Revolutionizing C++ Security: Detecting Software Vulnerabilities through Advanced Assembly Code Analysis and Deep Learning Techniques

Ankit Chowdhury - 20ME31009

Under the supervision of

Prof Ayantika Chatterjee
Advanced Technology Development Centre, IIT Kharagpur

Abstract

Code reuse is prevalent in software development, introducing a significant array of vulnerabilities that pose a threat to software security. The proliferation of the Internet of Things (IoT) exacerbates the impact of code reuse-related vulnerabilities. This research addresses the critical need for effective software vulnerability detection, particularly in the context of C++ programming language, through the utilization of advanced assembly code analysis and deep learning techniques. A comparative study was conducted, evaluating existing software vulnerability detection models such as Asm2Vec, DeepBinDiff and PalmTree which uses natural language processing and predominantly operate as mono-architecture models. Most existing models focus on source code analysis, with limited attention to assembly-level code. This research explores the significance of detecting vulnerabilities in assembly-level code, employing reverse engineering tools such as Ghidra for C and C++ codes, and Androguard, JADX, and APKTools for various APKs. The study also involved a comparison of assembly code generated by Clang and GCC compilers, with the development of a code converter from Clang to GCC. The research investigated the effectiveness of source code, decompiled code, and disassembled code in vulnerability detection. Standard CWE-listed C++ vulnerabilities were analyzed and used to train the proposed deep learning model. The proposed model utilizes word-to-vector conversion to represent the assembly code as a standardized vector. Each word is encoded into a vector of length 40, ranging from -1 to 1, with a maximum slice length of 500. This vector serves as input to a Bidirectional Gated Recurrent Unit (BGRU) deep learning model constructed using the Keras library. The BGRU architecture incorporates bidirectional layers and dropout layers to enhance the model's ability to capture information from both past and future contexts while preventing overfitting. The model's effectiveness in processing and analyzing sequential data for software vulnerability detection is demonstrated through extensive experimentation and evaluation.

Introduction

In the dynamic landscape of today's interconnected world, the art of coding stands as a cornerstone of technological progress, underpinning virtually every facet of our daily lives. As a fundamental tool in software development, coding empowers the creation of innovative applications, websites, and software solutions that drive efficiency, connectivity, and convenience. From the algorithms that power search engines to the intricate logic behind social media platforms, coding plays an indispensable role in shaping the digital fabric of our society. Its importance extends beyond the realms of technology, influencing fields as diverse as healthcare, finance, and entertainment. In this era of rapid technological advancement, the mastery of coding not only opens doors to creative problem-solving but also equips individuals with the skills to navigate and contribute meaningfully to an increasingly digital and interconnected world.

This research embarks on the crucial task of uncovering and comprehending vulnerabilities embedded in C++ applications. As technological landscapes evolve, traditional methods for identifying these vulnerabilities encounter challenges, particularly in the face of widespread code reuse. Our study not only navigates the complex terrain of C++ security, exposing vulnerabilities arising from code reuse, but also introduces an innovative approach for their mitigation. This pursuit carries significance not only within the realm of IoT but extends to broader contexts, underscoring the critical importance of robust vulnerability detection for ensuring software security across diverse applications and scenarios. In an era where digital systems permeate every facet of our lives, addressing vulnerabilities becomes not just a technical necessity but a fundamental element in safeguarding the integrity and functionality of our interconnected digital world.

Various existing software vulnerability detection models were scrutinized initially, and a distinction was observed between mono-architecture models such as Asm2Vec, DeepBinDiff, and PalmTree, and cross-architecture models like Vulhawk. Notably, specialization in a specific computer architecture charac-

terizes mono-architecture models, whereas a broader range is addressed by cross-architecture models, exemplified by Vulhawk. The operation of these tools primarily on the original source code, with a simpler approach, was the focus, but attention was shifted towards the assembly code for a more detailed analysis, considering its numerous advantages.

