Applied Mathematics and Computational Intelligence
Volume 15, No. 2, 2026 [132-143]

Software Vulnerability Assessment and Risk Identification Using Recurrent Nural Network

Ali Hussein[1], Azri Azmi[2], Hafiza Abas[3]

[1,2,3]Faculty of Artificial Intelligence, Universiti Teknologi Malaysia (UTM), Kuala Lumpur, Malaysia

*Corresponding author: hussein.a@graduate.utm.my

Received: 9 May 2025
Revised: 28 May 2025
Accepted: 8 July 2025

ABSTRACT

Identifying software vulnerabilities is an effective method for improving technological quality and optimizing testing management by enabling the early detection of deficiencies in simulation models before the actual testing phase begins. These predictive insights assist technology developers in efficiently allocating resources to the components most susceptible to flaws. This study introduces a vulnerability prediction model utilizing a Deep Learning (DL) approach, and a Recurrent Neural Network (RNN) is employed to classify source code, incorporating various soft computing techniques. Several preprocessing and data filtration methods have been applied to ensure data normalization and balance. To create a Vector Space Model (VSM), Term Frequency-Inverse Document Frequency (TF-IDF) and relationship-based feature extraction techniques have been implemented. The classification process is conducted using RNN on both training and validation datasets. The evaluation of performance of proposed work is performed using various real-time and synthetic accessible databases. It is observed from the experimental results that the proposed framework performs better when different datasets are evaluated.

Keywords: Software Vulnerability, RNN, Deep Learning, TF-IDF, Security, Machine Learning.

1 INTRODUCTION

The recognition of vulnerabilities in source code or software is a popular field of research throughout the software field [1]. In spite of the fact that existing research has demonstrated the usefulness of utilizing different detection techniques, models, and software flaw assessment tools, improve the effectiveness of detection tools and models is an essential challenge for researchers. Yearly, numerous security vulnerabilities are discovered in virtual tools, with many being publicly disclosed in general vulnerability databases or hidden within obfuscated code [2]. The threats arise in less obvious ways, making them difficult for programmers and code reviewers to detect. There is a strong motivation to analyze the patterns of these vulnerabilities directly from raw data [3]. In this work, using deep learning, security technology is proposed. Inspired by the progress of the design in these areas of study, a conceptual framework is utilized to investigate the practicability of the design within the context of the area of vulnerability identification. The research provides an overview of the static analytics software and discusses the Deep Neural Network (DNN) based conceptual perspective for detecting attacks.

In order to address the disparity between domains, it is suggested that a potential solution might involve treating each feature within a program in the field of computer vision as a neural network. This approach would enable fault detectors to determine the security status of a feature and provide comprehensive details regarding the positions of vulnerabilities. Thus, a finer-grained interpretation of fault management programs are needed. Certain exceptions exist in this analysis: most program comments may not exhibit any uncertainty, suggesting that only a limited number of instances are affected. Additionally, multiple comments are not necessarily considered as a semantically connected unit.

This study presents an innovative approach that utilizes active tracking with deep learning to automatically extract features for predicting weaknesses in runtime environments. In source code, interdependent components may be distributed across different sections, requiring specific code tokens to appear together due to program configurations or API usage requirements (e.g., Lock() and activate()). However, these dependencies are not always explicitly enforced. The extracted syntactic features accurately represent both the semantic meaning of code symbols and the structural organization of source code. This method removes the necessity for conventional feature selection techniques, which can be labor-intensive and time-consuming. Finally, extensive testing on a large repository of applications demonstrates that the proposed methodology achieves high accuracy in identifying code vulnerabilities.

