Journal of Information Systems Engineering and Management
2025, 10(10s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

Android Malware Detection and Classification with Analysing Permission API's using Recurrent Neural Network

Vijay Sonawane[1], Dr. Pratap Singh Patwal[2], Dr. Vinod S. Wadne[3]
[1]*Research Scholar (Computer Science Engineering), Nirwan University Jaipur, Rajasthan, India*
Sonawanevijay4@gmail.com
[2]*School of Engineering & Technology, Nirwan University Jaipur, Rajasthan, India*
Pratappatwal@gmail.com
[3]*HOD of Computer Engineering, JSPM's Imperial College of Engineering & Research, Pune, Maharashtra, India*
vinods1111@gmail.com

ARTICLE INFO

Received: 02 Nov 2024

Revised: 26 Dec 2024

Accepted: 04 Jan 2025

ABSTRACT

Android applications often contain vulnerabilities that can be exploited by malicious software, making automated malware detection is crucial. Since software vulnerabilities can have severe repercussions, a great deal of research and development effort has been put into developing mitigation strategies. Over the past few years, a number of different approaches have been tried in an effort to lessen the danger posed by vulnerabilities in software. The proposed approach involves multiple stages, including APK recompilation, feature extraction from manifest permissions, model training, testing, and performance analysis. The process begins with APK recompilation, where Android package files are decompiled using specialized tools to extract critical components such as manifest.xml, which contains permissions requested by the application. These permissions serve as key indicators of potential malicious intent. The extracted XML-based permissions and additional metadata are then transformed into structured feature vectors. Feature selection techniques are applied to retain the most relevant attributes, reducing noise and enhancing classification accuracy. The core classification module leverages Recurrent Neural Networks (RNNs) to analyze the extracted features and identify malicious patterns. The model undergoes supervised training using a labeled dataset comprising both benign and malicious APKs. During training, the RNN learns temporal dependencies and feature interactions, improving its ability to detect sophisticated malware. Once trained, the model is tested on a separate dataset to evaluate its classification performance. Various evaluation metrics such as accuracy, precision, recall, and F1-score are used to measure effectiveness. Finally the proposed model is compared with traditional machine learning classifiers and existing deep learning techniques. The results demonstrate that RNN-based classification significantly enhances malware detection accuracy due to its ability to process sequential patterns within application permissions. This research contributes to the field of Android security by presenting an efficient, scalable, and automated malware detection framework, leveraging deep learning for real-time classification of malicious applications.

Keywords: Feature Extraction, RNN, Android Malware, Malware Classification, Mobile Security, Android Security, Malware Detection

Introduction

The rapid evolution of mobile technology has led to a surge in the use of Android devices, which now dominate the global smartphone market. However, this widespread adoption has made Android an attractive target for malware developers, posing a significant threat to users' privacy, security, and financial assets. The open-source nature of Android, while fostering innovation, has also contributed to an ecosystem that is increasingly vulnerable to malicious attacks. Malware incidents not only compromise individual users but can also lead to large-scale disruptions in enterprise networks. Traditional malware detection methods, such as signature-based and heuristic approaches, struggle to keep pace with the sophistication of modern malware, which often employs obfuscation and evasion techniques. Consequently, there is a growing need for advanced detection systems that can analyze complex

behavioral patterns and identify malware with high accuracy. Machine learning, particularly deep learning, has emerged as a promising solution in this domain, offering robust and scalable mechanisms for detecting and classifying Android malware.

This research focuses on leveraging Recurrent Neural Networks (RNNs) to detect and classify Android malware by analyzing Permission APIs. Permissions requested by apps and their interactions with Application Programming Interfaces (APIs) are critical indicators of an app's intent and behavior. By extracting and analyzing these features, it becomes possible to differentiate between benign and malicious applications with higher precision. The primary objective of this study is to develop an efficient Android malware detection framework that utilizes RNNs to model sequential patterns in Permission API usage. The proposed approach aims to address the limitations of existing systems by enhancing detection accuracy, reducing false positives, and enabling real-time analysis. The findings of this research are expected to contribute significantly to the field of mobile security by offering an advanced and reliable method for combating Android malware threats. This manuscript describes literature survey in section 2, the research methodology describes in section 3 while results and discussion demonstrated in section 4. Finally, section 5 describes conclusion and future work proposed model.

Literature Survey

According to J. B. S et al. [1] Due to the rapid pace at which smartphones are evolving in the modern world, data protection has emerged as one of the most pressing concerns. When there is a lack of protection in the globe, it raises concerns for the safety of those who use mobile devices. Security is an essential component of human life. One of the biggest and most significant threats to the safety and security of cellphones is malware. Malware assaults on mobile devices are rapidly expanding in both sophistication and frequency. Since Android is an open-source platform and because it now dominates the market, malware developers consider it to be their top priority target. Cross-comparison is challenging to do because the most advanced methods for detecting mobile malware that have been published in the scientific community use a wide array of metrics and models. In this work, many existing approaches are contrasted with one another, and a considerable effort is made to quickly discuss android malwares, different techniques to identify android malwares, and to present a clear impression of the evolution of the android system and different malware detection classification algorithms.

According to M. Masum et al. [2] Over the past few years, Android has seen enormous growth in its popularity as an operating system (OS) for smart devices, and it is currently the most prevalent OS. Due to its widespread adoption and open nature, the Android operating system has grown into an alluring target for malicious software applications. These apps pose a significant risk to the safety of individuals, enterprises, and financial institutions. Traditional anti-malware defenses are no match for the sophisticated computer viruses that are constantly being developed. As a consequence of this, there is a growing demand for automated malware detection systems in order to limit the risks associated with harmful actions. In the past few years, machine learning techniques have been exhibiting encouraging results in identifying malware. The majority of the methods used in this classification are shallow learners, such as Logistic Regression (LR), however these techniques are showing promise in this area. As a result, architecture for DL that has been given the name Droid-NNet has been presented for the categorization of malware. The proposed method, Droid-NNet, on the other hand, is a deep learner that outperforms other cutting-edge machine learning algorithms that are currently in use. For the purpose of evaluating Droid-NNet, all of the studies are carried out on two datasets (Malgenome-215 and Drebin-215) of Android applications. The study's outcome demonstrates both the reliability and the efficiency of Droid-NNet.