In the subsequent phase, a thorough report was unfolded, systematically unraveling the distinctive advantages and drawbacks associated with diverse reverse engineering software tools. The scope expanded beyond a functional evaluation, delving into the intricacies of tools such as Ghidra for C and C++ codes, alongside Androguard, JADX, and APKTools, specifically tailored for a spectrum of Android applications. This comprehensive analysis brought to light the efficacy and limitations inherent in each tool's ability to dissect complex layers of code. Ghidra, for instance, was identified as a stalwart solution for unraveling intricacies in C and C++ programming, offering profound insights into potential vulnerabilities. Conversely, specialized competencies in parsing through the intricacies of Android applications were demonstrated by tools like Androguard, JADX, and APKTools, presenting distinctive viewpoints on security nuances. The examination extended beyond mere functional assessment to include considerations of user-friendliness, speed, and comprehensiveness, providing a nuanced understanding of each tool's capacity to navigate the complexities of code. This meticulous scrutiny ensured that the selected tools seamlessly aligned with the intricate demands of the software vulnerability detection study.

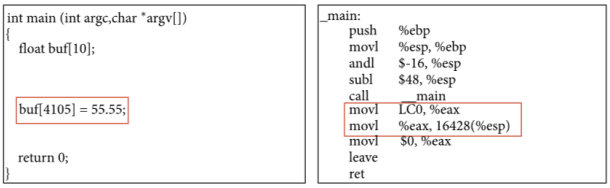
In consideration of the multifaceted nature of vulnerabilities, a deep learning model tailored for the intricacies of C++ security is introduced in this research. Through the utilization of word-to-vector conversion, assembly code is processed as a standardized vector, enabling a seamless integration of advanced machine learning techniques. The adaptability of the approach is underscored by the utilization of a Bidirectional Gated Recurrent Unit (BGRU) architecture, enhanced with bidirectional layers and dropout layers, contributing to the model's capability in capturing sequential patterns and improving generalization.

As this journey is undertaken, the synthesis of advanced assembly code analysis and deep learning techniques emerges as a promising avenue for fortifying C++ applications against an evolving security threat landscape. The research strives to not only contribute to the theoretical understanding of software vulnerabilities but also enhance practical security measures in the development and deployment of C++ applications.

Background Details

Reverse Engineering

In the context of software systems, reverse engineering is a multifaceted process involving the analysis of binary code or executables to unveil source code and algorithms. This practice holds significant importance for various purposes, including comprehending proprietary technologies, detecting vulnerabilities, and crafting interoperable alternatives. Software reverse engineering is accomplished through tools such as disassemblers, which aid in decoding machine code, and debuggers, which facilitate the inspection of program execution. Notably, in the field of cybersecurity, reverse engineering is indispensable for security analysts to uncover vulnerabilities, understand malware behavior, and formulate effective countermeasures. For instance, when encountering a new strain of malware, analysts employ reverse engineering techniques to dissect its code, identify malicious functionalities, and uncover potential vulnerabilities it exploits. This in-depth analysis enables the development of signatures and patches, enhancing the ability to protect systems from specific threats. The role of reverse engineering in proactive cybersecurity defense is paramount, providing crucial insights into the methods of cyber threats and contributing to ongoing innovation in cybersecurity practices. Legal and ethical considerations, particularly regarding intellectual property rights, are integral aspects of the reverse engineering process.



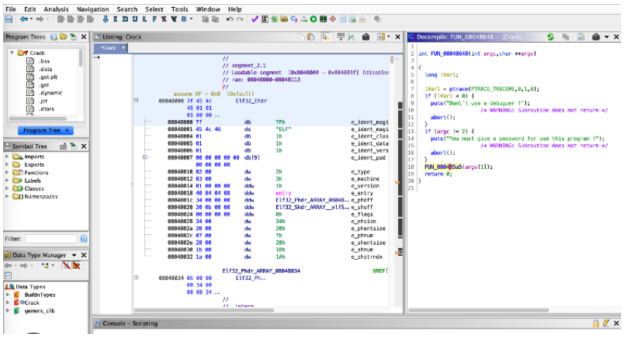
Ghidra

In this research, Ghidra takes precedence over other reverse engineering tools such as Radare2 or IDA Pro, primarily due to its distinct advantages and features that align seamlessly with the intricate requirements of the study.