2 LITERATURE SURVEY

Traditional programming utilizing k-means cluster analysis and the generative adversarial model, the proposed method according to Terada and Watanobe [1] for detecting bugs in large sets of randomly generated code using k-means clustering and a generative adversarial model. Their approach utilizes k-means clustering for code transformation, integrating an interactive analysis framework and software code refactoring techniques to optimize code selection. Tai et al. [2] introduced a system designed for object-oriented code analysis. In this model, a Long Short-Term Memory (LSTM) network is structured as a tree, enabling it to analyze conceptual relationships between coding pairs and assess sentiments within the code. Pedroni et al. [3] conducted a review on the communication patterns within large-scale source code compilation. Their study aimed to assist novice developers in identifying errors and understanding corrective measures. By examining historical messaging patterns in coding environments, they explored how specific code snippets contribute to vulnerabilities. Saito and Watanobe [4] proposed an optimal learning capacity map to enhance language learning models. Their approach focused on identifying the highest distribution improvements and utilizing key characteristics as input for system testing. Teshima [5] introduced a method for detecting software vulnerabilities using LSTM. To improve classification accuracy and optimize the model's performance, the number of LSTM neurons was adjusted. Their findings demonstrated that the LSTM model effectively detects system software bugs with high accuracy.

Through system software bug prediction models, and an attention-based RNN was introduced by Fan et al. [6]. The model performance measures were often the F-measure performance and the coefficient of determination. This model has strengthened the classification approach of the data files. Accordingly, the F-measure performance and the area under curve were 14% and 7% higher than the province versions. A software source code classification algorithm that utilizes Convolutional Neural Network (CNN) was suggested by Ohashi et al. [7]. Dependent on the specific algorithm throughout the code, the model segregates the source code. All reference codes are translated into a

basic code configuration during the CNN classification process without any parameters, features, or phrases. The reliability of the CNN model for identification is very strong. A hybrid strategy to optimize the mechanism of identification was suggested by Zhou et al. [8]. The strategies for data and text analytics were merged in this research investigation. In the beginning, the overview of the bug report was obtained by utilizing text analytics, and this was followed by its categorization into phases. Then, other frameworks and functions were collected and provided to machine learner at the second level. To association of these two steps, numerous data grafting strategies were used. the bug report has generated based on the supplied test dataset. Jin et al. [9] concluded with improved returns using the same sample and used it in an original assessment. Compared to a text and schema of the standard bug intensity report, utilizing Naive Bayes (NB) was developed. The use of standard bugs and additional fields was not factored in other research. F-measures of both the Eclipse and Mozilla configurations were measured to be 80% and 83%. In order to characterize the defects into extreme as well as non-severe groups, various machine learning architectures such as NB, Linear Regression, (SVM), Random-Forest (RF), including k-nearest Neighbours (K-NN) were implemented. The researchers indicated that perhaps the classifier's output depends mostly on subset and achieves reliability in the 75 to 83% range.

In their analysis of the NASA dataset, Goseva et al. [10] demonstrate both supervised as well as unsupervised classification techniques. In all approaches, the TF-IDF, TF and Bag of Word (BoW) occurrence feature vector techniques have been utilized to generate the feature vectors. In addition, Various classification algorithms, like SVM, KNN, RF, and Decision Tree, are used classification approach. The findings showed that the managed strategy worked better than unsupervised algorithms. Kukkar et al. [11] suggested a technique to enhance the K-NN classification model's effectiveness by using TF-IDF with bi-gram matrix factorization methods. For testing and training results, the research assessed three additional fields with both the textual field. While using the F-score, the efficiency was calculated. The output was found to rely on the repository. The estimation of software defects is a useful approach to minimize testing costs for the software component. By supervised learning from fault datasets of the previous version, it can effectively classify defect-prone software modules. According to Tong et al. [12] It is possible to divide current SDP studies through four categories: identification, correlation, guidelines of the mining organization and rating. First-class experiments use categorized architectures (also referred to as classifiers) as predictive models to classify software systems into deficiency classes and non-defect classes or different traumatic experiences of defects. The training discrepancy we concentrate on is focused on the analysis of the classification problem in this system. A significant number of tools have been proposed to solve challenging to understand. Most of these techniques are grouped into two categories: the amount of data and stage of the algorithm. The data augmentation methods primarily research the impact of adjusting classes' distribution to deal with extracted features. implementing a preprocessing mechanism to rebalance classification performance is probably a positive approach. The primary benefit of research methods is that they are autonomous of the classification model. In addition, data level approaches could be easily implemented as application-level approaches in distributed machine learning. Numerous existing researchers have used much soft computing and machine learning approaches in these frameworks to predict failure structures and lower software construction and preservation time. For JavaScript use a hybrid, compressed analysis, Wei et al. [13] introduced blended taint evaluation. The collection of data was performed for even those difficult conditions to dynamically evaluate by implementing a static response. A static network, which also incorporates a request builder, is perpetuated with complex results. The formidable challenge calls are used in this call-graph builder module. In pure topology optimization, they caught up with the

WALA instrument to create a static call-graph. The proposed method works similarly as previously stated and encourages external call diagram builder resources to be used in the research flow.