M. N. M. Ahmad et al. [3] It is important to remember that ransomware is a notable sort of assault that is widely recognized for the danger it poses to consumers. This risk invariably results in significant disagreements, which are exactly what the field of cybersecurity works to avoid. The majority of the time, ransomware will encrypt or lock data on the personal computer or handheld device that it is focusing on, and then it requires payment in order to decrypt those files. In order to avert or mitigate the severity of this problem, it is essential to put into action some of the potential solutions. There have been several research papers written that have established specific tools and methods to either prevent attacks by ransomware before they emerge or detect them when they do occur; nevertheless, these tools and methods are typically compromised in a short amount of time. This survey's goal is to offer a comprehensive analysis of the latest developments in ransomware research, as well as approaches for detection and protection. The discussion that follows will, for the most part, center on three key concerns, all of which are investigated in turn via the perspective of risk administration: The manner in which can an OS like Android be made anticipating a

ransomware assault and defended against it before it even takes place; In the event that a user is victimized by ransomware, what measures can they take to thwart the assault or mitigate the collateral harm it causes while it is ongoing; In what way can an individual whose existence has been thrown off track as a result of ransomware attacks get themselves back on course after an attack. In the event that the solutions to these concerns are found, it is possible to take another step toward building an original strategy for mitigating the danger posed by Android ransomware. As a result of this, effectively controlling the risk will make it nearly difficult for an android ransomware assault to be effective. Even in the most catastrophic scenario, an individual will be able to retrieve the compromised data by referring to the backed copy.

According to P. Kotak et al. [4], users of mobile applications face a substantial risk of having their data stolen. The increasing significance of digitization is what encourages the wide variety of applications that are currently available. In this research, an innovative and lightweight strategy is presented for dividing Android apps into low-, medium-, and high-risk areas based on their potential vulnerabilities. The proposed method relies heavily on what are known as the "other permissions" of apps for Android which is also referred to as "hidden permissions." It has been suggested to use a method that relies on linear regression in order to divide the apps into their respective risk categories. It demonstrates how additional permissions can be utilized as a powerful signal for the purpose of classifying risks. The K-means clustering is applied in order to validate and explain the choice that was made about our strategy. The proposed method determines the risk associated with an app in a review with 500 apps and 101 additional permissions; the rationalization that is supplied for each detection reveals relevant aspects of the risk that was found.

According to Ahmad et al. [5] It is essential to have an Intrusion Detection System (IDS) in place on a smartphone in order to prevent imminent security breaches and to safeguard the privacy of users. Surveys conducted within the stock market indicate that Android is currently the most widely used mobile operating system. Because of this, the smartphone is an especially appealing target for prospective attacks. The creation of dangerous programs and gaming, the vast majority of which are readily accessible to users, is the source of the risk. The central processing unit (CPU), memory, and the utilization of their batteries are the primary issues with the IDS that were designed for smartphone technology. This is due to the fact that is restricted sources of energy for smartphones and other handheld devices. The second issue is that the vast majority of additional security tools, such as anti-malware and antiviruses have an ongoing requirement for upgrading their malware signatures from servers, and this process eats up a greater amount of energy than the first upgrade. While this is happening, hackers are employing novel methods of attack to break into cellphones, even though the server side does not have fingerprints for these types of attacks. In the current investigation, the primary concentration is placed on the effectiveness analysis of IDS for Smartphone transactions and interactions. In addition, a discussion is held on the implications of this research with the intention of enhancing the performance of existing intrusion detection systems. In-depth comparative research of the many existing methods of IDSs for the purpose of improving and enhancing smartphone privacy is described in this article. The purpose of this study is to highlight the benefits and drawbacks of the intrusion detection method that is currently being used for smartphones.

The research conducted by Z. Shan et al. [6] focuses on assessing and identification of such strategies as, for example, hiding the app or deleting traces which is referred to as Self Hiding Behavior" (SHB). In particular, it provided (1) an in-depth characterisation of SHB, (2) a set of static analysis methods for identifying such conduct, and (3) a collection of detectors that use SHB to differentiate among benign and malicious programs. It has been noticed that SHB can conceal an app's traces in a variety of ways, such as by restricting phone messages and calls or erasing messages and phone calls from logs. Concealing the existence or activity of the app is just one of these ways. When the suggested methods for static analysis are used to a huge dataset consisting of 9,452 Android apps (both benign and malicious), an average frequency of 12 of these SH behaviors are uncovered. The strategy that was recommended unearthed the fact that fraudulent programs, on average, make use of 1.5 SHBs per app. Unexpectedly SH conduct is also used by authorized apps (sometimes known as "benign" apps), which can have a negative impact on users in a variety of different ways. High accuracy as well as recall can be achieved with the method when it is used to distinguish between malicious and benign applications (the cumulative F-measure for the method is 87.19%).

C. Schindler et al. [7] proposes a solution to integrate free and open-source tools in order to assist programmers in verifying their app in order to ensure that it does not create any security risks as a result of using libraries developed by third parties. It is possible to undertake tests to discover data breaches caused by third-party apps by combining the application of tools such as FlowDroid, Frida, and mitm-proxy in a method that is both straightforward and

effective. The configuration and set up that is being recommended gives normal app developers the ability to protect the confidentiality of users without having to be committed security experts.