- The open-source nature of Ghidra promotes collaborative analysis and transparency within the cybersecurity community, making it a preferred choice for this research.
- Its multi-platform support ensures accessibility across various operating systems, allowing for widespread usage and collaboration.
- The interactive disassembly feature and integrated decompiler simplify the complex task

of analyzing assembly code, providing a user-friendly and efficient experience.

- Additionally, Ghidra's collaborative analysis and scripting capabilities, along with extensive processor support, make it a versatile and adaptable tool for in-depth exploration.



In using Ghidra for this research, it's crucial to acknowledge that, like any tool, Ghidra has its share of limitations. One notable drawback is its challenge in properly reverse engineering certain codes.

1. Ghidra faces difficulties in fully supporting true inheritance and mixing structure definitions with class methods directly as a Class data type. Despite being on the agenda for quite a while, attempts to add general support for classes, often in the form of namespaces and Class folders in the datatypes, have fallen short. These attempts are primarily conventions and lack enforcement, leading to issues, especially when recovering hierarchies or dealing with edited method signatures and new class relationships. Additionally, conventions pose challenges when recovering RTTI type information, as they offer a static view of classes.
2. Another area of consideration is Ghidra's handling of "shifted pointers," a concept proposed in a pull request. While the idea is good, implementing support for pointers declared as pointing to an offset into a larger structure requires careful consideration. This change necessitates a database refactor to accommodate such modifications in a comprehensive manner.
3. Furthermore, Ghidra encounters challenges with function signatures treated as first-class type objects, both as a definition in the type system and an entry in the data type manager. This dual representation of information can introduce complexities in maintaining consistency and synchronization between the two.

Despite these limitations, Ghidra's advantages in accessibility, collaboration, and versatile analysis make it a valuable choice for this research, and efforts will be made to address these challenges within the research's context.

Bidirectional Gated Recurrent Unit

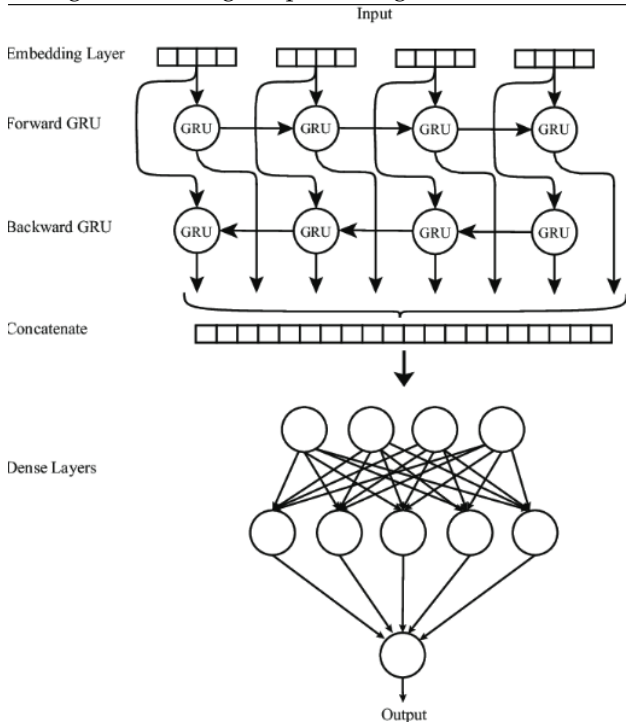
The Bidirectional Gated Recurrent Unit (BGRU) is a specialized architecture within the realm of recurrent neural networks (RNNs). RNNs are designed for processing sequential data where the order of elements is crucial. The Gated Recurrent Unit (GRU), a variant of traditional RNNs, introduces gating mechanisms to selectively update information, addressing challenges like the vanishing gradient problem and enhancing the ability to capture long-range dependencies. In the context of the provided script, the BGRU incorporates bidirectional processing, meaning it analyzes sequences in both forward and backward directions. This bidirectional approach enables the model to comprehend information from both past and future contexts, making it particularly adept at understanding complex sequential patterns.