To determine the essential elements likely to be caused by a shift in technology, Musco et al. [14] utilized three kinds of call-graphs. However, to determine the effect of a shift in the software, they used feedback on different. The same approach should be used, and with a small modification, it may add a vulnerability or mitigate a vulnerabilities transformation rather than using an unpredictable change. Munaiah et al. [15] through the method, "Proximity" but "Dangerous Walks" metrics have presented new novel patch management measurements. The call-graph interpretation of the system distinguishes all of these. Their current studies have shown that utilizing their measurements to construct a forecasting model can help forecast more accurately because their measurements are statistically linked to susceptible components. A novel paradigm to detecting influence weaknesses called VGDetector has been developed by Cheng et al. [16]. In a combined analysis, the latest graph fully convolutional neural network framework is used to embed source code.

The fully automated way to map weaknesses to development tools and an interpersonal therapy Vulture was introduced by Kazi et al. [17] which can generate determinants in a new module to predict weaknesses. They found that dependencies and calls for features affect whether or not a part is susceptible. They also performed a Mozilla compiled code assessment that revealed that their methodology is precise. Lee et al. [18] proposed a method to produce semantic fingerprints from intrusion prevention programs. The call-graph, termed the software graph was retrieved of both the API call pattern created by malicious. For the conceptual signature, this diagram is used. Even when the malware is confused, or the brief software moment from its older iterations, semantic fingerprints allow users to upload. Arthur et al. [19] proposed a request algorithm to estimate and improve survival in a specified number, is probably the most comparable to our research. The connection between their measurements and many forms of bugs has been examined. It is observed that there is a connection in software engineering between call-graph dependent measurements and bugs. The frequency response machine learning methods were compared to J84 and SMO, and tested deep learning methods to identify possible bugs in the program. It has also concentrated on various source code metrics and also introduced pairing metrics for a so composite request. Pu et al. [20] suggested a source code corrections approach based on LSTM by utilizing returns generated similarities. For both the code major revision, the research utilized the pattern (seq2seq) neural network architecture with machine learning activities. White et al. [21] suggested a deep RNNs-based computer language model. The extensive experiments have shown that in a Java repository, the model exceeds conventional transfer learning called n-gram and directory listing n-gram. The software learning algorithm in the context of software development looks very promising. Terada et al. [22] proposed an LSTM-based methodology for software training in which the framework, by analyzing unfinished, raw data, predicts the following term. When developing a complete program, software engineers usually begin the process from the very start. The model suggests the next term finish a program in order to support everyone. The Classifier model achieved a high level of predictive power.

In summary, various promising methodologies have been proposed for application vulnerability detection. Many researchers have employed conventional classifiers, Recurrent Neural Networks (RNNs), or Convolutional Neural Networks (CNNs) for these tasks. While RNNs outperform traditional models such as n-gram models, they face challenges in handling long-sequence data. An enhanced version of RNNs has been developed to overcome these limitations. The model presented in this study integrates an awareness-driven approach, utilizing the RNN for evaluating and

identifying source code vulnerabilities based on a predicted upper bound. Despite relying on estimates from the most recent hidden state, RNN is outperformed by the AttM network. However, for prediction tasks, RNN takes into account all preceding hidden state effects. Most studies have employed various system software classification models for fault detection, functional programming identification, archival code analysis, and basic error detection. The proposed framework explicitly detects logical, grammatical, and other system-related software errors. Furthermore, instead of merely identifying errors, the model predicts the appropriate terms to replace them.

3 PROPOSED SYSTEM DESIGN

A system overview describing the flow of the execution procedure as shown in Figure 1. In the initial stages, a dataset consisting of many different software codes is taken into consideration. These programs comprise a wide range of processes and functions. The data set has been handled by Natural Language Processing using certain basic methods, including tokenization, which has resulted in the data being separated into individual words. The Porter stemmer algorithm was utilized in order to extract features, and ultimately, a filtration strategy was applied in order to get rid of instances that had been classified incorrectly or null values.