An Android-based client-side approach has been presented by L. J. Mwinuka et al. [8] for identifying the existence of bogus access points in a boundary by using details acquired from probe responses. The approach that has been suggested takes into account the distinction between security data and signal degree of an access point. The method of detection is shown in three different kinds of networks: open networks, locked networks, and networking with captive portals. In contrast to other efforts, the solution that is being suggested doesn't need root access in order to do detection. Additionally, it has been designed to be portable and to have improved performance. The technique was tested in both open as well as closed networks, and the findings showed that it could detect fraudulent access points with a detection rate of 99% and 99% at a mean of 24 and 7.7 milliseconds, correspondingly, in public networks.

The Talos application, which was developed by H. C. Takawale et al. [9], is a lightweight technique of analyzing malware. This method makes use of on-device ML and TensorFlow. It utilizes 'Requested Permissions' as the input variables with the intention of resolving the issue of malware detection. The identification procedure is carried out entirely on the mobile device, and it is not necessary to have an internet connection in order for it to function. TensorFlow is utilized in the process of developing the ML algorithm. Following the freezing of the model's graph in the protocol buffer format, it is subsequently exported so that it can be deployed on a handheld device. Talos has showed an accuracy level of 93.2% throughout the various testing. Even on less powerful smartphones with Android, it could do an analysis of hundreds of applications in a single second.

Machine learning is used in K. Kim et al.'s [10] proposal for a method that evaluates the Application Programming Interfaces (APIs) of mobile applications for Android in order to determine the risk of security associated with these programs. The primary concept behind the technique that has been presented is to use reverse engineering analysis to retrieve the APIs from the program's execution code of the program. This is done so that every single API can be contrasted with the detrimental API database that has been constructed using the already known malware dataset. The suggested system assigns a risk score to each application rather than merely deciding whether or not it is malicious or innocent. An ensemble of tree-boosting ML techniques is utilized in order to carry out this quantitative evaluation. An investigation is conducted with a collection of benign and malicious actual-world specimens in order to demonstrate the practicability of the suggested scheme. The findings of the study are contrasted with other schemes that are already in place. The findings of the experiments demonstrate that traditional methods based on Naive Bayes and basic ensemble algorithms operate less well and are less accurate than the experimental method.

The detection of Android malware has received a growing amount of attention in recent years due to the explosive proliferation of mobile malicious software, as stated by C. Zhang et al. [11]. Nevertheless, operating an Android malware detection tool that is hosted in the cloud typically results in expensive hardware as well as bandwidth costs. This conundrum is what drives us to devise a strategy for reusing an existing in-cloud image-classification NN for the purpose of identifying Android malware. The suggested approach, when presented with an Android application, first incorporates the application's features into a picture, then cleverly modifies the image that contains the embedded features, and finally inputs the amended picture into an in-cloud image classification method. The outputs of the classification methods are mapped into an outcome for detecting malware. Furthermore, two additional methods, known as perturbation hiding and group mapping, have been presented in order to enhance the effectiveness of detection while also lowering the possibility that repurposing behavior will be identified. Investigations have shown that human beings are not typically able to identify the perturbations, and our solution beats both conventional based detectors in terms of detection performance.

Research Methodology

In this section, the methodology behind vulnerability evaluation and bug cleanup is dissected down into its different elements briefly. The process of execution flow is depicted in its entirety in Figure 4.3, along with an explanation of how it interacts with other algorithms. At first, a dataset consisting of many different software programs is employed. These scripts comprise a variety of operations and functionalities. Natural Language Processing was employed along with some fundamental techniques to analyze the data set. Tokenization was then utilized in order to break the data down into its component terms. Another approach that has been utilized to get rid of stop words that already exist in software programs or processes is called stop word elimination. In order to acquire features, the Porter stemmer

method was applied, and then, as a last step, the filtration approach was utilized to get rid of occurrences that were classified incorrectly or had zero values. The entire architecture divides into 3 different parts as mentioned below;

- To develop a algorithm for data extraction and heterogeneous feature selection from unstructured data such as apk files
- To develop an algorithm for code risk identification as well as vulnerability parameter checking using s Machine learning algorithms.
- To design and develop algorithm for automatic risk identification using Deep Learning Algorithm

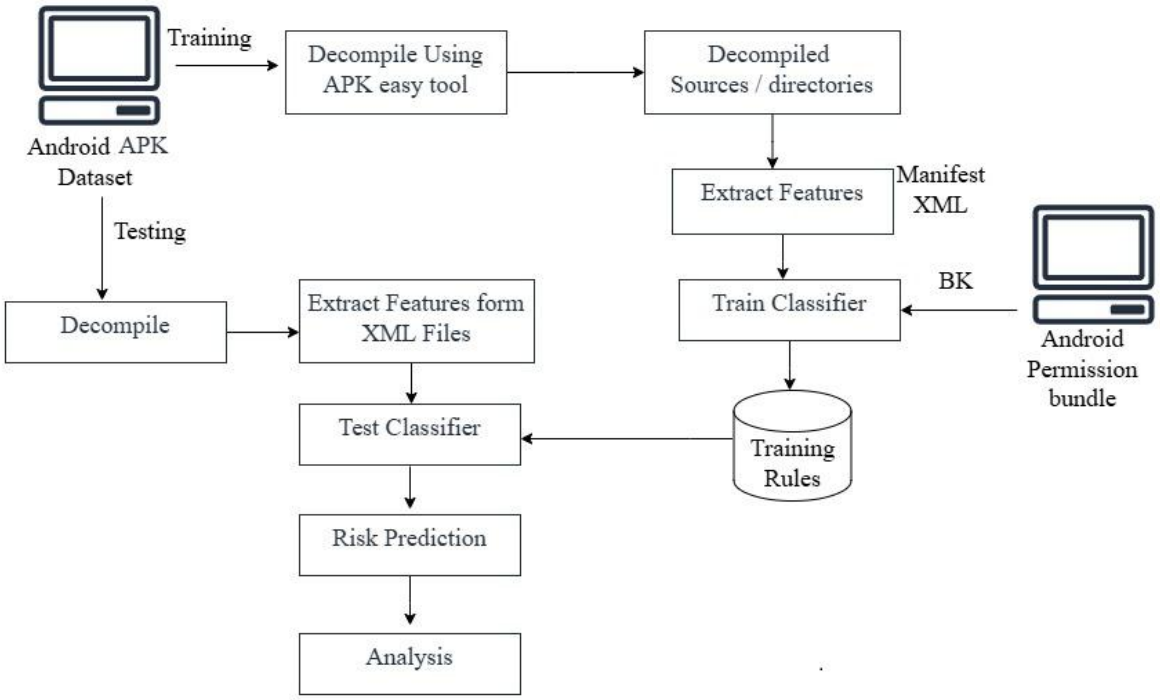

Figure 4.3: Framework of proposed software vulnerability detection using classification techniques

