Overflow                 8001
Integer Overflow                            4163
Buffer Underread                            3287
Buffer Underwrite                           3287
Integer Underflow                           2546
Buffer Overread                             2273
Not prone to overflow type vulnerability    1144
Free of Pointer - not at Start of Buffer     905
Name: label, dtype: int64
```

Implementation - Feature Extraction

```
[ ] train_df.head(10)
```

	code	label	internal_feature_set	external_feature_set
24367	~ ~ ~#include ~#include ~ ~#define CHAR_ARR...	Integer Underflow	{if (fgets(input_buf, CHAR_ARRAY_SIZE, stdin) ...	{'AltCountLineCode': 52, 'AltCountLineComment'...
29673	~ ~#ifndef OMITGOOD ~ ~#include ~#include ~...	Integer Overflow	{#endif, if(data > 0), #ifndef OMITGOOD}	{'AltCountLineCode': 12, 'AltCountLineComment'...
5832	~ ~ ~#include ~ ~#ifndef _WIN32 ~# include <...	Stack-based Buffer Overflow	{#ifndef OMITGOOD, #ifndef _WIN32, if(static_r...	{'AltCountLineCode': 88, 'AltCountLineComment'...
27509	~ ~ ~#include ~#include <vector> ~ ~using na...	Integer Overflow	{if (data < (CHAR_MAX/2)), #ifndef OMITGOOD, #...	{'AltCountLineCode': 38, 'AltCountLineComment'...
22338	~ ~ ~#include ~ ~#include <wchar.h> ~ ~#ifnd...	Buffer Underwrite	{#ifndef OMITGOOD, if(CWE124_Buffer_Underwrite...	{'AltCountLineCode': 70, 'AltCountLineComment'...
29421	~ ~ ~#include ~ ~#include <math.h> ~ ~#ifnde...	Integer Overflow	{ffun37 (fun34,fun35,fun36);, #ifndef OMITBAD,...	{'AltCountLineCode': 41, 'AltCountLineComment'...
28432	~ ~ ~#include ~ ~#include <math.h> ~ ~#ifnde...	Integer Overflow	{fun5 (fun2,fun3,fun4);, #ifndef OMITGOOD, #if...	{'AltCountLineCode': 179, 'AltCountLineComment...
16888	~ ~ ~#include ~ ~#ifndef _WIN32 ~# include <...	Heap-based Buffer Overflow	{#ifndef OMITGOOD, #ifndef _WIN32, #ifdef INCL...	{'AltCountLineCode': 54, 'AltCountLineComment'...
8935	~ ~ ~#include ~ ~#include <wchar.h> ~ ~wchar...	Heap-based Buffer Overflow	{#ifndef OMITBAD, #endif, #ifndef OMITGOOD, #...	{'AltCountLineCode': 38, 'AltCountLineComment'...
7479	~ ~ ~#include ~ ~#include <wchar.h> ~ ~#ifnd...	Stack-based Buffer Overflow	{for(i = 0; i < 0; i++), #ifndef OMITGOOD, #if...	{'AltCountLineCode': 66, 'AltCountLineComment'...

Table 4
Sensitive Function.

	Function
String memory function	*fscanf, vscanf, vsscanf, vfscanf, vsprintf, gets, getopt, getpass, strcpy, strecpy, strcat, strncat, strtrns, strlen, sprint, scanf, sscanf, strcmp, strncmp, strncpy, strlwr, strtok, strset, strstr, strchr, strrchr, strerror, strupr, strrev, streadd, realpath, strsep, etc.*
Dynamic allocation function	*malloc, calloc, free, alloc, realloc, etc.*
Memory manipulation function	*memset, memcpy, memmove, memset, memchr, etc.*

Table 5
External feature.

Feature	Specific content of feature
AltCountLineCode	Number of lines containing source code, including inactive regions.
AltCountLineComment	Number of lines containing comment, including inactive regions.
CountInput	Number of calling subprograms plus global variables read.
CountLine	Number of all lines.
CountLineCode	Number of lines containing source code.
CountLineCodeDecl	Number of lines containing declarative source code.
CountLineCodeExe	Number of lines containing executable source code.
CountLineComment	Number of lines containing comment.
CountLinePreprocessor	Number of preprocessor lines.
CountOutput	Number of called subprograms plus global variables set.
CountPath	Number of possible paths, not counting abnormal exits or gotos.
CountPathLog	log10, truncated to an integer value, of the metric CountPath.
CountSemicolon	Number of semicolons.
CountStmt	Number of statements.
CountStmtDecl	Number of declarative statements.
CountStmtExe	Number of executable statements.
Cyclomatic	Cyclomatic complexity.
CyclomaticModified	Modified cyclomatic complexity.
CyclomaticStrict	Strict cyclomatic complexity.
Essential	Essential complexity.
Knots	Measure of overlapping jumps.
MaxEssentialKnots	Maximum Knots after structured programming constructs have been removed.
MaxNesting	Maximum nesting level of control constructs.
MinEssentialKnots	Minimum Knots after structured programming constructs have been removed.
RatioCommentToCode	Ratio of comment lines to code lines.

Implementation - **Embedding Layer**