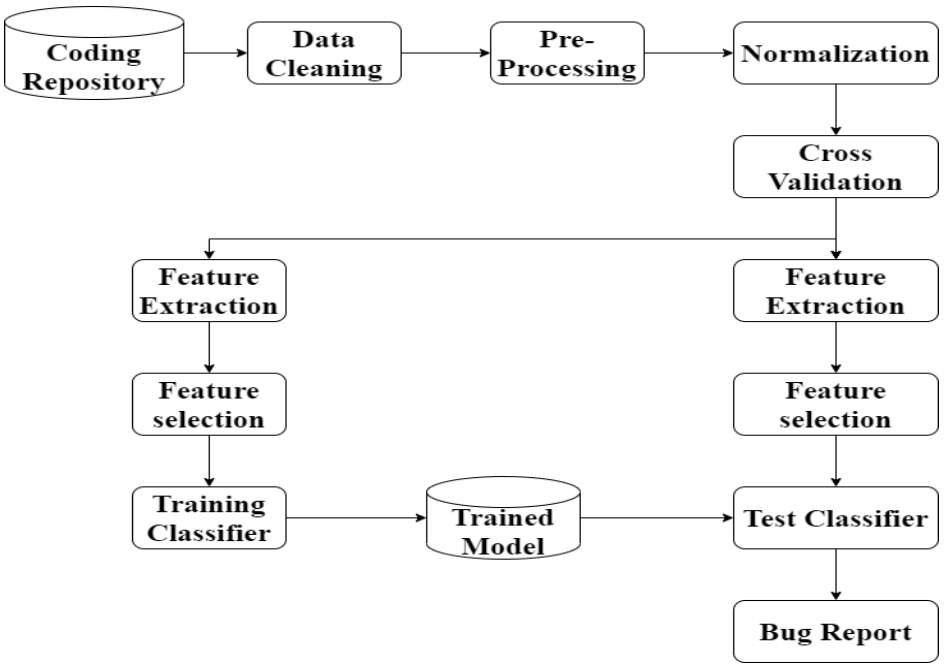

Figure 1: Proposed System Architecture Design

The TF-IDF features have been extracted by considering the density of relevant tokens; this method for obtaining features is employed in both the training and testing phases, correspondingly. The vector space model has developed with the objective of feature selection and enhancing with data acquisition in order to obtain the best feature possible from the model's vector space. There are three distinct machine learning algorithms that have been demonstrated during the training and testing. After the training has been completed, the system will begin to generate some background knowledge in accordance with a method of supervised learning. Utilizing this method, it is possible

to test data sets on a variety of platforms and improve detection accuracy for different types of data. For detecting the Vulnerability across the full dataset RNN classifier is utilized. The selection of Recurrent Neural Network (RNN) and Term Frequency-Inverse Document Frequency (TF-IDF) in this study is grounded in their respective strengths in handling sequential and text-based data. RNNs are particularly well-suited for learning dependencies across time steps or token positions, making them ideal for analyzing source code, where the order and context of tokens significantly influence vulnerability patterns. Unlike traditional machine learning algorithms, RNNs can remember and use long-term dependencies, which are often essential in modeling complex software behavior and detecting latent vulnerability patterns spread across large codebases. TF-IDF, on the other hand, was chosen for its robustness in quantifying the importance of individual code tokens within a corpus, allowing the model to focus on the most informative features while reducing noise from less relevant terms. This combination ensures both effective feature extraction and sequence-aware classification, leading to improved vulnerability detection performance compared to baseline methods. the dataset used in this study comprises a mixture of benign and malicious mobile applications. Malicious samples were sourced from publicly available repositories such as GitHub, ExploitDB, which are widely recognized in malware analysis research. Benign applications were collected from verified apps on the applications Store. Each mobile application was decompiled to extract readable files and code components necessary for analysis. During preprocessing, permissions, intents, and other static features were parsed from those files. Redundant or irrelevant attributes were filtered, and feature vectors were structured using techniques such as TF-IDF for textual representation. Data was normalized where necessary to ensure uniformity across samples. To balance the dataset, techniques such as undersampling or oversampling were considered depending on observed class distribution. the code is transformed into feature vectors using preprocessing techniques, enabling accurate analysis and metric-based evaluation.