Both the training and the testing phases make use of the multiple features that were retrieved depending on the overall density of the appropriate tokens. The vector space framework has developed for the goal of choosing features and enhancing with data acquisition in order to obtain the most effective feature possible from the model's vector space. Both the training phase and the testing phases each make use of one of three distinct machine learning methods. After the training has been completed, the system will begin to generate a certain foundational information in accordance with a method of supervised learning. Utilizing this method, we have been able to test data sets on a variety of platforms and classify detection performance for heterogeneous data. RNN is a classification technique that was utilized for the purpose of detecting the bug across the full dataset utilizing the provided methods. In contrast, static analysis often necessitates a significant amount of computing time in order to achieve the desired level of efficiency, but dynamic evaluation can be as fast as the code's performance. The fundamental obstacle faced by research approaches for dynamic bug identification is a significant mistake rate in their performance. Complicated analytical techniques have the drawback of not being able to ensure a review of all the viable code snippets that ought to be executed. Due to this, the dynamic evaluation cannot be visually represented and is typically utilized to highlight the existence of programming risks. The efficient approaches for finding bugs go through a number of phases, which are outlined in the following paragraphs.

Data Collection:

- The data required for proposed module we used the android APK file.
- The normal malicious applications APK and malicious APK has downloaded from various sources.
- The normal APK contains normal permission while the malicious APK contains malicious requests.

APK Decomplication Module:

- The APK is not in readable due to in executable file format, so, it need to decompile.
- In this module different APK has decompile using APKEasy software tool.
- This tool basically converts .apk file into different directories with available class files which also contain manifest.xml, configuration file etc.

Feature extraction and selection: Compilation takes place on the entirety of the original code or components, which contain actual statistics. In this approach, the behaviour of code is analysed for the purpose of discovering vulnerabilities. In the course of the analysis, a total of four distinct approaches of dynamic analysis, including fault infusion, mutation appropriate beginning, dynamically taint evaluation, and dynamical system verify, were utilized in order to construct the vector Space Model from the characteristics that were extracted. Throughout the module training process, a variety of feature selection approaches have been utilized. In this approach, an aspect of code is evaluated for the purpose of vulnerability identification. The function that is being used for compiling the entire source code or modules using actual statistics. When examining a more comprehensive dataset, it is less important to take into account each of the variables; nonetheless, the level of complexity increases proportionally with the number of variables. As a consequence of this, it is frequently advisable to cut down on the total number of variables contained in a dataset and make use of essential variables. Utilizing an approach known as "Function Selection," each parameter may be narrowed down, and the numerical value of the variable can be extracted from a dataset.

- The feature extraction has been done for all APK based manifest.xml file where all permission has been mentioned by developer or code writer.
- The major objective behind extraction of those features to build a robust module using ML or DL algorithms.
- The various feature extraction techniques have been used to extract the selected features.
- The entire phase is considered as data pre-processing which deals with data acquisition, classified instance elimination, null value removal and systematic data sampling these techniques have used extract the unique features from input data set

Module Training: Monitoring and evaluating the environment, such as the registers and framework, the code is running in is used to analyse the consequent behaviour of the function based on every random input and scheduling option. In this configuration, each state stores the set status of the system. The matching input value and scheduling option are examined as counter instances if the process progresses to a state in which the setup is disrupted. This module has provided the guarantee that the module level source code does not contain any bugs. Using a various machine learning techniques and Recurrent Neural Network (RNN) module that has been trained for the system, Background Knowledge (BK) is produced in the appropriate manner.

- The feature extraction has been done during the data pre-processing phase and the selected features as used to train the entire module.
- RNN is the deep learning algorithm has used for train the module and generate background knowledge accordingly.
- The Train.arff and Test.arff has generated in above functional form, the arff file format used for re-classification using machine learning techniques of those extracted features data.
- In this module we also evaluated similar data using hybrid machine learning and that file has been generated for classification of machine learning techniques.
- T
- Recurrent Neural Network (RNN) module has been trained for system and generate Background Knowledge (BK) accordingly

Module Testing:

- This model basically works for classifying the test instances from the extracted feature vector, the outcome of this module provides the specific APK contains some malicious information and classify it normal or abnormal respectively.
- In detection of vulnerability of module a machine learning classification for both files, and it is a pure supervised classification method.

- The various machine learning classification algorithm have been used of classification. The hybrid machine learning is our major contribution for classification.
- The Weka 3.7 framework has used the utilization of both machine learning algorithms

Analysis: After completion of the entire execution finally, the system will illustrate the classification accuracy of proposed system as well as explore and validate the proposed system with some existing machine learning algorithms and proposed DL based RNN.

Vulnerability Detection : The detection of vulnerabilities has been carried out according to the features that have been retrieved from the data used for training set. RNNs, which include an algorithm for long-term and short-term memory, have been utilized in the categorization process. This type of detection is also useful for preventing assaults on software-as-a-service systems that run web apps. The exposed vulnerabilities in the code allow for illicit entry to be granted to individuals on the outside as well as for attacks to be launched against the system itself. The primary goal of identifying vulnerabilities is to detect the presence of handling exceptions and buffer overflow attacks while the code is being executed. The code snippet benefits from improved detection accuracy due to the method that was proposed.