+ Bidirectional Gated Recurrent Unit (BGRU) architectures demonstrate enhanced effectiveness compared to unidirectional RNNs and CNNs, surpassing the performance of Deep Belief Networks (DBNs) and other shallow learning models. This hierarchy in effectiveness underscores the power of BGRUs in capturing complex patterns within sequential data.
+ However, it's essential to recognize that the effectiveness of BGRUs is significantly influenced by the nature of the training data. Notably, if certain syntax elements (e.g., tokens) frequently appear in pieces of code marked as vulnerable or non-vulnerable, these elements can contribute to high false-positive rates and corresponding false-negative rates. In other words, the prevalence of specific syntax elements plays a pivotal role in influencing the model's predictive accuracy. This observation provides valuable insights into the causes of false-positives and false-negatives, offering a basis for understanding and potentially mitigating these misclassifications.

The script utilizes Keras, a high-level neural networks API in Python, to implement the BGRU model. Keras simplifies the construction and training of neural network models through its Sequential API, making it accessible to researchers and practitioners without requiring in-depth knowledge of neural network intricacies. Various Keras modules are employed in the script for essential components like masking, bidirectional GRU layers, and dense layers.

In summary, the script's use of BGRU and Keras suggests a focus on tasks involving sequential data, such as natural language processing or time-series

analysis. The BGRU's bidirectional nature enhances its ability to understand complex sequential patterns, while Keras provides a user-friendly interface for building and training deep learning models.



Comparative Studies

Mono-Architecture v/s Cross-Architecture

Mono-architecture tools like Asm2Vec, DeepBinDiff and PalmTree are specifically designed to detect vulnerabilities in software that is compiled for a single architecture. This means that they can be very accurate in detecting vulnerabilities that are specific to that architecture. These are typically less computationally expensive than cross-architecture tools. These tools may miss vulnerabilities that are not specific to the architecture

- Asm2Vec is a approach to representing assembly code as vectors, allowing for the application of machine learning techniques to assembly code analysis.
- DeepBinDiff is a deep learning-based approach to binary diffing. Binary diffing is the process of identifying the differences between two binary files.
- PalmTree is a static analysis tool for detecting vulnerabilities in C and C++ code. PalmTree uses a novel approach to static analysis that is based on program dependency graphs.

Cross-architecture tools like VulHawk, VulDeepecker can be used to detect vulnerabilities in software that is compiled for multiple architectures. These tools are typically more computationally expensive than mono-architecture tools. This is because they need to analyze code for multiple architectures.

- VulDeepecker uses a technique called deep symbolic execution to detect vulnerabilities. Deep symbolic execution is a method of static analysis that uses a deep neural network to symbolically execute a program
- VulHawk is a cross-architecture software vulnerability detection tool that uses entropy-based binary code search to identify vulnerabilities in software that is compiled for different architectures

Source Code v/s Disassembled Code

Source code is the human-readable code that programmers write. It is the most natural and understandable form of code, and it is often the best starting point for vulnerability detection. However, source code can be complex and difficult to analyze, and it can be difficult to automate vulnerability detection from source code.

Disassembled code is the machine code that is generated from source code. It is the code that the computer actually executes, and it is the most direct representation of the program. Disassembled code is often easier to analyze than source code, and it is easier to automate vulnerability detection from disassembled code. However, disassembled code can be difficult to understand, and it can be difficult to correlate it back to the source code.

Features	Source Code	Disassembled Code
Understandability	More understandable	Less understandable
Correlation to source code	Easy to correlate	Difficult to correlate
High-level vulnerability detection	Better	Worse
Low-level vulnerability detection	Worse	Better

Ghidra v/s Radare2

Ghidra is a robust open-source software reverse engineering framework renowned for its user-friendly graphical interface and advanced features. Developed by the National Security Agency (NSA), Ghidra caters to both beginners and experienced reverse engineers, offering automatic analysis and a built-in decompiler that aids in understanding assembly code. Its collaborative features and support for a variety of architectures make it a valuable tool for diverse reverse engineering tasks.