```
print(embedding_layer)
```

```
[[ 0.          0.          0.         ...  0.          0.
   0.        ]
 [-0.29068458  0.53642339 -0.60703635 ...  0.          0.
   0.        ]
 [-0.29068458  0.53642339 -0.60703635 ...  0.          0.
   0.        ]
 ...
 [-0.20125018  0.33763164 -0.40286806 ...  0.          0.
   0.        ]
 [-0.22764328  0.34522724 -0.22111468 ...  0.          0.
   0.        ]
 [-0.30095315  0.55227083 -0.58074278 ...  0.          0.
   0.        ]]
```

```
[15] print(embedding_layer.shape)
```

```
(27707, 4600)
```

Implementation - **Model**

```
from keras.layers import Dense,Dropout,GRU,Bidirectional
from keras.models import Sequential
from keras.optimizers import SGD
# The GRU architecture
regressorGRU = Sequential()
# First GRU layer with Dropout regularisation
regressorGRU.add(GRU(units=50, return_sequences=True, input_shape=(embedding_layer.shape[1],1), activation='tanh'))
regressorGRU.add(Dropout(0.2))
# Second GRU layer
regressorGRU.add(GRU(units=50, return_sequences=True, input_shape=(embedding_layer.shape[1],1), activation='tanh'))
regressorGRU.add(Dropout(0.2))
# Third GRU layer
regressorGRU.add(GRU(units=50, return_sequences=True, input_shape=(embedding_layer.shape[1],1), activation='tanh'))
regressorGRU.add(Dropout(0.2))
# Fourth GRU layer
regressorGRU.add(GRU(units=50, activation='tanh'))
regressorGRU.add(Dropout(0.2))
# The output layer
regressorGRU.add(Dense(units=1))
# Compiling the RNN
regressorGRU.compile(optimizer=SGD(lr=0.01, decay=1e-7, momentum=0.9, nesterov=False),loss='mean_squared_error')
# Fitting to the training set
regressorGRU.fit(embedding_layer,train_df['Label'],epochs=10,batch_size=150)
```

Implementation - Result