- The vulnerability detection has been performed based on extracted features from the training data set.
- The vector space model has been generated according to extracted features such as relational features, and some bigram features.
- The classification has been done with recurrent neural networks, including long short-term memory algorithm.
- This detection is also effective for prevention of software-as-a-service attacks for web applications.
- The vulnerable code finds generation of internal as well as external attacks and grant un-authorized access to external users.
- The major objective of detection vulnerability is automatic detection of exception handling and buffer overflow attack during the code execution. The proposed algorithm provides better detection accuracy in the code snippet.

A clone finder ought to make an effort to search through the resource which contains the structure for snippets of code that can be used for evaluation. Only those portions of code that share a high degree of resemblance with the code that came before them should be identified as clones. The most important problem is the fact that it is currently unknown which areas of the code could possibly be altered. In point of fact, even after spotting parts that may have been cloned, more analysis and techniques may be necessary in order to differentiate the true clones from the imposters.

Algorithm Design

The given algorithm outlines the training process of a Recurrent Neural Network (RNN) for classification tasks. The process begins by taking in a training dataset, an array of activation functions, and a predefined threshold as inputs. The goal is to extract meaningful features that will be used in the trained model for classification. The first step involves setting up the input data block, defining the activation function, and determining the number of training epochs. These parameters help structure the learning process and optimize the network's performance.

Algorithm 1: RNN Training process for classification

Step 1: Define Input Parameters

- Let $d=\{d1,d2,...,dn\}$ be the input block of data, where represents individual data points.
- et $f(x)$ be the activation function (e.g., sigmoid, ReLU, etc.).
- Let E be the epoch size, which determines how many times the training process iterates over the dataset.

Step 2: Feature Extraction

We extract features from the input data:

Features.pkl←ExtractFeatures(d)

Mathematically, this can be represented as:

$$F=\phi(d)$$

where $\boldsymbol{\phi(d)}$ is a function that extracts meaningful features from the raw data $\boldsymbol{d}$. This could involve transformations like relational features, statistical moments, or any other feature engineering techniques.

Step 3: Feature Optimization

The extracted feature set FFF is optimized to remove redundant or irrelevant features:

$$Feature\_set \leftarrow optimized(Features.pkl)$$

we define an optimization function:

$$F'=\psi(F)$$

where ψ(F) is an optimization function that selects the most relevant features using techniques such as feature selection (e.g., LASSO, PCA) or feature scaling (e.g., normalization, standardization).

Step 4: Return Optimized Feature Set

The final optimized feature set F' is returned:

This means the refined feature set F', which is a subset or transformation of F, is ready for further processing, such as feeding into a machine learning model.

The step-3 proceeds with feature extraction. The function ***ExtractFeatures(d[])*** processes the input data block ***d[]*** and generates a feature representation, which is stored in a file named ***Features.pkl***. This step ensures that important patterns and characteristics from the dataset are captured. Once the features are extracted, the next step optimizes them using the function ***optimized(Features.pkl)***. Optimization refines the extracted features by removing redundancies, enhancing relevant patterns, and improving the model's ability to generalize across different data samples. This results in a structured feature set stored in ***Feature_set[]***. Finally, the optimized feature set is returned as output, completing the training process. These extracted features will be used in subsequent classification tasks, enabling the trained RNN model to make predictions effectively. The algorithm emphasizes efficiency by structuring the data input, systematically extracting useful features, and optimizing them before finalizing the training process. This structured approach ensures that the RNN model learns meaningful patterns from the dataset, leading to better classification performance. The selection of activation functions and threshold values plays a crucial role in determining how the network learns and adapts to the training data. By following this process, the algorithm ensures that the model is well-prepared for accurate classification tasks.

Algorithm 2: RNN Testing process for classification

Step 1: Read Test Instances

Each test instance is processed to extract its feature set from a database of test instances (TestDBList).

$$testFeature(m) = \sum nfeatureSet\big[A[i], A[i+1], \dots, A[n]\big] \leftarrow TestDBList$$

This means that for each test instance m, a feature set is collected from the database.

Step 2: Extract Feature Vector (Hot Vector Representation)

Each extracted test feature is converted into a vector representation.

$$Extracted_{FeatureSetx[t,\dots,n]} = x = 1\sum nt \leftarrow testFeature(m)$$

Extracted_FeatureSet contains the features of the respective domain. This represents encoding features into a structured format suitable for processing.

Step 3: Read Train Instances

Similar to Step 1, we extract the feature set for each training instance from a database of training instances (TrainDBList)

$$ErainFeature(m)1\sum nfeatureSet\big[A[i], A[i+1], \dots, A[n]\big] \leftarrow TrainDBList$$

Each training instance m is processed to extract its features.

Step 4: Extract Training Feature Vector

Following the same procedure as in Step 2, the features from the training instances are transformed into a structured feature set.

$$Extracted_{FeatureSetx[t,\ldots,n]} = x = 1\sum nt \leftarrow trainFeature(m)$$

Again, this step ensures that extracted features from the training set are structured similarly to the test set.

Step 5: Compute Similarity Between Test and Training Feature Sets

$$weight = calcSim(FeatureSetx \parallel i = 1\sum nFeatureSety[y])$$

Now, each test feature set is compared with all training feature sets using a similarity function.

Here, calcSim is a similarity function (such as cosine similarity, Euclidean distance, or another measure) that computes how closely the test feature set matches each training feature set.

Step 6: Return Computed Weights

Finally, the computed weights, representing the similarity scores between the test and training instances, are returned.

Return weight

This weight can be used for classification, clustering, or other decision-making processes.