The preprocessing pipeline consists of several steps: (1) raw code is cleaned by removing comments and formatting inconsistencies; (2) tokenization is performed using standard NLP tokenizers adapted for code structure; (3) stemming is applied; (4) TF-IDF is used to convert token sequences into numerical feature vectors. These vectors are then passed to the classification model. Each step sequentially refines the data: cleaned code feeds into tokenization, tokenized code feeds into vectorization, and vectorized data feeds into the RNN classifier. training and testing, the dataset was randomly split using an 70:30 ratio. The training set is used to learn model weights, while the testing set is used for final evaluation. The model is trained using TensorFlow and Keras in a Python environment. Evaluation metrics—including accuracy, precision, recall, and F1-score—were calculated using 10-fold cross-validation to ensure robustness and generalizability. The proposed Recurrent Neural Network (RNN) architecture includes five layers. TF-IDF was selected due to its lower computational complexity and interpretability when working with short code snippets. Unlike embeddings that require significant contextual training data, TF-IDF provides a lightweight, domain-agnostic feature extraction method suitable for static code. RNNs were chosen because they better capture the sequential nature. The performance of the proposed model was evaluated using standard classification, including accuracy, precision, recall, and micro-score as shown in Figure 2. Provide a well-rounded view of the model's ability to correctly detect and classify vulnerabilities.

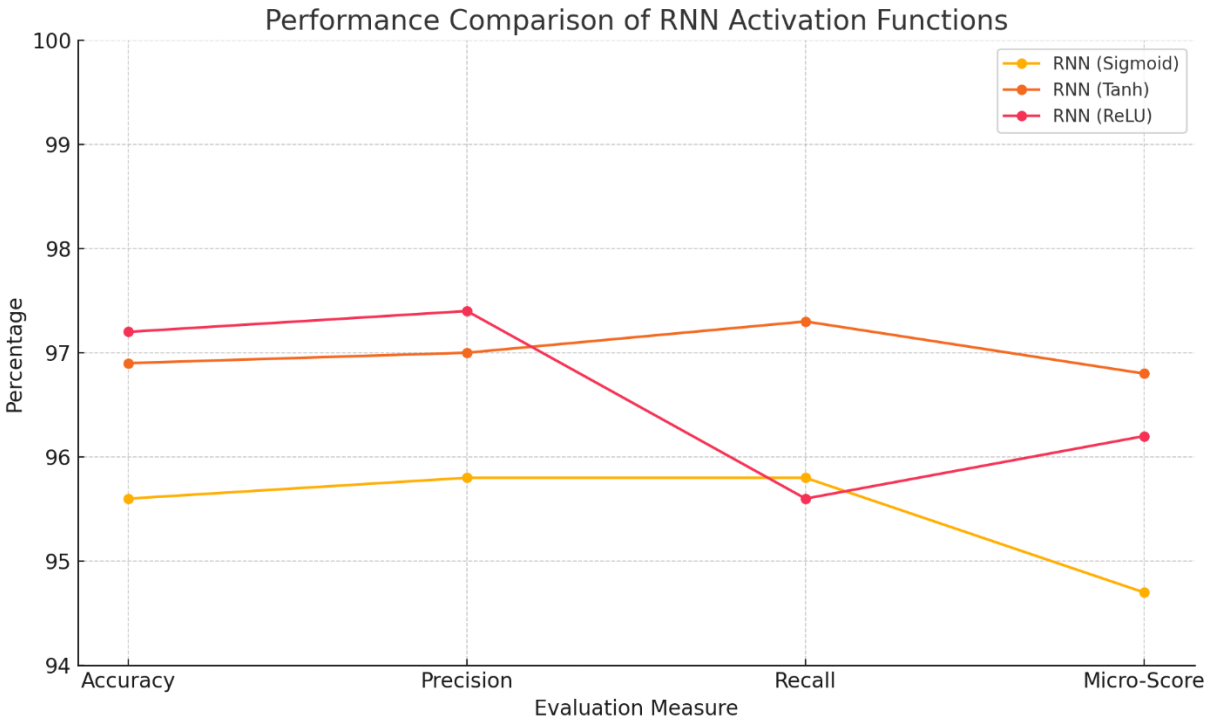

Figure 2: RNN Activation functions performance

4 ALGORITHM DESIGN

The algorithm which is described below have been used throughout the process of features extraction using natural language processing. However, the complete set of NLP features was insufficient for classification, thus these algorithms also generated the train modules. Certain attributes are extracted by utilizing the following algorithm, while certain NLP features provide an enhanced vector space for the selected features.