Radare2, in contrast, is a highly flexible and powerful command-line-oriented reverse engineering

framework. Its strength lies in its versatility, supporting a vast range of architectures and file formats. While Radare2 has a steeper learning curve, it appeals to advanced users who appreciate its scripting capabilities and the ability to handle complex reverse engineering challenges. Radare2's modular design and cross-platform compatibility contribute to its popularity among seasoned reverse engineers seeking a customizable and extensible tool for in-depth analysis and exploration of binaries.

In terms of strengths, Ghidra boasts a robust decompiler, a large user base, and a well-established reputation. Radare2 shines in its flexibility, extensibility, and powerful command-line interface. Regarding weaknesses, Ghidra's complexity and limited extensibility compared to Radare2 could be drawbacks. Radare2's relative newness and less powerful decompiler compared to Ghidra's could also be considered limitations. Ultimately, the choice between Ghidra and Radare2 hinges on specific user requirements. If power and maturity are priorities, Ghidra is a strong choice. If flexibility, extensibility, and a command-line interface are preferred, Radare2 is an excellent option.

Data Pre-processing

Slice Labelling

A simple code is written to give each code slice a label indicating the presence of vulnerabilities. All the vulnerability functions have been defined and given by the CWE(Common Weakness Enumeration). The procedure of the code is described as follows:

- Obtain the file names, vulnerability locations, vulnerability types, and other information from the vulnerability file.
- Traverse the training codes and the vulnerability information to obtain the vulnerability code and vulnerability function names. Collect this information to populate the vulnerability function library.
- Traverse each code slice and vulnerability function library to determine whether the code slice has vulnerability functions or call vulnerability functions.
- Divide each code slice into two categories:
  - If a code slice has vulnerability functions or call vulnerability functions, it will be labeled with "1."
  - Otherwise, it will be labeled with "0."

Assembly Code to Vector

The neural network requires vectors as input, leading to the use of Word2Vec for converting the data into vector form. The constructed dataset serves as a corpus to train Word2Vec, producing word embeddings crucial for the subsequent task. Code slices are tokenized and converted into fixed-length vectors through the trained Word2Vec model, with each code slice having a length of 40 and a word embedding dimension of 500.

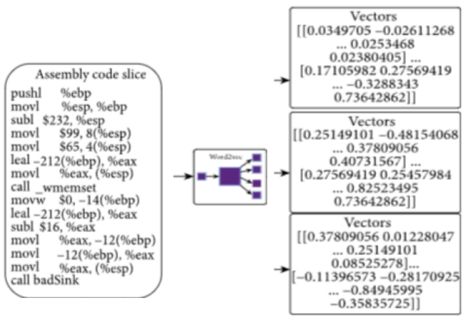
– This code written uses a Python implementation for converting assembly code into vectors using Word2Vec models. The process involves refining raw assembly code, tokenizing it into instructions, and generating vector representations for each instruction. The resulting vectors capture both opcode positions and operand information.

– The refine-asmcode function preprocesses the raw assembly code by removing unnecessary elements, such as prefixes and symbols, ensuring a standardized input for vectorization. This step enhances the model's performance by providing a consistent and clean assembly code input.

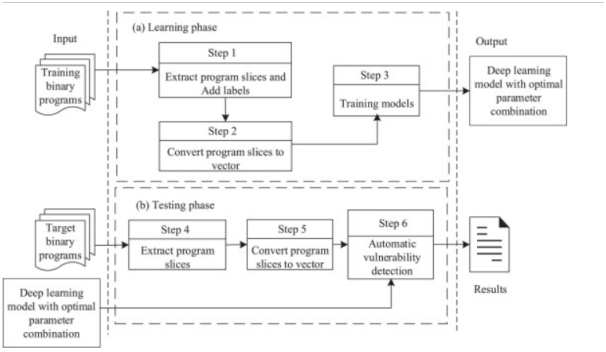
– The instruction2vec function transforms each assembly instruction into a vector representation. It takes into account opcode positions and operand details, including registers, hexadecimal addresses, and integers. The vectors are concatenated to form a comprehensive representation of the assembly code, capturing both the structural and semantic aspects of the instructions.