The algorithm 2 begins by extracting features from test instances stored in a predefined database (TestDBList). For each test instance, a feature set is collected and represented as testFeature(m). Each extracted feature set is transformed into a structured hot vector representation or an input neuron. This ensures that the extracted features are numerically represented and can be used for further processing. The resulting feature set, Extracted_FeatureSet$x[t]$ contains the relevant features of the test instance.

Similarly, the training instances are processed to extract their respective feature sets from the training database (TrainDBList). The extracted feature set, trainFeature(m), contains structured feature information from the training data. The training feature set undergoes the same transformation into structured hot vectors. This ensures consistency in representation between test and training feature sets, making them comparable. The extracted test features are mapped to the training feature sets using a similarity function. This function, calcSim, calculates the degree of similarity between the test feature vector and all training feature vectors. This comparison is crucial for classification or clustering tasks.

Returning the Computed Similarity Weights: Finally, the similarity scores (weights) are computed and returned. These weights indicate the closeness of test instances to training instances, which can be used for decision-making in machine learning tasks such as classification or clustering.

Results and Discussion

In this experiment, the accuracy of classification of RNN (Sigmoid) has been demonstrated using a synthetic dataset. Similar experiments have been conducted with various types of cross validation, and outcomes are presented in table 1. In light of the findings of this investigation, it has been discovered that employing RNN with Sigmoid function in conjunction with 20-fold cross validation yields the greatest classification accuracy of 96.10%.

Table 1: Classification of performance for identification of software vulnerability using RNN (Sigmoid)

RNN (Sigmoid)	Fold 10	Fold 15	Fold 20
Accuracy	95.60	95.90	96.10
Precision	95.80	96.10	97.00
Recall	95.80	96.00	96.30
Micro-Score	94.70	95.90	96.05

Figure 2 illustrates how RNN function delivers considerably greater accuracy than the typical machine learning techniques during module testing.

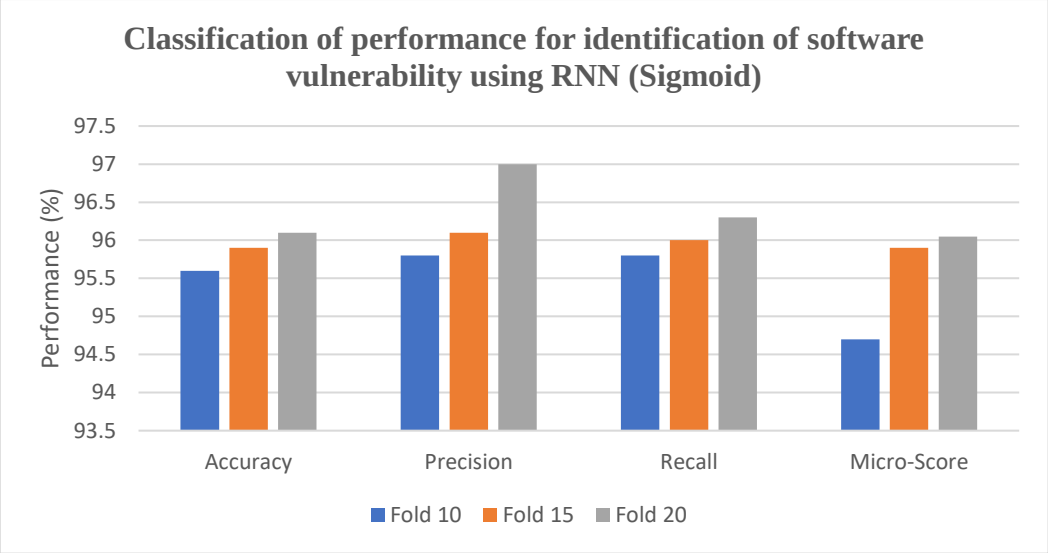

Figure 2: Detection of accuracy using RNN (Sigmoid) with 20-fold data cross validation

This is demonstrated that the 20-fold cross validation additionally achieves highest classification accuracy of 96.10% with RNN using sigmoid function.

Experiment using Recurrent Neural Network (Tanh):

The accuracy of classification of the RNN can be shown in figure 3, and similar tests have been conducted using several types of cross validation, with the outcomes being displayed in table 2.

Table 2: Classification of performance for identification of software vulnerability using RNN (Tanh)

RNN (Tanh)	Fold 10	Fold 15	Fold 20
Accuracy	96.90	97.50	97.25
Precision	97.00	97.40	97.60
Recall	97.30	97.50	97.30
Micro-Score	96.80	96.70	96.90

As a result of this investigation, it is observed that 20-fold cross validation yields the maximum classification accuracy of 97.25% for RNN utilizing Tanh.

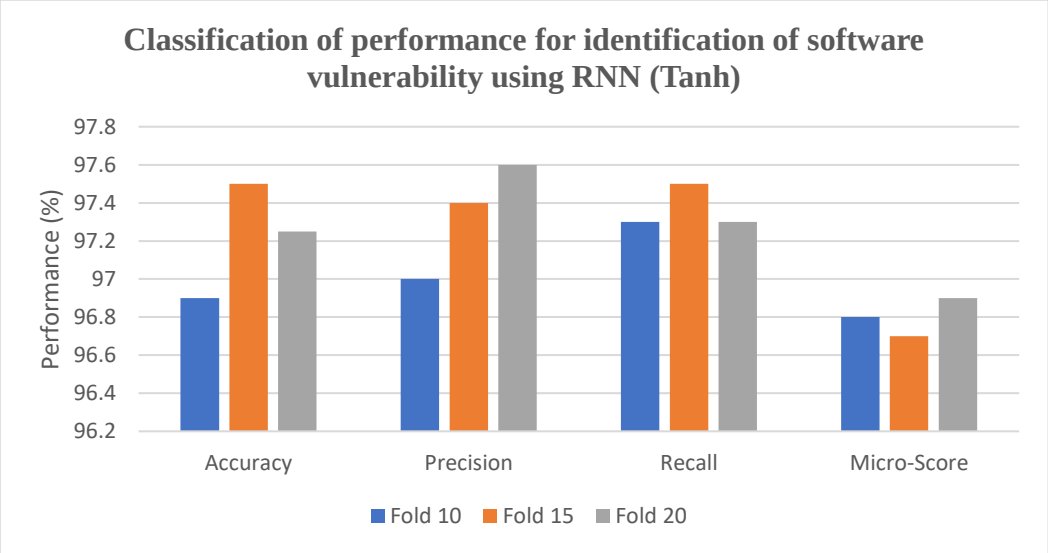

Figure 3: Detection of accuracy using RNN (Tanh) with 20-fold data cross validation