Algorithm: Term Frequency and Inverse Document Frequency

Input: Code line snippet which comprises of Term[1…..m]

Output: TF-IDF weight for every Term

Steps:

1. Data_Vec = {Tk1, Tk2, Tk3…. Tkm}

2. for every (Tr into Data_Vector)

Compute the Term_Frequency weight for Tr

3. Term_Frequency tf (tr,doc) = (tr,doc) tr=particular term doc= particular document in a term is to be found

4. idf = tr sum(doc)

5. return tf * idf

5 RESULTS AND DISCUSSION

The RNN classifier is used for fault anticipating, which comprises datasets, to validate the assessment of the recommended Vulnerability forecast method. confusion matrix, as shown in Table 1.

Table 1: Confusion matrix evaluation

Confusion-matrix		**Predicted**	
		Positive	**Negative**
Actual	**Positive**	True-Positive(TP)	False-Negative (FN)
	Negative	False-Positive (FP)	True-Negative (TN)

TN: It provides negative prediction for actual negative classes.

FP: It provides positive as prediction to every negative classes.

FN: It provides negative to every positive classes.

TP: It provides positive prediction for positive classes.

The accuracy of predictions is measured as the proportion of times the prediction was correct out of the total number of instances it was utilized.

Accuracy = $\frac{TP+TN}{TP+TN+FP+FN}$

The performance evaluation of the proposed system was carried out utilizing the proposed classification strategies with ten different synthetic datasets. The following Table 2 provides an illustration of the number of instances of Vulnerability that have been accurately identified in each individual dataset, together with the proportion of Vulnerability records that are present in each dataset.

Table 2: Accuracy Analysis

Dataset Name	Accuracy Rate
Intercafe [1]	77.80%
Synapse-1.0 [2]	66.45%
Berek [5]	88.18%
Intercafe [7]	80.68%
Termo [9]	77.5%
Pbean2 [13]	83.46%
Safe [16]	88.07%
Apache [18]	86.92%
Forrest-0.8 [19]	92.12%
Camel1.6 [21]	90.91%

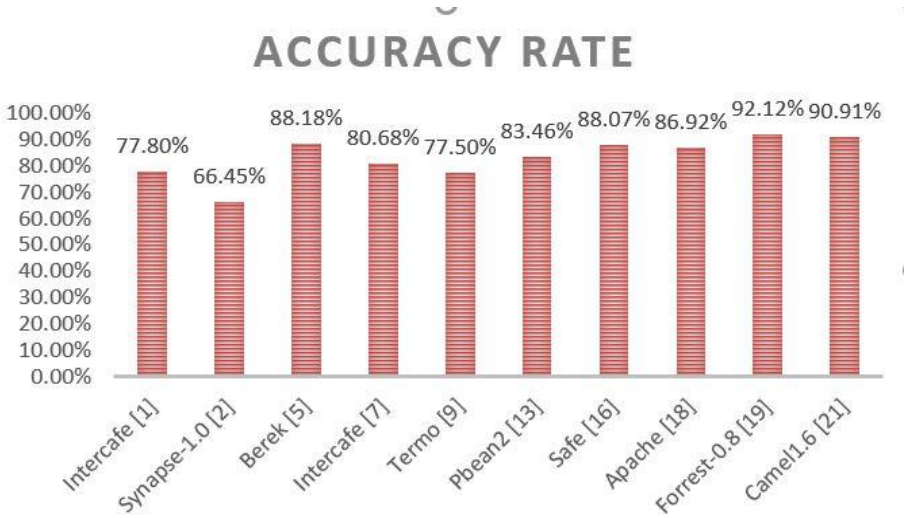

Figure 3: Detection accuracy of various existing systems with different datasets

As shown in Figure 2 and Figure 3, It is observed from the experimental findings that the Vulnerability predictive performance using RNN classifier is of high accuracy. Due to the limitations imposed by the data size, several datasets fail to adequately capture actual software problems and new vulnerabilities. When implemented in actual software programs, the approach that has been offered can be evaluated with additional testing.