Functions like read-asmcode-corpus and gen-instruction2vec-model facilitate the overall vectorization process. The former reads and refines an entire assembly code corpus, preparing it for further processing. The latter generates a Word2Vec model based on the refined corpus, allowing the mapping of assembly instructions to meaningful vectors.

The implementation includes error handling to address potential issues during the vectorization process. When errors are encountered, the code prints details to aid in debugging, ensuring robustness in the face of diverse assembly code structures.



In summary, this comprehensive approach considers both opcode and operand information, providing a robust method for converting assembly code into meaningful vectors. These vectors serve as crucial input features for subsequent tasks, such as vulnerability detection or code classification. The code effectively incorporates advanced machine learning techniques for code analysis.



Model - Bidirectional Gated Recurrent Unit

Model Architecture:

The BGRU model is constructed using the Keras Sequential API.

- The model includes Bidirectional Gated Recurrent Unit (BGRU) layers, a masking layer, and a dense layer.
- The masking layer is used to handle variable-length sequences, with a mask value set to 0.0.
- BGRU layers are employed with specified units, activation functions, and recurrent activation functions.

The model is compiled using binary cross-entropy loss and the Adamax optimizer. Additionally, metrics such as true positives (TP), false positives (FP), false negatives (FN), precision, recall, and F-beta score are monitored.

Data Loading and Processing:

The code loads training data from the specified directory (traindataSetPath).

- The data is assumed to be preprocessed and stored in binary files.
- The training dataset is shuffled, and labels are binarized to represent the presence or absence of vulnerabilities.

Data is then split into batches, and a data generator (generator-of-data) is created to yield batches during training.

Training the Model:

The model is trained using the fit-generator method, where the generator yields batches of training data. The training process is monitored, and the weights of the trained model are saved to the specified path (weightPath).

Testing and Evaluation:

The code also includes testing functionality on a separate test dataset. Test data is loaded, and the model is evaluated using the evaluate-generator method. Metrics such as TP, FP, FN, precision, recall, F-beta score, false positive rate (FPR), false negative rate (FNR), accuracy, training time, and test time are calculated and printed.

Result Analysis:

The code performs result analysis by writing the TP, FP, and FN filenames to corresponding text files. Additional result metrics are written to a result file, including FPR, FNR, accuracy, precision, recall, F-beta score, and training/test times.

Test Real Data:

The code includes a function (testrealdata) to test the model on real data. It loads a model with pre-trained weights and evaluates it on a real test dataset.

Additional Functionality:

The code includes functionality to save a dictionary (dict-testcase2func) mapping test cases to functions and their corresponding classifications (TP, FP).

Main Execution:

The main function sets the parameters for batch size, vector dimensions, maximum length, number of layers, and dropout. The main function is called with the specified parameters, triggering the training and evaluation process.

In summary, this code provides a comprehensive pipeline for training a BGRU model on a vulnerability dataset, evaluating its performance, and analyzing the results. It includes features for handling real test data and storing useful information for further analysis.

Model Comparison Metrics

False Positive Rate (FPR): FPR was calculated for the BGRU model and compared with other models

in the experiment. Performance in terms of misclassifying non-vulnerable instances was analyzed.

False Negative Rate (FNR): Similarly, FNR was computed for the BGRU model and contrasted with other models, focusing on misclassification of vulnerable instances.

Accuracy (A): The overall accuracy of the BGRU model was compared with other models.

Precision (P): Precision, representing the ratio of true positives to total predicted positives, was evaluated for the BGRU model and other models.

F1 Score (F1): The F1 score, a harmonic mean of precision and recall, was computed for the BGRU model and used for comparison.