Experiment using Recurrent Neural Network (ReLU)

In this experiment, the classification accuracy of ReLU by utilizing a synthetic dataset is demonstrated. Similar evaluations have been conducted using different types of cross validation, and the outcomes are given in table 3 and figure 4.

Table 3: Classification of performance for identification of software vulnerability using RNN (ReLU)

RNN (ReLU)	Fold 10	Fold 15	Fold 20
Accuracy	97.20	97.90	97.50
Precision	97.40	96.90	97.60
Recall	95.60	97.20	97.90
Micro-Score	96.20	95.80	97.20

Based on this investigation, it is observed that the system produces the best possible accuracy of 97.5% for the 20-fold cross validation for RNN using ReLU

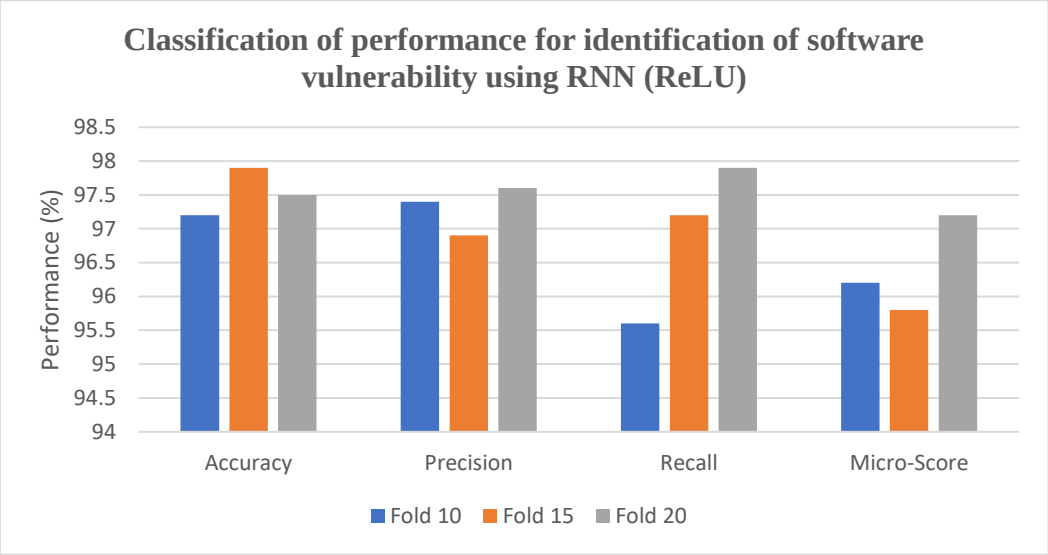

Figure 4: Detection of accuracy using RNN (ReLU) with 20-fold data cross validation

Experiments have been described above proposes a deep learning classification method in conjunction with a machine learning method. Based on the results of this investigation, it is observed that the RNN with a sigmoid activation function offers higher detection accuracy than the remaining two activation functions and the machine learning method. Each of the outcomes of the experimental tests are compared in table 4.8 and figure 4.

Table 4: Classification accuracy with 20 folds cross-validation for all methods

Method / Measure	ANN	SVM	Adaboost	RNN (Sigmoid)	RNN (Tanh)	RNN (ReLU)
Accuracy	85.60	95.2	81.30	96.10	97.25	97.50
Precision	84.99	94.80	74.50	97.00	97.60	97.60
Recall	77.72	96.2	70.30	96.30	97.30	97.90
Micro-Score	81.10	94.75	72.30	96.05	96.90	97.20

It has been found that the RNN method that was proposed achieves the highest level of predicting performance.

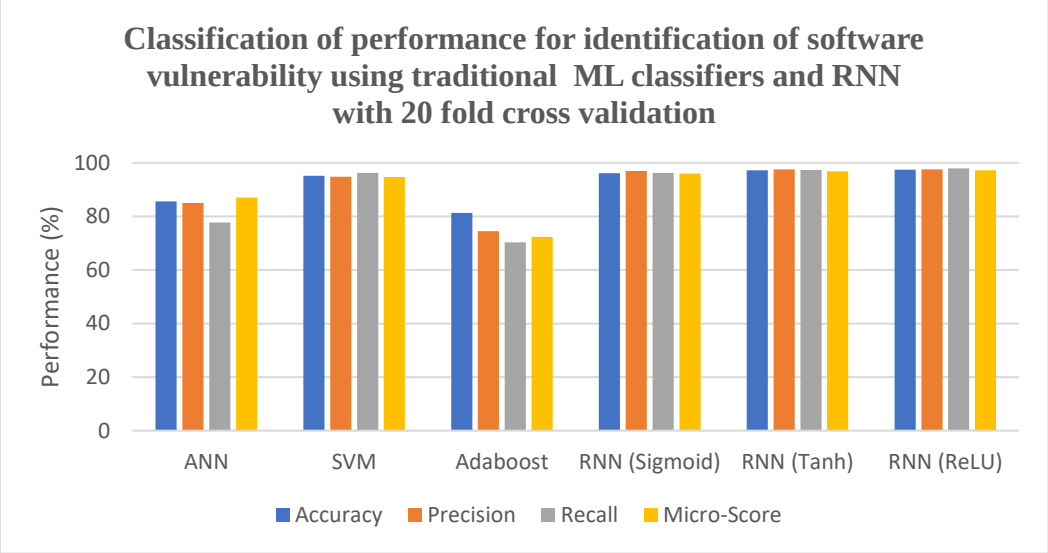

Figure 5: Classification accuracy with 20-fold cross-validation for all methods

In addition, due to the limitations imposed by the completeness and breadth of the data, many datasets do not accurately reflect all of the real software problems and new vulnerabilities. When implemented in actual software programs, the approach that was provided can be subjected to additional testing. The three data splitting method can be utilized for 10--Fold, 15--Fold, and 20-Fold cross-validation, respectively.