6 CONCLUSION AND FUTURE WORK

The identification of vulnerabilities in imbalanced source codes is an exhausting and time-consuming task. Vulnerable code makes it possible to execute software attacks on remote users. At times, susceptible code develops internal attacks such as session hijacking, buffer overflow, bypass

authorization, and so on while it is being executed. Sometimes flaws manifest themselves in subtle ways that software testers are unable to uncover or opt to overlook. Constructing predictive models to detect vulnerabilities is a common application of machine learning methods, which are a prominent way. The proposed methodology demonstrates the data-driven approach to identify flaws through the application of deep learning technique along with a potential fix. The system is capable of working with a variety of datasets to acquire features and identify vulnerabilities. RNN provides a classification that is superior to that provided by conventional ML classifiers. The study effectively demonstrates that employing RNN classifiers, particularly with activation functions such as ReLU, sigmoid, and Tanh, significantly enhances the accuracy and reliability of vulnerability detection in mobile apps. The integration of TF-IDF and feature selection methods also contributes to a more robust classification system. These findings offer a practical pathway for applying the proposed model to real-world mobile application security audits. While the proposed RNN-based detection framework demonstrates high accuracy in identifying software vulnerabilities, certain limitations must be acknowledged. the model's performance is influenced by the quality and representativeness of the dataset; if the training data does not capture the full spectrum of real-world malware behaviors or coding practices, the generalization ability may be constrained. Additionally, deep learning models like RNNs require considerable computational resources during training, which might limit scalability in resource-constrained environments. The model's reliance on static features such as permissions and manifest data also means that it may struggle to detect sophisticated malware that uses dynamic obfuscation or runtime behavior changes. Addressing these limitations in future work could further enhance the robustness and applicability of the system.

References

[1] K. Terada and Y. Watanobe, "Automatic generation of fill-in-the-blank programming problems," in 2019 IEEE 13th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC), Singapore, Oct. 1–4, 2019, doi: 10.1109/MCSoC.2019.00034.

[2] K. S. Tai, R. Socher, and C. D. Manning, "Improved semantic representations from tree-structured long short-term memory networks," in Proc. 53rd Annual Meeting of the Association for Computational Linguistics and 7th International Joint Conference on Natural Language Processing (ACL-IJCNLP), Beijing, China, Jul. 26–31, 2015, doi: 10.3115/v1/P15-1150.

[3] M. Pedroni and B. Meyer, "Compiler error messages: What can help novices?" in Proc. 39th SIGCSE Tech. Symp. Comput. Sci. Educ., Portland, OR, USA, Mar. 12–15, 2008, doi: 10.1145/1352322.1352192.

[4] T. Saito and Y. Watanobe, "Learning path recommendation system for programming education based on neural networks," Int. J. Distance Educ. Technol., vol. 18, pp. 36–64, 2020, doi: 10.4018/IJDET.2020010103.

[5] Y. Teshima and Y. Watanobe, "Bug detection based on LSTM networks and solution codes," in Proc. 2018 IEEE Int. Conf. Systems, Man, and Cybernetics (SMC), Miyazaki, Japan, Oct. 7–10, 2018, doi: 10.1109/SMC.2018.00599.

[6] G. Fan, X. Diao, H. Yu, K. Yang, and L. Chen, "Software defect prediction via attention-based recurrent neural network," Scientific Programming, vol. 2019, pp. 1-14, 2019, doi:

10.1155/2019/6230953.

[7] H. Ohashi and Y. Watanobe, "Convolutional neural network for classification of source codes," in Proc. 2019 IEEE 13th Int. Symp. Embedded Multicore/Many-core Systems-on-Chip (MCSoC), Singapore, Oct. 1–4, 2019, doi: 10.1109/MCSoC.2019.00035.