Matthews Correlation Coefficient (MCC): MCC, a correlation coefficient between observed and predicted binary classifications, was calculated for the BGRU model and comparative models.

$$\mathrm{FPR} = \frac{\mathrm{FP}}{\mathrm{FP} + \mathrm{TN}},$$

$$\mathrm{FNR} = \frac{\mathrm{FN}}{\mathrm{TP} + \mathrm{FN}},$$

$$A = \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{TP} + \mathrm{TN} + \mathrm{FP} + \mathrm{FN}},$$

$$P = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}},$$

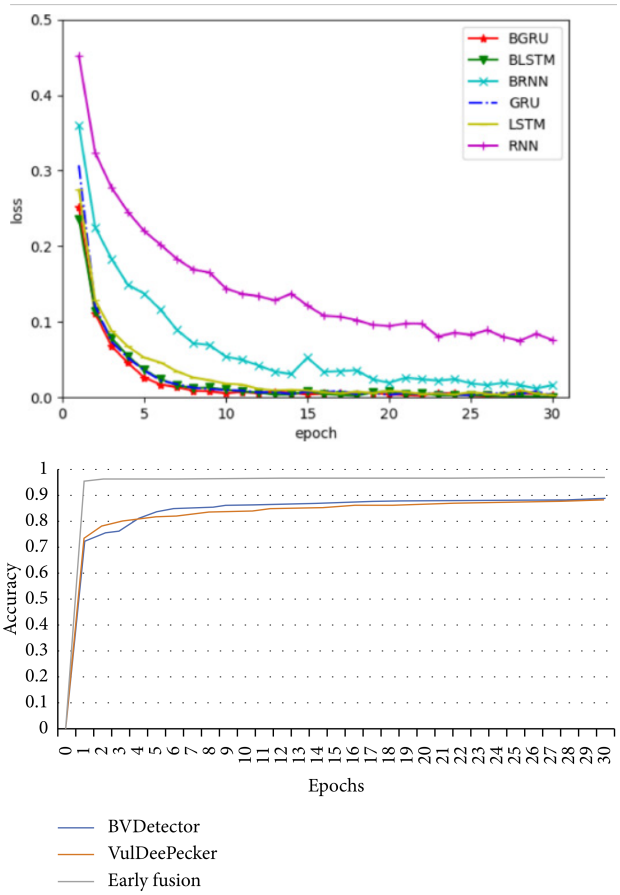
$$\mathrm{F1} = \frac{2 \cdot P \cdot (1 - \mathrm{FNR})}{P + (1 - \mathrm{FNR})}.$$

Expected Results and Future Scope

A comprehensive evaluation was conducted on our Bidirectional Gated Recurrent Unit (BGRU) model, comparing its performance with other models across various earlier mentioned Model Comparison Metrics.

The performance of the BGRU model (trained on Assembly Code) was compared with the Vuldeepecker model (trained on Source Code). Focus was given to the accuracy metric, and its variation against epochs was plotted. This comparison aimed to assess the effectiveness of our model in vulnerability detection in assembly code compared to source code.

Model	FPR	FNR	A	P	F1	MCC
LR	2.0	45.5	92.1	80.8	65.1	62.5
MLP	2.0	37.3	93.1	82.1	71.1	68.1
DBN	2.0	44.0	91.6	82.1	66.6	63.5
CNN	2.0	17.9	95.7	85.6	83.8	81.4
LSTM	2.0	21.7	95.2	85.2	81.6	79.0
GRU	2.0	17.6	95.7	85.7	84.0	81.7
BLSTM	2.0	15.7	96.0	86.2	84.3	83.0
BGRU	2.0	14.7	96.0	86.4	85.8	83.7



Looking forward, an improvement in model accuracy can be achieved by combining both source code and assembly code. This entails utilizing insights from both code types to enhance the model's ability to identify vulnerabilities. With this approach, a wider range of coding situations can be comprehended, and potential security risks can be identified more effectively. Additionally, exploration of intelligent methods to leverage existing knowledge in source code models is encouraged, thereby strengthening the vulnerability detection system and making it adaptable to various code types.