Conclusion

In conclusion, the proposed system successfully integrates various machine learning and deep learning techniques, particularly Recurrent Neural Networks (RNNs), to enhance the detection and classification of malicious applications. By leveraging feature extraction and selection methodologies, the model effectively identifies critical attributes from APK files, ensuring a comprehensive analysis of application behavior. The combination of supervised learning techniques and hybrid machine learning models enables accurate classification and detection of vulnerabilities within the dataset. Furthermore, the implementation of dynamic analysis techniques such as fault infusion and dynamic taint evaluation strengthens the system's ability to identify programming risks, mitigating the drawbacks associated with traditional static analysis approaches. The modular training and testing process ensures that the system efficiently processes extracted features, optimizes classification performance, and refines detection capabilities. The use of the Weka 3.7 framework for machine learning classification further enhances the model's reliability, as it facilitates a comparative analysis of multiple classification algorithms. The evaluation results demonstrate the effectiveness of the proposed RNN-based system in distinguishing between normal and malicious applications. By generating Background Knowledge (BK) during the training phase, the system improves its predictive accuracy over time, leading to more robust malware detection with 97.50% detection accuarcy. Finally, this approach provides a scalable and efficient solution for identifying security threats in Android applications. The proposed model's ability to preprocess, extract, and classify features using machine learning and deep learning techniques ensures its applicability across diverse datasets. Future improvements may focus on refining feature selection techniques and integrating real-time detection mechanisms to further enhance system performance. The combination of RNNs, hybrid machine learning, and dynamic analysis positions this methodology as a significant advancement in malware detection and vulnerability identification within software applications.

References

[1] J. B. S and K. R, "A state-of-the-art Analysis of Android Malware Detection Methods," 2022 6th International Conference on Trends in Electronics and Informatics (ICOEI), Tirunelveli, India, 2022, pp. 851-855, doi: 10.1109/ICOEI53556.2022.9777170.

[2] M. Masum and H. Shahriar, "Droid-NNet: Deep Learning Neural Network for Android Malware Detection," 2019 IEEE International Conference on Big Data (Big Data), Los Angeles, CA, USA, 2019, pp. 5789-5793, doi: 10.1109/BigData47090.2019.9006053.

[3] M. N. M. Ahmad and W. Elmedany, "A Review on Methods for Managing the Risk of Android Ransomware," 2022 International Conference on Data Analytics for Business and Industry (ICDABI), Sakhir, Bahrain, 2022, pp. 773-779, doi: 10.1109/ICDABI56818.2022.10041528.

[4] P. Kotak, S. Bhandari, A. Zemmari and J. Joshi, "Unmasking Privacy Leakage through Android Apps Obscured with Hidden Permissions," 2021 18th International Conference on Privacy, Security and Trust (PST), Auckland, New Zealand, 2021, pp. 1-5, doi: 10.1109/PST52912.2021.9647851.

[5] I. Ahmad, S. A. Ali Shah and M. Ahmad Al-Khasawneh, "Performance Analysis of Intrusion Detection Systems for Smartphone Security Enhancements," 2021 2nd International Conference on Smart Computing and Electronic Enterprise (ICSCEE), Cameron Highlands, Malaysia, 2021, pp. 19-25, doi: 10.1109/ICSCEE50312.2021.9497904.

[6] Z. Shan, I. Neamtiu and R. Samuel, "Self-Hiding Behavior in Android Apps: Detection and Characterization," 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), Gothenburg, Sweden, 2018, pp. 728-739, doi: 10.1145/3180155.3180214.

[7] C. Schindler, M. Atas, T. Strametz, J. Feiner and R. Hofer, "Privacy Leak Identification in Third-Party Android Libraries," 2022 Seventh International Conference On Mobile And Secure Services (MobiSecServ), Gainesville, FL, USA, 2022, pp. 1-6, doi: 10.1109/MobiSecServ50855.2022.9727217.

[8] L. J. Mwinuka, A. Z. Agghey, S. F. Kaijage and J. D. Ndibwile, "FakeAP Detector: An Android-Based Client-Side Application for Detecting Wi-Fi Hotspot Spoofing," in IEEE Access, vol. 10, pp. 13611-13623, 2022, doi: 10.1109/ACCESS.2022.3146802.

[9] H. C. Takawale and A. Thakur, "Talos App: On-Device Machine Learning Using TensorFlow to Detect Android Malware," 2018 Fifth International Conference on Internet of Things: Systems, Management and Security, Valencia, Spain, 2018, pp. 250-255, doi: 10.1109/IoTSMS.2018.8554572.

[10] K. Kim, J. Kim, E. Ko and J. H. Yi, "Risk Assessment Scheme for Mobile Applications Based on Tree Boosting," in IEEE Access, vol. 8, pp. 48503-48514, 2020, doi: 10.1109/ACCESS.2020.2979477.

[11] C. Zhang, S. Yin, H. Li, M. Cai and W. Yuan, "Detecting Android Malware With Pre-Existing Image Classification Neural Networks," in IEEE Signal Processing Letters, vol. 30, pp. 858-862, 2023, doi: 10.1109/LSP.2023.3294695.