[8] Y. Zhou, Y. Tong, R. Gu, and H. Gall, "Combining text mining and data mining for bug report classification," J. Softw. Evol. Process, vol. 28(3), pp. 150-176, 2016, doi: 10.1002/smr.1770.

[9] K. Jin, A. Dashbalbar, G. Yang, J.-W. Lee, and B. Lee, "Bug severity prediction by classifying normal bugs with text and meta-field information," Adv. Sci. Technol. Lett., pp. 19-24, 2016, doi: 10.14257/ASTL.2016.129.05.

[10] K. Goseva-Popstojanova and J. Tyo, "Identification of security-related bug reports via text mining using supervised and unsupervised classification," in Proc. IEEE Int. Conf. Software Quality, Reliability and Security (QRS), Lisbon, Portugal, Jul. 16–20, 2018, doi: 10.1109/QRS.2018.00047.

[11] A. Kukkar and R. Mohana, "A supervised bug report classification incorporating textual field knowledge," Procedia Computer Science, vol. 132, pp. 352-361, 2018, doi: 10.1016/j.procs.2018.05.194.

[12] H. Tong, B. Liu, and S. Wang, "Software defect prediction using stacked denoising autoencoders and two-stage ensemble learning," Information and Software Technology, vol. 96, pp. 94-111, 2018, doi: 10.1016/j.infsof.2017.11.008.

[13] S. Wei and B. G. Ryder, "Practical blended taint analysis for JavaScript," in Proc. Int. Symp. Software Testing and Analysis (ISSTA), Lugano, Switzerland, Jul. 15–20, 2013, doi: 10.1145/2483760.2483788.

[14] V. Musco, M. Monperrus, and P. Preux, "A large-scale study of call graph-based impact prediction using mutation testing," Software Quality Journal, vol. 25, pp. 921-950, 2017, doi: 10.1007/s11219-016-9332-8.

[15] N. Munaiah and A. Meneely, "Beyond the attack surface: Assessing security risk with random walks on call graphs," in Proc. ACM Workshop on Software Protection, Vienna, Austria, Oct. 24–28, 2016, doi: 10.1145/2995306.2995311.

[16] X. Cheng, H. Wang, J. Hua, M. Zhang, G. Xu, L. Yi, and Y. Sui, "Static detection of control-flow-related vulnerabilities using graph embedding," in Proc. 24th Int. Conf. Engineering of Complex Computer Systems (ICECCS), Hong Kong, China, Nov. 10–13, 2019, doi: 10.1109/ICECCS.2019.00012.

[17] K. Z. Sultana and V. Anu, "Using software metrics for predicting vulnerable classes and methods in Java projects: A machine learning approach," Journal of Software: Evolution and Process, Mar. 2021, vol. 33(3), e2303, doi: 10.1002/smr.2303.

[18] J. Lee, K. Jeong, and H. Lee, "Detecting metamorphic malwares using code graphs," in Proc. ACM

Symp. Applied Computing (SAC), Sierre, Switzerland, Mar. 22–26, 2010, doi: 10.1145/1774088.1774505.

[19] A.-J. Molnar and A. Neamţu, "Longitudinal evaluation of software quality metrics in open-source applications," in Proc. 14th Int. Conf. Evaluation of Novel Approaches to Software Engineering (ENASE), 2019, doi: 10.5220/0007725600800091.

[20] Y. Pu, K. Narasimhan, A. Solar-Lezama, and R. Barzilay, "SK_p: A neural program corrector for MOOCs," in Proc. ACM SIGPLAN Int. Conf. Systems, Programming, Languages and Applications: Software for Humanity (SPLASH), Amsterdam, The Netherlands, Oct. 30–Nov. 4, 2016, doi: 10.1145/2984043.2989222.

[21] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk, "Toward deep learning software repositories," in Proc. 12th Working Conf. Mining Software Repositories (MSR), Florence, Italy, May 16–17, 2015, doi: 10.1109/MSR.2015.38.

[22] K. Terada and Y. Watanobe, "Code completion for programming education based on recurrent neural network," in Proc. 2019 IEEE 11th Int. Workshop Computational Intelligence and Applications (IWCIA), Hiroshima, Japan, Nov. 9–10, 2019, doi: 10.1109/IWCIA47330.2019.8955090.