References

1. CVE (Common Vulnerabilities and Exposures) `https://cve.mitre.org/`
2. E. Stepanov and K. Serebryany, "MemorySani-

tizer: Fast Detector of Uninitialized Memory Use in C++," in Proceedings of the 2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pp. 46–55, San Francisco, CA, USA, March 2015. https://ieeexplore.ieee.org/document/7054176

3. Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "SysEVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities," IEEE Transactions on Dependable and Secure Computing, vol. 2021, Article ID 3051525, 1 page, 2021. https://ieeexplore.ieee.org/document/9478262
4. M. Vimpari, "An Evaluation of Free Fuzzing Tools," University of Oulu, Oulu, Finland. https://www.oulu.fi/sites/default/files/content-groups/cs/publications/Thesis_Vimpari.pdf
5. S. M. Ghaffarian and H. R. Shahriari, "Software Vulnerability Analysis and Discovery Using Machine-Learning and Data-Mining Techniques," ACM Computing Surveys, vol. 50 https://dl.acm.org/doi/10.1145/3057277
6. S. Kim, S. Woo, H. Lee, and H. Oh, "Vuddy: A Scalable Approach for Vulnerable Code Clone Discovery," in Proceedings of the 2017 IEEE Symposium on Security and Privacy, pp. 595–614, San Jose, CA, USA, June 2017. https://ieeexplore.ieee.org/document/7993666
7. J. Jang, A. Agrawal, and D. Brumley, "Redebug: Finding Unpatched Code Clones in Entire OS Distributions," in Proceedings of the 2012 IEEE Symposium on Security and Privacy, pp. 48–62, San Jose, CA, USA, July 2012. https://ieeexplore.ieee.org/document/6234431
8. N. H. Pham, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Detection of Recurring Software Vulnerabilities," in Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, pp. 447–456, Antwerp, Belgium, September 2010. https://ieeexplore.ieee.org/document/5677923
9. G. Grieco, G. L. Grinblat, L. Uzal, S. Rawat, J. Feist, and L. Mounier, "Toward Large-Scale Vulnerability Discovery Using Machine Learning," in Proceedings of the 6th ACM Conference on Data and Application Security and Privacy, pp. 85–96, New Orleans, LA, USA, March 2016. https://dl.acm.org/doi/10.1145/2857705.2857730
10. A. Younis, Y. Malaiya, C. Anderson, and I. Ray, "To Fear or Not to Fear That is the Question: Code Characteristics of a Vulnerable Function with an Existing Exploit," in Proceedings of the 6th ACM Conference on Data and Application Security and Privacy, pp. 97–104, New Orleans, LA, USA, March 2016. https://dl.acm.org/doi/10.1145/2857705.2857716
11. Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable Graph-Based Bug Search for Firmware Images," in Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 480–491, Vienna, Austria, October 2016. https://dl.acm.org/doi/10.1145/2976749.2978343
12. J. Tian, W. Xing, and Z. Li, "BVDetector: A Program Slice-Based Binary Code Vulnerability Intelligent Detection System," Information and Software Technology, vol. 123, Article ID 106289 https://www.sciencedirect.com/science/article/pii/S0950584920300577
13. G. Papandreou, A. Katsamanis, V. Pitsikalis, and P. Maragos, "Multimodal Fusion and Learning with Uncertain Features Applied to Audiovisual Speech Recognition," in Proceedings of the 9th Workshop on Multimedia Signal Processing, pp. 264–267, Chania, Crete, Greece, October 2007. https://ieeexplore.ieee.org/document/4450053
14. S. Bedoya and T. H. Falk, "Laughter Detection Based on the Fusion of Local Binary Patterns, Spectral and Prosodic Features," in Proceedings of the 18th International Workshop on Multimedia Signal Processing, pp. 1–5, Montreal, QC, Canada, September 2016. https://ieeexplore.ieee.org/document/776
15. A. Hovsepyan, R. Scandariato, W. Joosen, and J. Walden, "Software Vulnerability Prediction using Text Analysis Techniques," in Proceedings of the 4th International Workshop on Security Measurements and Metrics, pp. 7–10, Lund, Sweden, September 2012. https://ieeexplore.ieee.org/document/6336254
16. Y. Pang, X. Xue, and A. S. Namin, "Predicting Vulnerable Software Components through N-gram Analysis and Statistical Feature Selection," in Proceedings of the 2015 IEEE 14th International Conference on Machine Learning and Applications, pp. 543–548, Miami, FL, USA, December 2015. https://ieeexplore.ieee.org/document/7364057