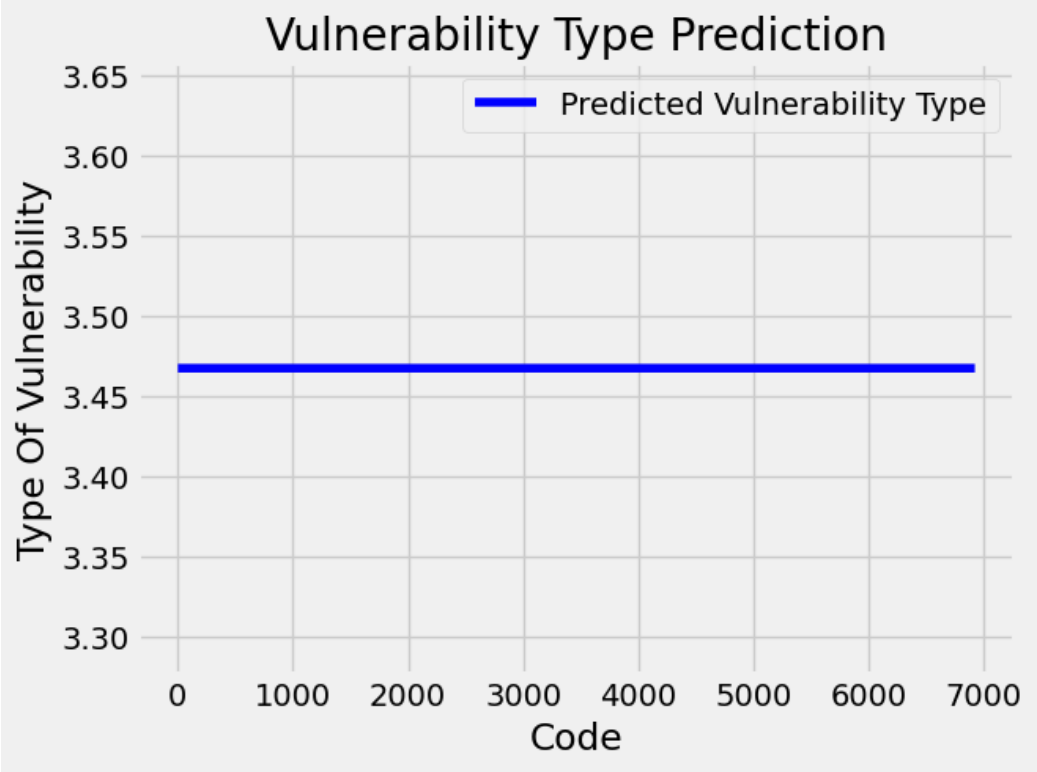

Table 7
Experimental results of *RNN*, *LSTM*, and *GRU* bidirectional recurrent neural network.

Neural Network	Accuracy (%)	Precision (%)	Recall rate (%)	F1 (%)
Bidirectional RNN	82.22	64.57	63.43	62.43
Bidirectional LSTM	98.14	97.08	97.07	97.05
Bidirectional GRU	96.95	95.22	94.24	94.69

Challenges faced during implementation

1. The dataset was too large containing 88820 codes due to which the code was crashing.
2. The model was not accepting the generated embedding layer. Input shape and embedding layer shape was not matching.
3. The model is not getting trained enough to predict the accurate output as we don't have enough space and GPU.

References

1. Z. Zheng, B. Zhang, Y. Liu, J. Ren, X. Zhao and Q. Wang, ”An approach for predicting multiple-type overflow vulnerabilities based on combination features and a time series neural network algorithm,” Computers Security, vol. 114, pp. 102572, 2022. DOI: 10.1016/j.cose.2021.10257
2. Dahl, William Arild, László Erdődi and Fabio Massimo Zennaro. “Stack-based Buffer Overflow Detection using Recurrent Neural Networks.” ArXiv abs/2012.15116 (2020): n. pag.
3. S. Checkoway et al., ”The (in)security of proprietary cryptography,” in Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS), Chicago, IL, USA, Oct. 2011, pp. 413-424.
4. T. Ye, L. Zhang, L. Wang and X. Li, "An Empirical Study on Detecting and Fixing Buffer Overflow Bugs," 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST), Chicago, IL, USA, 2016, pp. 91-101, doi: 10.1109/ICST.2016.21.
5. S. Nicula and R. D. Zota, "Exploiting stack-based buffer overflow using modern day techniques," in Proc. Computer Science, vol. 160, pp. 9-14, 2019, ISSN 1877-0509.

Thank You